SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–31
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

RepOK-Based Reduction of Bounded-Exhaustive Testing

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

[1]*Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina.*
[2]*Department of Computer Science, University of Illinois at Urbana-Champaign, USA.*
[3]*Departamento de Ingeniería Informática, Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina.*
[4]*Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina.*

SUMMARY

While the effectiveness of bounded-exhaustive test suites increases as one increases the *scope* for the bounded-exhaustive generation, both the time for test generation and for test execution grow exponentially with respect to the scope. In this article, a set of techniques for reducing the time for bounded-exhaustive testing, by either reducing the generation time or reducing the obtained bounded-exhaustive suites, are proposed. The representation invariant of the software under test's input, implemented as a `repOK` routine, is exploited for these reductions in two ways: *(i)* to factor out separate representation invariants for disjoint substructures of the inputs, and *(ii)* to partition valid inputs into equivalence classes, according to how these exercise the `repOK` code. The first is used in order to split the test input generation process, since disjoint substructures can be independently generated. The second is used in order to reduce the size of a bounded-exhaustive test suite, by removing from the suite those tests that are equivalent to some tests already present in the suite.

Received . . .

1. INTRODUCTION

A major challenge in Software Engineering is the development of methodologies and techniques to ensure the correctness of software systems, i.e., to provide guarantees that a given system correctly fulfils the purpose for which it was built [10]. Testing plays an important role in ensuring correctness, as it is, despite its inherent incompleteness, a very effective and widely used mechanism for software verification. This technique essentially consists of executing a piece of software, whose correctness needs to be assessed, in a number of different situations, or *test cases*. These cases often correspond to instantiating parameters of the software with different inputs, and in order to increase the chances of detecting bugs, one typically seeks these inputs to be as many and as varying as possible, so that the software under test is more thoroughly exercised [29].

*Correspondence to: Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto. Ruta Nac. No. 36 Km. 601, Río Cuarto (5800), Argentina. E-mail: vbengolea@dc.exa.unrc.edu.ar

Since inputs are a primary means for executing software in different scenarios, test-input generation is a highly relevant task in testing. While it may be easy for basic datatypes and when the programs to test are simple, for most programs test input generation is in general a difficult task: one often has to come up with inputs satisfying complex constraints in order to make the software run through particular paths or exhibit particular behaviours. Test generation in general, and test input generation in particular, have traditionally been done manually, but in the last few years, various approaches and tools have been developed to perform *automated* test generation, including techniques based on random generation [7, 21], SAT/SMT constraint solving [26, 1], and different forms of model checking and search [5, 11, 15, 27].

A particularly challenging problem in the context of automated test generation is that of systematically producing test inputs for programs that manipulate complex data structures, like directed graphs, linked lists or balanced trees. This is a difficult problem because this kind of input is typically required to satisfy structural constraints such as balance, acyclicity, connectedness, etc., to be valid. An approach that has been very successful to test programs handling complex data structures (as well as programs in other domains with similar characteristics) is *bounded-exhaustive testing* [5, 15, 18, 11, 8]. This technique consists of generating *all* the inputs that satisfy the constraints corresponding to the wellformedness of the generated structures, within certain prescribed bounds, or *scopes*. Tools following this approach usually involve some form of constraint-solving, e.g., based on search, model checking, or combinations of these.

The rationale behind bounded-exhaustive testing dwells on the *small-scope hypothesis* [12], which conjectures that in some contexts, if a program has bugs, then most of these bugs can be reproduced using small inputs. There are situations, however, where larger scopes are necessary to achieve coverage and detect bugs. For instance, some insertion and deletion procedures in balanced trees require structures of larger sizes to force rotations or enable other rebalancing mechanisms; so, if one wants to cover the rebalancing cases, the smaller scopes are insufficient. But, while the effectiveness of bounded-exhaustive test suites increases as one increases the scope for the bounded exhaustive generation, both the time for test generation and the time for test execution grow exponentially with respect to the scope. More precisely, in many cases, considering a particular scope for bounded exhaustive testing might be necessary to cover certain input classes of interest, but it may also require excessive time for test generation. Moreover, even in situations where the generation process can be completed, the obtained suite may be so large that the time required to execute it would be prohibitive. For instance, if one wants to test a merge routine on binomial heaps, building its corresponding bounded exhaustive test-suite bounding by 7 nodes both binomial heaps (merge is a binary operation on heaps) takes over 15 hours on a modern workstation, and has 11,538,197,056 tests; despite the fact that one may be willing to spend 15 hours for the generation, actually testing the routine on such large suite is impractical.

In this paper, a set of techniques are proposed, to overcome the above described problems. These techniques aim at reducing the time for bounded exhaustive testing, by either reducing the test generation time, or adequately reducing the obtained bounded exhaustive suites. They do so by exploiting the *representation invariant*, i.e., the constraint that indicates whether a structure is well-formed or not, of the inputs of the software under test. More precisely, the presented techniques require the representation invariant to be provided *imperatively*, as a `repOK` routine [16], which is used for two tasks:

- to factor out separate representation invariants for disjoint substructures of the inputs, and
- to partition valid inputs into equivalence classes, according to how these exercise the `repOK` code.

The first of the above tasks aims at improving the generation of bounded exhaustive suites, and applies when inputs are composed of disjoint substructures. These are treated separately, by independently generating suites for each of the (disjoint) substructures. This approach has various advantages, especially because tools for bounded exhaustive test generation typically have their efficiency tied to the way in which `repOK` is implemented. In particular, if the inputs are composed of two separate substructures s_1 and s_2, then checking the overall wellformedness by checking first the wellformedness of s_1 and then that of s_2 can be completely different, from the point of view of efficiency, from checking first s_2 and then s_1. If the generation of the substructures are independent, the effort of `repOK` "engineering" is reduced for efficiency purposes. Moreover, disjoint substructures can be exploited for parallelising the generation stage, and even if these are sequentially generated, the problem of reconsidering every valid structure of the second substructure for each valid one of the first substructure, is avoided.

The second of the above mentioned tasks aims at reducing the time for testing, by adequately reducing bounded exhaustive suites. This approach works on the observation that the `repOK` code provides information regarding the "variability" of the inputs. Essentially, if one considers a white box coverage criterion on the `repOK`, one can define an equivalence relation between valid test inputs. Different valid test inputs will be considered equivalent if they exercise the `repOK` code in a similar way, according to the selected 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. This proposal corresponds to 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, this 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).

In summary, the above mentioned tasks contribute to bounded exhaustive testing in the following way. Firstly, by separately generating disjoint substructures one reduces the space of candidate inputs to be considered, and therefore also the time spent in input generation. This separate generation is applicable in cases in which inputs are composed of disjoint substructures; it produces bounded exhaustive suites, i.e., it does not affect the produced suites with respect to standard bounded exhaustive generation, but it produces these suites more efficiently. On the other hand, the other introduced technique aims at reducing the time spent in testing by reducing the sizes of already generated bounded exhaustive suites, according to a selected white-box testing criterion over the `repOK` code of the structure under test.

To assess the effectiveness of these techniques, some case studies are carried out. These case studies show that, for structures composed of non trivial disjoint substructures, the generation time can be significantly improved by independently generating the substructures. Regarding the second introduced technique, the case studies show that when some `repOK`-based input equivalences are used to reduce the sizes of bounded-exhaustive test suites up to two orders of magnitude, the test suites obtained have an effectiveness comparable to that of the corresponding "full" bounded-exhaustive test suites, in terms of their mutant-killing ability.

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* [29, 9]; 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 code of 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, namely the *generative* approach and the *filtering* approach [11]. 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 [6], 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 [16].

Given a class C, newly created objects of C must satisfy the representation invariant, i.e., their public constructors must ensure the representation invariant holds when they terminate. Also, public methods that modify objects (e.g., insertion and deletion routines) of class C must preserve the representation invariant, i.e., assuming that the representation invariant holds before calling the method, this method must ensure that the representation invariant also holds when it terminates. As put forward by Liskov et al. [16], developers should equip their complex structures' implementations with `repOK` routines, since these routines will greatly help in debugging the implementations. These `repOK` routines can be called in tests, to evaluate the fact that certain methods establish or preserve the invariant, or directly within a class' methods and constructors, just before they return, as described above.

Bounded-Exhaustive Testing. Bounded-exhaustive testing is a testing technique that has proved useful in certain testing contexts, in particular, testing programs that manipulate complex data structures. Examples of such programs 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 scope, consisting of a bound on the size of inputs (maximum number of objects for class-based types, ranges for basic datatypes), all valid inputs whose size lies within the scope, 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 structures. Thus, by testing a program on all possible input structures bounded in size by some relatively small scope one would be able to exhibit many bugs.

3. INDEPENDENT GENERATION OF DISJOINT SUBSTRUCTURES

In this section, a technique to build bounded exhaustive test suites, by reducing the state space explored during generation and based on the use of the `repOK` of the structure, is presented. This technique fits better with filtering approaches to test generation, since in these contexts having a representation invariant is often a requirement. In particular, this technique applies to the cases in which the code under analysis manipulates complex structures made up of parts allocated in disjoint portions of the heap. Roughly speaking, the proposed technique consists of generating the disjoint parts of the given structure separately, by factoring out separate representation invariants for disjoint substructures of the inputs, which are then put together to obtain the final structure.

The technique is based on the observation that, during the generation process, many ill-formed structures are produced by combining well-formed parts with ill-formed ones. The proposed approach reduces the candidate state space by avoiding these kinds of ill-formed structures by individually constructing sub-structures that are disjoint in the heap, and then combining them to get well-formed complete structures. In this way, only well-formed sub-structures are combined. In order to describe more precisely the technique, let C be a class for which a bounded exhaustive suite with scope k has to be built, and let `repOK` be C's imperative representation invariant. The proposed technique consists of the following steps:

- Identify substructures $s_1, s_2, \ldots, s_n$ of instances of class C, which are *disjoint*, in the sense that these are always allocated in independent parts of the heap in instances of C that satisfy `repOK`, and which, put together, conform C (i.e., $s_1, s_2, \ldots, s_n$ are a *partition* of C).
- Define auxiliary data structures $S_1, S_2, \ldots, S_n$ to represent the disjoint substructures previously identified.
- From `repOK`, obtain representation invariants $\texttt{repOK}_1, \texttt{repOK}_2, \ldots, \texttt{repOK}_n$ such that, if an instance c of C satisfies `repOK`, then the "subinstance" c_{s_i} of c, corresponding to substructure s_i, satisfies $\texttt{repOK}_i$.
- Generate independently bounded exhaustive suites for $S_1, S_2, \ldots, S_n$, with scope k.
- Combine instances of the bounded exhaustive suites constructed previously, to build instances of C.
- Filter out those instances of C that do not satisfy `repOK`.

A simple example to illustrate the technique is the following. A data structure available as part of Apache Commons collections is *node caching linked list*. This data structure corresponds to an implementation of lists that tries to reduce object creation and garbage collection by maintaining a list of deleted nodes in a cache. Essentially, the structure consists of a dummy circular doubly linked list, the actual contents of the list, and a singly linked list of cached nodes. The structural definition of `NodeCachingLinkedList`, including the `LinkedListNode` class used in the structure, is shown in Figure 1.

Now suppose that a developer needs to test a routine manipulating a node caching linked list, and that he/she decides to do so by bounded exhaustive testing. That is, the routine will be tested for all valid node caching linked lists, within certain scope (bound in the number of nodes, ranges for values, etc). Basically, the objects to be produced are composed of a dummy circular doubly linked list (referenced by a `header`), and a singly linked list of cached nodes (referenced by a `firstCachedNode` header). Notice that these two substructures are *disjoint*, i.e., in every valid instance of the class, no node belongs to both lists.

```
public class NodeCachingLinkedList {
    private  LinkedListNode header;
    private int size;

    private  LinkedListNode firstCachedNode;
    private int cacheSize;
    private int maximumCacheSize;
    ...
}

public class LinkedListNode {
    Object value;
    LinkedListNode previous;
    LinkedListNode next;
    ...
}
```

Figure 1. Structural definition of classes `NodeCachingLinkedList` and `LinkedListNode`.

Consider the following generation scenario: All the instances of `NodeCachingLinkedList` containing up to 8 nodes, with size up to 3 for the dummy circular doubly linked list and size up to 4 for the cache, are generated using a bounded exhaustive test input generation tool. Notice that the additional node, with respect to the sum of the sizes of the two lists, has to do with the fact that the circular doubly linked list has a dummy head node, which is not taken into consideration in the corresponding size fields. In this generation using the tool Korat, 13,164 candidate structures are explored, 450 of which are valid. Many of the invalid visited candidates are composed of a valid dummy circular doubly linked list with an invalid cache, or vice versa. Figure 2 shows such a candidate that is visited during the bounded exhaustive generation, where a well-formed list of size 3 (plus a dummy header) is set in combination with an invalid cache list (the cache list should be singly linked, i.e., all `previous` references must be set to `null`, which is not the case in this example). This kind of invalid candidates, composed of valid substructures combined with invalid ones, are rather common in structures with disjoint substructures. Moreover, their number increases as the scope is increased, and is an important source of inefficiency in the bounded exhaustive generation process.

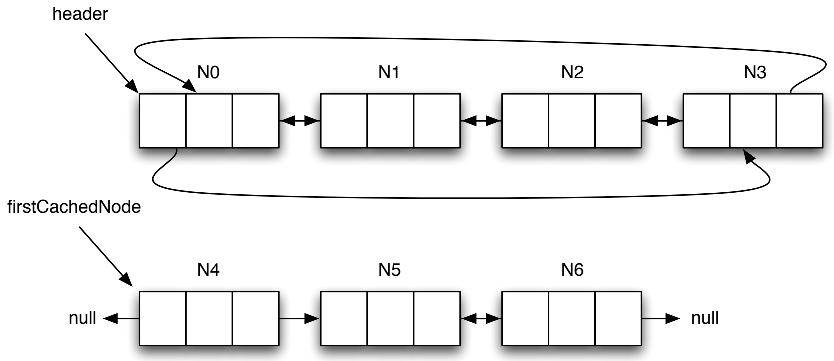

Figure 2. Ill-formed instance of `NodeCachingLinkedList`.

If it is known that different parts of a structure are always disjoint, as in the above case, one may take the representation invariant for the whole structure, and *slice* the `repOK` code in order to obtain independent "smaller" representation invariants that predicate on the disjoint portions of the structure. These `repOK`'s on smaller structures can be used to independently generate the substructures, which are then combined to form instances of the whole structure.

Some technical questions arise at this point. First, how can one determine whether every instance of the data structure is *always* composed of disjoint substructures? This information may be obtained from different sources, the most obvious being that it is provided by the developer, i.e., the developer is in charge of detecting which parts of the structure may be generate separately, and provides this information as an input of the separate generation process. For instance, for the case of `NodeCachingLinkedList`, the required information is that the *cache* list referenced by field `firstCachedNode` is disjoint from the structure referenced by field `header`, i.e., the *circular double linked list*. Another way of obtaining this information is from the *scope definition*. Basically, providing the scope in order to perform bounded exhaustive test generation consists of building *domains* for different fields in the structure to be generated. If disjoint domains are provided for fields of the same type, one can take advantage of this information for processing the `repOK`, as indicated above. As an example, suppose that bounds on the number of objects and possible values for fields, to be used to build instances of the structure, are given using a programmatic notation. Following the `NodeCachingLinkedList` example, one may define the scope as follows:

```
...
Domain EntriesList = createDomain(LinkedListNode , numEntry)
EntriesList.setNullAllowed();
Domain EntriesCache = createDomain(LinkedListNode , numEntry)
EntriesCache.setNullAllowed();
setScope(header, EntriesList);
setScope(header.*next, EntriesList);
setScope(header.*previous, EntriesList);
setScope(firstCachedNode, EntriesCache);
setScope(firstCachedNode.*next, EntriesCache);
setScope(firstCachedNode.*previous, EntriesCache);
...
```

In this case, two domains are defined, namely `EntriesList` and `EntriesCache`, both composed of `numEntry` instances of `LinkedListNode` (plus `null`). Then, the scopes for different fields are defined, in this case clearly stating that the fields of the lists referenced by `header` and `firstCachedNode` do not share objects. One may exploit this valuable information in the detection of disjoint portions of the structure.

Finally, the information about disjoint parts of a structure may be present in the `repOK`, either explicitly or implicitly. For instance, for the case of `NodeCachingLinkedList`, the representation invariant may include checking that the cache and the circular list do not share nodes.

Another important technical question has to do with how `repOK` may be automatically processed in order to obtain separate representation invariants for the disjoint substructures. This is not a trivial issue, and it is clear that, in many cases, the original `repOK` may not be fully "modularised": even though by slicing the original one may obtain useful "local" representation invariants, there might be "global" wellformedness conditions that have to do with the relationship between the disjoint substructures (notice that this global representation invariant is in fact included in the steps of the technique). For instance, if `NodeCachingLinkedList` was used to implement a set, and the cache was checked for membership before an insertion (so that creating a new node is avoided), then part of the `repOK` might state that the cache and the circular linked list do not share stored values. In many cases this separation process is straightforward. Consider, for instance, a routine that one wishes to test that receives as parameters more than one structure. Some examples of this situation are a merge routine for binomial heaps (takes two binomial heaps), set union/intersection for set implementations, and even membership, insertion and deletion routines, which usually take two parameters, the collection to be modified or queried, and the value to insert, delete or search for. In all these cases, the input is *composite*, and the `repOK` for the input is the conjunction of the `repOK`'s for the parameter structures.

These issues, although important, are beyond what is studied in this paper. For the first, it is simply assumed that the information about disjoint substructures is provided by the developer. For the second, the processing of `repOK` performed in this article for the purpose of experimentation is a straightforward slicing that examines which part of the structure being visited is relevant to each sentence in the processed representation invariant. Thus, besides the "local" `repOK` methods constructed for each substructure, a "global" one is obtained composed of sentences in which more than one disjoint substructure may be involved.

As an example, consider a representation invariant for `NodeCachingLinkedList`, shown in Figure 3. This `repOK` checks whether a structure is a well-formed node caching linked list by checking various things: *(i)* that the list is a well-formed circular doubly linked list, *(ii)* that cache size does not exceed the maximum size, *(iii)* that the size of the list is consistent with the number of nodes in it, *(iv)* that the cache list is acyclic, and *(v)* that the size of the cache is consistent with the number of nodes in the list. This `repOK` could be split in two: a part referring only to the circular list (and its corresponding size field), and another one referring only to the cache list (and its corresponding size field). These two "local" `repOK`'s are shown in Figure 4.

Later on in this paper, this technique is assessed for a number of case studies.

```
public boolean repOK() {
   if (header == null) return false;
   if (header.next == null) return false;
   if (header.previous == null) return false;
   if (cacheSize > maxCacheSize) return false;
   if (MAXIMUM_CACHE_SIZE != 20) return false;
   if (size < 0) return false;

   int cyclicSize = 0;
   Node n = this.header;
   do{
       cyclicSize++;
       if (n.previous == null) return false;
       if (n.previous.next != n) return false;
       if (n.next == null) return false;
       if (n.next.previous != n) return false;
       if (n != null)  n = n.next;
   } while (n != header && n != null);

   if (n == null) return false;
   if (size != cyclicSize - 1) return false;
   int acyclicSize = 0;
   Node m = firstCachedNode;
   Set<Node> visited = new HashSet<Node>();
   visited.add(firstCachedNode);
   while (m != null){
       acyclicSize++;
       if (m.previous != null) return false;
       if (m.value == null) return false;
       m = m.next;
       if (!visited.add(m)) return false;
   }
   if (cacheSize != acyclicSize) return false;
   return true;
}
```

Figure 3. Representation invariant for class `NodeCachingLinkedList`.

3.1. On the Correctness of the Technique

The correctness of separate generation of disjoint substructures is now discussed. The technique is sound and complete with respect to bounded exhaustive generation, in the sense that an instance is produced by bounded exhaustive generation for scope k if and only if this instance is produced by the separate generation approach, also for scope k. The proof of this fact is relatively straightforward. Notice that soundness is trivial due to the last step in the separate generation approach, that filters out from the bounded exhaustive suite produced by the separate generation approach those instances that do not satisfy `repOK`. Regarding completeness, the argument is the following. Let c be an instance of class C that satisfies `repOK`, and that is within scope k. For each substructure S_i of C, there is an instance c_{s_i} subsumed in c, that is within scope k, too (since it is a subinstance of c, which is bounded by k). Moreover, since c satisfies `repOK`, c_{s_i} satisfies `repOK`$_i$ (see step 3 in the separate generation approach). Then, instance c_{s_i} is produced as part of the bounded exhaustive generation for substructure S_i. Since this is a fact for every substructure S_i, $c_{s_1}, \ldots, c_{s_n}$ are built by the corresponding bounded exhaustive "local" generations. The combination of $c_{s_1}, \ldots, c_{s_n}$ corresponds to c ($S_1, \ldots, S_n$ are a partition of C), which satisfies `repOK`, and therefore is produced by the separate generation approach.

```
public boolean repOK() {
    if (header == null)
        return false;
    if (header.next == null)
        return false;
    if (header.previous == null)
        return false;
    if (size < 0)
        return false;
    int cyclicSize = 0;
    Node n = header;
    do{
        cyclicSize++;
        if (n.previous == null)
            return false;
        if (n.previous.next != n)
            return false;
        if (n.next == null)
            return false;
        if (n.next.previous != n)
            return false;
        if (n != null)
            n = n.next;
     }while(n!=header && n!=null);
     if (n == null)
        return false;
     if (size != cyclicSize - 1)
        return false;
     return true;
}
```

```
public boolean repOK() {
   if (cacheSize > maxCacheSize)
      return false;
   if (MAXIMUM_CACHE_SIZE != 20)
       return false;
   int acyclicSize = 0;
   Node m = firstCachedNode;
   Set<Node> visited = new HashSet
       <Node>();
   visited.add(firstCachedNode);
   while (m != null){
       acyclicSize++;
       if (m.previous != null)
           return false;
       if (m.value == null)
           return false;
       m = m.next;
       if (!visited.add(m))
           return false;
   }
   if (cacheSize!= acyclicSize)
       return false;
   return true;
}
```

Figure 4. Representation invariants for the disjoint substructures of `NodeCachingLinkedList`.

4. REDUCING BOUNDED-EXHAUSTIVE TEST SUITES

As it was previously described, bounded-exhaustive testing forces the tester to use very small bounds in some cases, because the sizes of bounded-exhaustive suites rapidly become too large, and consequently using these exhaustively for testing becomes in many cases impractical. In this section, an approach to help in reducing bounded-exhaustive test suites is presented. The approach assumes, as for the technique presented in the previous section, the availability of an imperative implementation of the representation invariant of the structure for which the bounded-exhaustive suite was produced. Thus, this technique also fits better with filtering approaches to test generation, as the one presented in the previous section.

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 equivalence relations on the set of 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.

It is worth to remark that this criterion defines a mechanism to consider inputs to be equivalent that takes into account the structure of the `repOK`, but disregards the code under test. This makes the approach black box, although it employs white box criteria on the code of the `repOK` routine. Since the technique is based on the `repOK` as a specification of the inputs, it assumes this routine to be correct, i.e., to faithfully capture the constraints that make inputs valid.

To describe how the technique works, it is essential to describe how coverage criteria using `repOK` are defined. Let C be a class, and let `repOK` be the imperative implementation of its representation invariant. As an example to drive the presentation, consider the Java classes, implementing binary trees of integers, given in Figure 5. The representation invariant for this class must 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 method from class `BinaryTree`, taken from the examples distributed with the Korat tool [5], shown in Figure 6.

```
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
   ...
}
```

Figure 5. Partial Java definition of binary trees.

```
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);
}
```

Figure 6. Imperative representation invariant for class `BinaryTree`.

Now suppose that one needs to test a routine that receives as a parameter a binary tree, e.g., a binary tree traversal routine. Notice that, as a (black-box) criterion for testing the traversal routine, a partition of all possible binary tree structures can be defined 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 similar, and therefore can be thought of as corresponding to the same class.

Still it is necessary to define what "exercise in a similar way" means. This can be done, in principle, by choosing any white-box coverage criterion, to be applied to `repOK`. For instance,

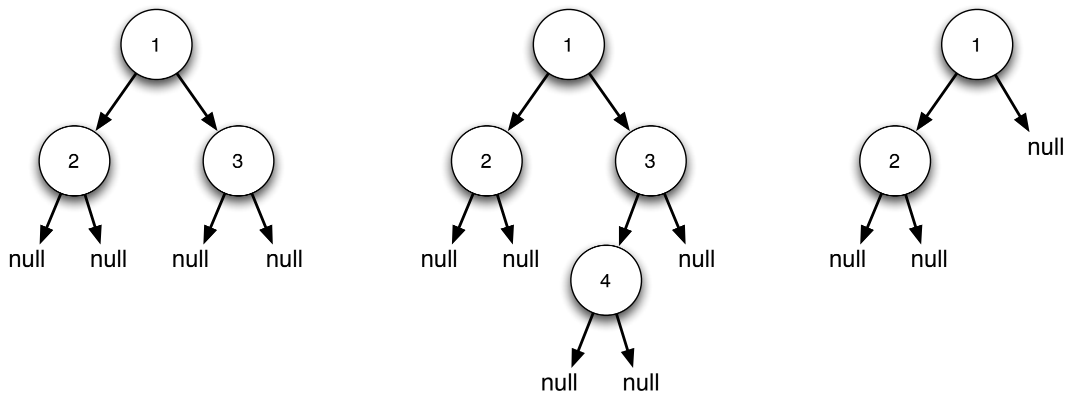

Figure 7. Sample binary trees. The first and second exercise `repOK` in the same way, according to decision coverage.

decision coverage on `repOK` can be considered; 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 three trees in Figure 7, 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 the other two, predicate `current.getRight() != null` never evaluates to true in this case). Notice that the decision points in the code under test are not taking in account for this criterion.

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, $[\![c]\!]^{P}_{Crit}$ denotes 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*. The technique defined in this section works by defining equivalence relations on the set of 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, c_1 is considered to be equivalent to c_2 (according to `repOK` under *Crit*), if and only if $[\![c_1]\!]^{\texttt{repOK}}_{Crit} = [\![c_2]\!]^{\texttt{repOK}}_{Crit}$.

In the above example, one of the simplest white-box coverage criteria was considered 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 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, for instance $N/10$. In this case, the following steps can be followed:

- 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 it was mentioned above, 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 this approach may result in different reduced suites.

In the section 5, the technique presented in this section will be assessed, as well as the one presented in the previous section, for a number of case studies.

4.1. *On the Correctness of the Technique*

As opposed to the technique introduced in the previous section, the `repOK`-based reduction of bounded exhaustive suites is sound, but not complete, with respect to bounded exhaustive generation. Again, soundness is relatively trivial, since `repOK`-based reduction works by first generating a bounded exhaustive suite, and then filtering out some cases according to some coverage criterion and the maximum number of test inputs to be considered per equivalence class.

The technique is also complete with respect to equivalence class coverage, in the sense that every equivalence class q coverable within scope k will be covered by the reduced suite, as long as the value max_q (maximum number of tests for equivalence class q) is greater than zero for every equivalence class q. In this respect, consider the following argument. Let B be a bounded exhaustive suite for class C and scope k. Let *Crit* be a white-box criterion to apply over `repOK`, the representation invariant for class C. Assume that there exists an equivalence class q for which its maximum is greater than zero, which is covered by B, but not covered by the reduced suite B_{Crit}. Then, there exists an instance c of C which is present in B, which covers class q and obviously is not in B_{Crit}. Moreover, since q is not covered by B_{Crit}, there is no input in B_{Crit} corresponding to equivalence class q. Since c was left out of B_{Crit}, it was removed during the suite reduction process over B. This can only happen if the limit max_q of instances was met in the reduction, meaning that we already have inputs in B_{Crit} that cover class q (since max_q is positive), thus arriving to a contradiction. Therefore, `repOK`-based reduction is complete with respect to equivalence class coverage.

5. EXPERIMENTAL EVALUATION OF THE TECHNIQUES

Various case studies for assessing the techniques introduced in the previous two sections are described below.

For the case of independent generation of disjoint substructures, the evaluation is based on three case studies, corresponding to analyses of routines on selected heap-allocated data structures, namely *binomial heaps*, *node caching linked lists*, and *AVL trees*. The selected routines for these structures correspond to `merge` of two *binomial heaps*, `addAll`, an operation that adds all the items in a given *list* to an *AVL tree*, and `getFirst` on *node caching linked lists*.

For the case of `repOK`-based test suite reduction, the analyses are based on several routines on *binomial heaps*, *binary search trees*, *doubly linked lists*, and *red black trees*, and different coverage criteria on `repOK`, in order to perform the filtering. Three coverage criteria were selected: *decision coverage*, *path coverage* and a variant of decision coverage, which the authors of this paper believe to be useful in the context of bounded exhaustive test suite filtering. This criterion, which is referred to as *counting decision coverage* (CDC), takes into account the number of times each decision in a program evaluates to true and false.

When available, implementations of the above mentioned structures provided in the Roops benchmark [22] were used.

The first presented technique deals with bounded exhaustive generation. The approach is relevant for any bounded exhaustive test generation tool, in particular for those based on a filtering approach. The experiments were carried out using the tool Korat [5]. The second technique does not deal with bounded exhaustive generation, but with *reducing* already computed bounded exhaustive suites. It is worth mentioning however that bounded exhaustive suites for various scopes, on which filtering is applied, were generated also using Korat.

5.1. Case Studies for Separate Generation of Disjoint Substructures

For each of the structures for which separate generation is evaluated, the `repOK` code was taken and automatically split it into various `repOK`'s asserting over disjoint portions of the structure, or over "disjoint" parameters (different parameters of a routine) as in the cases of merge of *binomial heaps* and `addAll` of *AVL trees*.

For the three selected cases studies we get different local `repOk`'s predicating on separated portions of the original structures. These `RepOK`'s were used in order to automatically generate inputs using Korat [5], in a bounded exhaustive fashion and up to a number of different bounds to analyse scalability.

For these experiments, the processing of `repOK` is a straightforward slicing with respect to the part of the structure being visited by each sentence in the processed `RepOK` code. More precisely, the slicing achieved on `repOK` is based on the *definition-use graph* [14] built on the `repOK` implementation. This graph is constructed from the *control flow graph* of the `repOK` code incorporating information regarding which variables are used (and defined) in each node and edge of the graph. Then, this graph is used to compute all statements in `repOK` that may affect the value of some given variables, related to some part of the structure.

Besides the "local" `repOK`'s, a "global" one is obtained taking those sentences in which more than one disjoint substructure may be involved.

Since we used Korat, we obtained candidates vectors representing substructures. This valid candidate vectors returned by Korat on different substructures were saved to subsequently combine them to build candidate vectors (and objects from them) corresponding to the whole structure, and to check, when necessary, the "global" `repOK` (if further constraints, besides those sliced in the representation invariants for the substructures, had to be checked). For each case study, the

Scope	BE	Korat		Disj. Generation	
		Explored	Time	Explored	Time
4,1,2,2	132	1485	0.207 s	183	0.42 s
6,2,3,2	1014	13,610	0.309 s	859	0.45 s
8,3,4,2	6840	102,426	0.654 s	3254	0.524 s
10,4,5,2	43,560	698,155	1.598 s	11,024	0.65 s
12,5,6,2	269,724	4,433,071	6.872 s	34,649	1.021 s
14,6,7,2	1,646,058	26,602,064	43.54 s	102,974	2.317 s
16,7,8,2	9,967,920	152,594,160	274.444 s	292,988	9.534 s
18,8,9,2	60,108,828	844,607,873	1768.611 s	805,192	54.092 s

Table I. Bounded exhaustive vs. separate bounded exhaustive generation of disjoint substructures for `NodeCachingLinkedList`, as the number of nodes increases.

time needed to perform this process was measured, including the time required to build all the (sub)structures, plus the time required to combine them.

Finally, the developers are in charge of providing the information about which parts of the structure are disjoint. That is, the process used to generate structures by separated generation assumes, in these experiments, that this information is provided as an input.

5.1.1. Node Caching Linked Lists. The first case study involves bounded exhaustive test generation for `NodeCachingLinkedList`, the caching circular doubly linked list presented in Section 3. Tables I and II show, for different scopes (each scope specifies the maximum number of nodes in the structure, the maximum size of the dummy circular doubly linked list, the maximum size of the cache and the number of keys allowed in the structure), the sizes of bounded-exhaustive suite (BE), the number of explored candidates (Explored) and the time expended during generation (Time). This information is provided both for bounded exhaustive generation and for *disjoint generation*, i.e., the independent generation (and *a posteriori* combination) of the disjoint substructures of `NodeCachingLinkedList`. Notice that the number of valid inputs is reported only once, since it is the same for both cases. These two tables show how the number of visited structures, and the time it takes to perform the visit, grows as the scope is increased, in two different dimensions. Table I shows how these numbers change as the number of nodes is increased, while maintaining the number of keys, whereas Table II shows how these numbers progress as the number of keys is increased, maintaining the number of nodes. As it can be seen, the disjoint generation technique proves effective very quickly (the highlighted rows of the tables indicate the cases in which the presented technique outperforms, with respect to time, traditional bounded exhaustive generation). In this case study, it is self evident that the technique provides a significant advantage with respect to the number of explored candidates. For instance, for scope 16,7,8,2, the 152,594,160 candidates explored by bounded exhaustive generation are reduced to 292,988 visited candidates with the presented technique, with a significantly smaller time consumption (274.444 s vs. 9.534 s). Figures 8 and 9 provide a more graphical representation of the information in Tables I and II, showing how, for this case study, this technique increases scalability in the generation.

5.1.2. Binomial Heaps. This case study corresponds to bounded exhaustive test generation of pairs of binomial heaps. As it was mentioned previously, this kind of bounded exhaustive generation is

Scope	BE	Korat		Disj. Generation	
		Explored	Time	Explored	Time
8,3,4,2	6840	102,426	0.654 s	3254	0.525 s
8,3,4,3	60,860	902,178	1.634 s	11,604	0.955 s
8,3,4,4	353,340	5,101,182	6.216 s	31,094	0.952 s
8,3,4,5	1,515,150	20,787,504	22.392 s	68,948	1.754 s
8,3,4,6	5,222,000	67,420,914	70.544 s	134,214	3.604 s
8,3,4,7	15,289,560	185,298,006	192.722 s	237,764	8.339 s
8,3,4,8	39,475,620	449,407,158	477.14 s	392,294	19.799 s
8,3,4,9	92,246,330	988,610,772	1018.966 s	612,324	48.154 s

Table II. Bounded exhaustive vs. separate bounded exhaustive generation of disjoint substructures for `NodeCachingLinkedList`, as the number of keys increases.

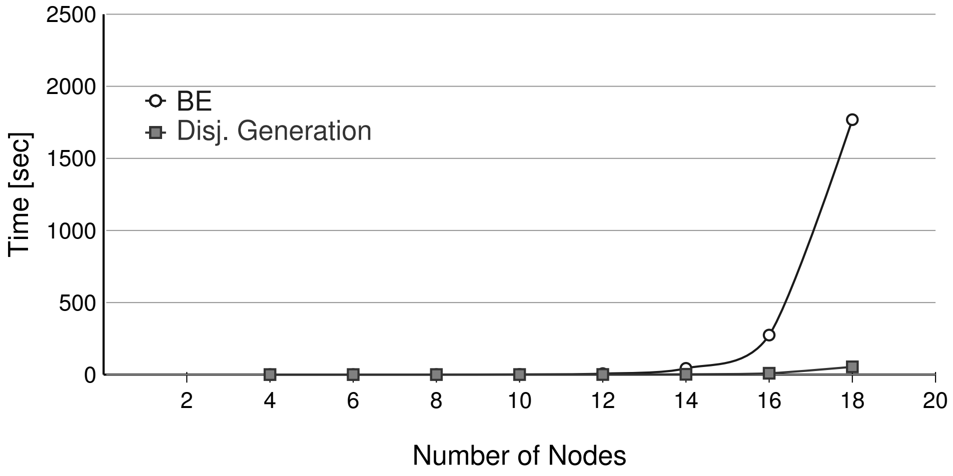

Figure 8. Bounded exhaustive vs. separate bounded exhaustive generation of disjoint substructures for `NodeCachingLinkedList`, as the number of nodes increases (graphical representation).

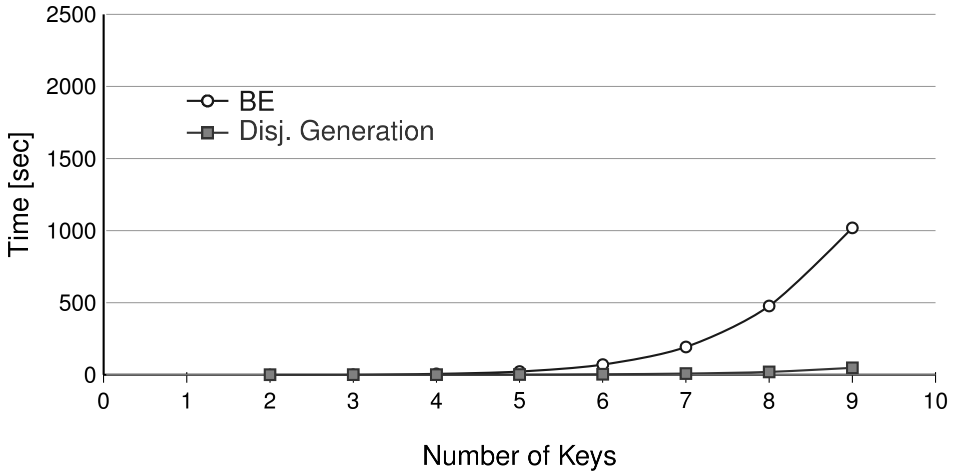

Figure 9. Bounded exhaustive vs. separate bounded exhaustive generation of disjoint substructures for `NodeCachingLinkedList`, as the number of keys increases (graphical representation).

typical of testing situations involving routines receiving more than one structure as a parameter (e.g., union and intersection routines in set implementations). For the particular case of binomial heaps, a relevant 'binary' routine is `merge`, a routine manipulating pairs of 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. As it was mentioned, in cases such as this one the `repOK` comes already modularised, since it simply corresponds to the conjunction of `repOK`'s for the different parameters. Table III shows, for different scopes, the sizes of the respective bounded exhaustive suites (BE), the number of explored candidates (Explored) and the time expended in generation (Time), both for traditional bounded exhaustive generation and for separate generation of disjoint substructures. Again, the number of valid inputs is reported once since it is the same in both cases. The scope in this case specifies the maximum number of elements for both heaps, and the range for the keys of the nodes, from zero to the specified value. Again, the disjoint generation technique is effective in this case study (the highlighted rows of the tables indicate the cases in which the presented technique outperforms, with respect to time, traditional bounded exhaustive generation), in number of visited structures, and consequently in generation time. For instance, for scope 6, the disjoint generation technique required visiting 42,815 structures, as opposed to the 274,808,123 structures visited by bounded exhaustive generation. Figure 10 provides a graphical representation of the information in Table III.

Scope	BE	Korat		Disj. Generation	
		Explored	Time	Explored	Time
2,2	36	348	0.26 s	58	0.427 s
3,3	784	5389	0.499 s	235	0.45 s
4,4	14,400	150,448	0.866 s	1666	0.559 s
5,5	876,096	3,125,314	7.187 s	8122	1.641 s
6,6	57,790,404	274,808,123	693.116 s	42,815	64.499 s
7,7		TO	TO	261,788	13470.976 s

Table III. Bounded exhaustive vs. separate bounded exhaustive generation of pairs of `BinomialHeap`.

5.1.3. AVL Trees. The last case study for separate generation of disjoint substructures corresponds to `AVLTree` and `SinglyLinkedList`. The motivation is the bounded exhaustive test input generation for a routine receiving these two data structures as parameters, such as an `addAll` routine, adding all the elements in a collection (in this case, a linked list) into an AVL tree. Tables IV and V show, for different scopes (where each scope specifies the maximum number of nodes in the tree, the range for the size of the tree, the maximum number of nodes in the list, the range for the size of the list, and the number of keys allowed in the tree and the list, in this order), the sizes of the corresponding bounded-exhaustive suites (BE), the number of explored candidates (Explored) and the time spent during generation (Time). This information is provided for bounded exhaustive generation, and for the separate generation of AVL trees and linked lists. Although a larger scope is necessary for the technique to outperform traditional bounded exhaustive generation, again the experimental results for this case study clearly show the benefits of the technique. Figures 12 and 11 provide a more graphical representation of the information in Tables IV and V, showing how, for this case study.

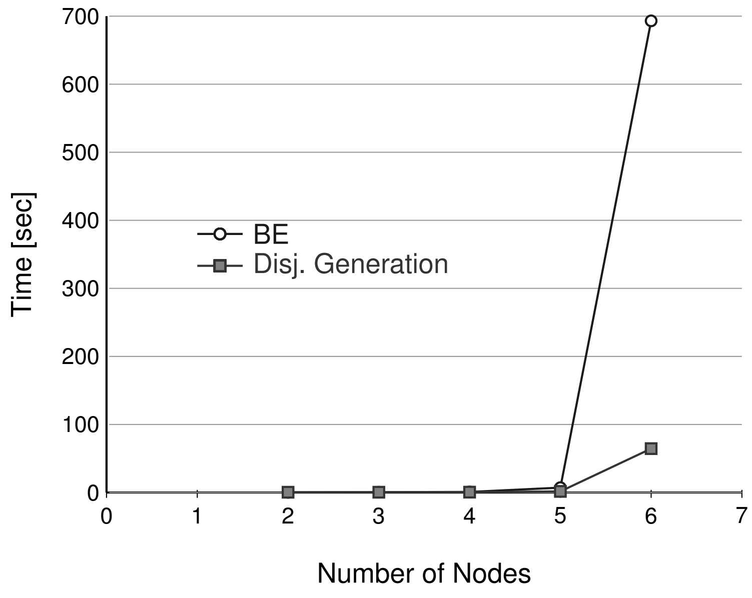

Figure 10. Bounded exhaustive vs. separate bounded exhaustive generation of pairs of `BinomialHeap` (graphical representation).

Scope	BE	Korat		Disj. Generation	
		Explored	Time	Explored	Time
2,0,2,3,0,2,6	1591	2925	0.303 s	442	0.546 s
3,0,3,4,0,3,6	14,763	22,878	0.533 s	3951	0.669 s
4,0,4,5,0,4,6	181,935	243,217	1.253 s	22,122	1.052 s
5,0,5,6,0,5,6	1,427,643	1,790,743	4.255 s	80,288	2.62 s
6,0,6,7,0,6,6	8,789,959	10,724,428	25.836 s	232,025	10.719 s
7,0,7,8,0,7,6	52,739,911	63,672,425	166.652 s	772,602	52.568 s
8,0,8,9,0,8,6	316,439,623	380,739,460	1076.359 s	2,511,653	339.56 s

Table IV. Bounded exhaustive vs. separate bounded exhaustive generation of pairs composed of an AVL tree and a linked list, as the number of nodes in these structures is increased.

Scope	BE	Korat		Disj. Generation	
		Explored	Time	Explored	Time
6,0,6,7,0,6,2	635	6884	0.339 s	5609	0.628 s
6,0,6,7,0,6,3	12,023	27,594	0.511 s	10,505	0.829 s
6,0,6,7,0,6,4	136,525	202,784	1.175 s	26,313	1.124 s
6,0,6,7,0,6,5	1,210,922	1,569,144	3.882 s	75,438	2.384 s
6,0,6,7,0,6,6	8,789,959	10,724,428	25.836 s	232,025	10.719 s
6,0,6,7,0,6,7	52,020,403	61,244,650	152.977 s	685,649	50.262 s
6,0,6,7,0,6,8	254,354,457	292,245,608	756.38 s	1,831,897	242.524 s
6,0,6,7,0,6,9	1,053,448,702	1,188,984,604	3154.268 s	4,389,842	953.695 s

Table V. Bounded exhaustive vs. separate bounded exhaustive generation of pairs composed of an AVL tree and a linked list, as the number of keys in these structures is increased.

5.2. Case Studies for `RepOK`-based Test Suite Reduction

First, a description of the structure of the experiments is presented. The `repOK` code for each of the structures analysed in this section was taken, and automatically instrumented to obtain, from a

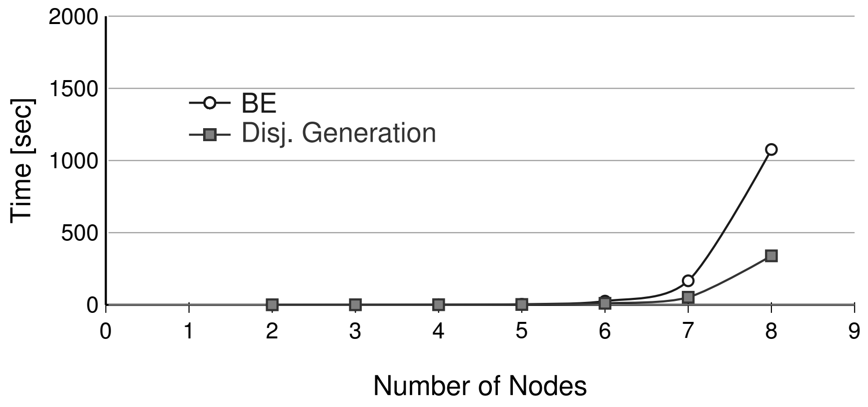

Figure 11. Bounded exhaustive vs. separate bounded exhaustive generation of pairs composed of an AVL tree and a linked list, as the number of nodes in these structures is increased (graphical representation).

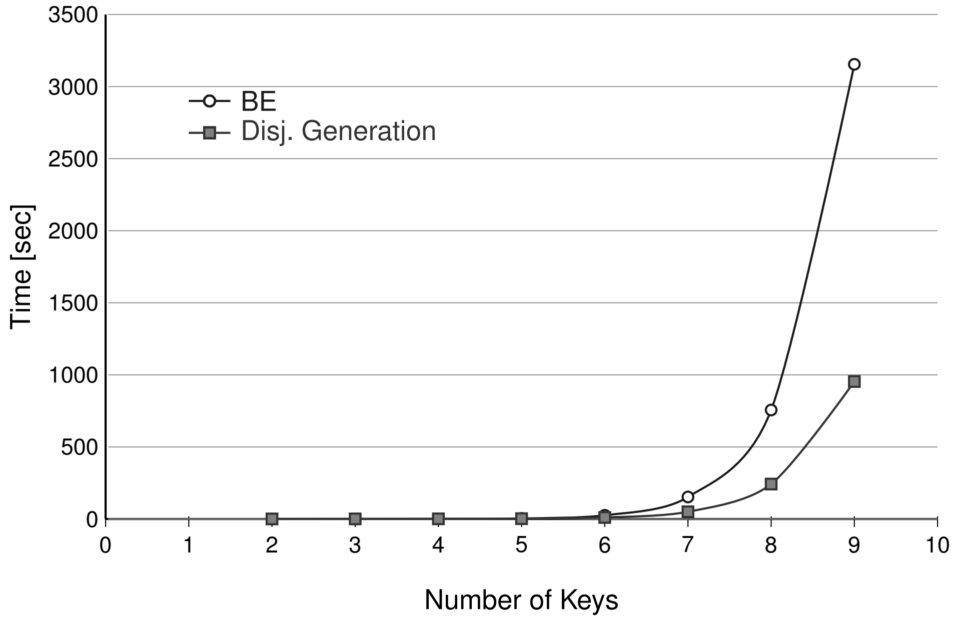

Figure 12. Bounded exhaustive vs. separate bounded exhaustive generation of pairs composed of an AVL tree and a linked list, as the number of keys in these structures is increased (graphical representation).

`repOK` call on a given valid structure, the equivalence class the structure belongs to, for each of the selected criteria. The instrumented `repOK` methods were ran on tests of the bounded-exhaustive test suite to collect their equivalence class information, then reduced test suites were built that select from a bounded-exhaustive test suite some test cases for each (coverable) equivalence class corresponding to the criterion. In particular, the bounded-exhaustive test suites were reduced 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, at least one test case for each covered equivalence class has been taken. Notice that, the number of covered classes (CC) is also reported, that is the number of different coverable equivalence classes for each of the selected criteria.

To measure the effectiveness of the approach, some sample routines manipulating the data structures selected for analysis were taken. 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. Mutants of these routines were generated and the effectiveness of the different suites, bounded-exhaustive and reduced, was measured with respect to their mutant killing ability. Also, the "one per class" (OPC) 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), were considered in the assessment. In order to generate mutants, muJava [20] was used. The achieved mutants are those obtained by the application of 12 different method-level mutation operators [17], including arithmetic, logical and relational operator replacement, when these ones were applicable to the selected routines.

It is worth to underline the relationship between covered classes (CC) and "one per class" (OPC) suites. CC is the number of covered equivalence classes for each of the selected criteria. The "one per class" suites are the ones built taking one test case for each of these covered classes. More precisely, the test cases selected to construct the "one per class" suites are the *first* generated test cases for each of the coverable equivalence classes.

Some potential threats to the validity of the experimental results were analysed. The case studies represent, in the opinion of the authors, typical testing situations in the context of the implementation of complex, heap allocated data structures (a main target for bounded-exhaustive testing). Case studies of varying complexities were chosen, 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`, implementations of these routines were taken as provided in Korat, instead of providing ad-hoc implementations. Also, for evaluation purposes, coverage criteria of varying complexities were selected: the rather simple decision coverage, the more thorough path coverage, and an intermediate one, counting decision coverage.

Counting decision coverage (CDC) is a coverage criterion introduced in this paper, and defined as follows. 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. The authors believe CDC to be useful in the context of bounded exhaustive testing 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 suites obtained using this criterion. For instance, while decision coverage considers as equivalent the first two trees in Figure 7, CDC will distinguish them.

5.2.1. Binomial Heaps (`merge`). The first case study to assess the approach to filtering bounded exhaustive suites involves testing the `merge` routine, which manipulates pairs of binomial heaps.

This is an example of a case in which the bounded-exhaustive suites quickly become too large, making bounded-exhaustive testing impractical. Table VI 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).

As mentioned before, the effectiveness of the suites is measured using mutation testing. The `merge` routine was mutated, obtaining a total of 113 mutants not equivalent to the original program. Then, the ability to kill mutants using the bounded-exhaustive suite, the reduced test suites and the minimal "one per equivalence class" (OPC), that is, the suite generated by taking exactly one input for each covered class (CC), was assessed. Table VII 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. Notice that 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.

5.2.2. Binomial Heaps (`insert`, `delete` and `find`). The second case study for the filtering technique involves routines manipulating a single binomial heap, namely *insert*, *delete* and *find*. Table VIII 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`; again for each criterion, the number of equivalence classes of inputs that have been covered (CC columns) is reported. The scope in this case simply indicates the size of the corresponding binomial heap.

Again, the effectiveness of the suites (bounded exhaustive, reduced to a 10% and 1%, and one per equivalence class (OPC), i.e, the suite generated by taking exactly one input for each covered class) is measured using mutation testing. Routines `insert`, `delete` and `find` were mutated (the number of mutants not equivalent to the original program obtained were 97, 170 and 28, respectively), and the effectiveness of the different suites on mutant killing was assessed. Table IX reports the results of the analysis for this case study, indicating the remaining live mutants in each case, and highlighting the cases in which the mutation score of the reduced suites matched that of the corresponding bounded exhaustive suite.

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 (18 vs. 17 out of 170) compared to the bounded-exhaustive suite. This missed mutant corresponds to an insertion of a "post-increment operator" in the code under test. The problem is that capturing this mutant depends on exactly which value in the heap is deleted and where the value is placed in the heap, i.e., in which node it resides. Since the coverage criteria on `repOK` only take into account the structure of the binomial heap, but not where the

Scope	BE	Decision Cov.		
		10%	1%	CC
2,2	36	3	3	3
3,3	784	76	4	4
4,4	14,400	1200	144	4
5,5	876,096	49,420	7506	4
6,6	57,790,404	2,455,826	342,166	4

Scope	BE	Count. Decision Cov.		
		10%	1%	CC
2,2	36	9	9	9
3,3	784	59	16	16
4,4	14,400	1060	119	25
5,5	876,096	42,500	6460	36
6,6	57,790,404	1,993,860	315,698	49

Scope	BE	Path Cov.		
		10%	1%	CC
2,2	36	9	9	9
3,3	784	59	16	16
4,4	14,400	1060	119	25
5,5	876,096	42,500	6460	36
6,6	57,790,404	1,993,860	315,698	49

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

Scope	BE	Decision Cov.			Count. Dec. Cov.			Path Cov.		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
2,2	34	96	96	96	38	38	38	38	38	38
3,3	4	9	82	82	4	10	10	4	10	10
4,4	3	3	9	82	3	3	8	3	3	8
5,5	3	3	3	82	3	3	8	3	3	8
6,6	3	3	3	82	3	3	8	3	3	8

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

value to be deleted resides, we have "too coarse" equivalence classes and the inputs that exercise the mutated code in this case were removed.

5.2.3. Doubly Linked Lists (`isPalindromic`). The 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. Table X 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`. The number of equivalence classes of inputs covered by each case is also indicated (CC columns). 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 21 mutants not equivalent to the original program. The effectiveness of the suites was assessed measuring how many of these mutants were

Sc.	BE	Decision Cov.			Count. Dec. 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

Table VIII. Sizes of bounded-exhaustive and suites with `repOK`-based reductions, for testing binomial heap's operations `insert`, `delete` and `find`.

Sc.	Oper.(#Muts)	BE	Decision Cov.			Count. Dec. Cov.			Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
2	insert(97)	34	42	42	42	42	42	42	42	42	42
	delete(170)	110	138	138	138	138	138	138	138	138	138
	find(28)	0	12	12	12	12	12	12	12	12	12
3	insert(97)	23	24	32	32	24	32	32	24	32	32
	delete(170)	67	92	135	135	92	135	135	92	135	135
	find(28)	0	8	12	12	8	12	12	8	12	12
4	insert(97)	23	23	32	32	23	32	32	23	32	32
	delete(170)	63	85	135	135	87	135	135	87	135	135
	find(28)	0	6	12	12	6	12	12	6	12	12
5	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	47	76	87	135	51	71	135	51	71	135
	find(28)	0	2	6	12	2	6	12	2	6	12
6	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	19	51	85	135	34	39	101	34	39	101
	find(28)	0	2	6	12	0	3	12	0	3	12
7	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	17	23	68	135	21	35	101	21	35	101
	find(28)	0	0	5	12	0	0	12	0	0	12
8	insert(97)	23	23	23	32	23	23	32	23	23	32
	delete(170)	17	40	56	135	18	34	101	18	34	101
	find(28)	0	2	5	12	0	0	12	0	0	12

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

killed by the different suites. Table XI reports the results of the analysis for this case study, indicating the mutants that remained live in each case, and highlighting those cases in which the reduced suites matched the mutation score of the bounded exhaustive suites.

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.

5.2.4. Search Trees (`insert`, `delete` and `search`). The next case study involves the data structure Search Tree, and its insertion, deletion and search routines. Table XII shows, for various scopes, the sizes of bounded exhaustive (BE) suites and suites with `repOK`-based reductions to

Scope	BE	Decision Cov.		
		10%	1%	CC
4,0,4,4	156	8	2	2
4,0,4,8	820	42	5	2
5,0,5,5	1555	78	8	2
5,0,5,10	16,105	806	81	2
6,0,6,6	19,608	981	99	2
6,0,6,12	402,234	20,112	2,012	2
7,0,7,7	299,593	14,980	1,498	2
7,0,7,14	12,204,241	610,213	61,022	2

Scope	BE	Count. Decision Cov.		
		10%	1%	CC
4,0,4,4	156	10	4	4
4,0,4,8	820	50	7	4
5,0,5,5	1555	100	13	5
5,0,5,10	16,105	777	108	5
6,0,6,6	19,608	1035	136	6
6,0,6,12	402,234	15,786	2,193	6
7,0,7,7	299,593	13,239	1,781	7
7,0,7,14	12,204,241	402,933	55,918	7

Scope	BE	Path Cov.		
		10%	1%	CC
4,0,4,4	156	10	4	4
4,0,4,8	820	50	7	4
5,0,5,5	1555	100	13	5
5,0,5,10	16,105	777	108	5
6,0,6,6	19,608	1035	136	6
6,0,6,12	402,234	15,786	2,193	6
7,0,7,7	299,593	13,239	1,781	7
7,0,7,14	12,204,241	402,933	55,918	7

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

Scope	BE	Decision Cov.			Count. Dec. Cov.			Path Cov.		
		10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
4,0,4,4	13	13	21	21	13	20	20	13	20	20
4,0,4,8	13	13	21	21	13	13	20	13	13	20
5,0,5,5	11	13	20	21	11	13	20	11	13	20
5,0,5,10	11	13	13	21	11	11	20	11	11	20
6,0,6,6	11	11	13	21	11	11	20	11	11	20
6,0,6,12	11	11	13	21	11	11	20	11	11	20
7,0,7,7	11	11	11	21	11	11	20	11	11	20
7,0,7,14	11	11	11	21	11	11	20	11	11	20

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

10% and 1%, for the three mentioned white box coverage criteria applied to `repOK`; again for each criterion, it is also indicated the number of equivalence classes of inputs that have been covered (CC columns). 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), and the effectiveness of the different suites on mutant killing was assessed. Table XIII reports the results obtained for the analysis, indicating the remaining live mutants in each case, and highlighting the cases in which the mutation score of the reduced suites matched that of the corresponding bounded exhaustive suite.

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). These two mutants correspond to a relational operator replacement and a conditional operator insertion. In this case these two mutants result to be equivalent. These mutants are missed by the reduced suites for a similar reason that mutants were missed on *binomial heaps*. Inputs are distinguished by structural conditions of the trees, number of non-null left and right children. When the `delete` operation is tested, the chances of exercising the mutated part of the code under test depend not only on the structure of the tree but also on the item to be deleted, and where it is located in the tree.

Scope	BE	Decision Cov.			Count. Dec. 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

Table XII. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `delete`, `insert` and `search` operations of Search Tree.

5.2.5. Red-Black Trees (`remove`, `add` and `contains`). The last case study presented involves routines manipulating *red-black trees*. These routines are `remove`, `add` and `contains`. Table XIV 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`; again, for each criterion it is also indicated the number of equivalence classes of inputs that have been covered (CC columns). The scopes 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 the analysis. Thus, in this case study, a *bounded* version of path coverage is considered, namely path coverage without taking into account repetitions of edges (known as *simple path coverage* [29]).

Routines `remove`, `add` and `contains` were mutated (the number of mutants obtained not equivalent to the original program were 138, 115 and 32, respectively), and the results of the analysis are reported in Table XV. In this case study, reduced test suites showed better effectiveness for the

Scope	Op.(#Muts)	BE	Decision Cov.			Count. Dec. 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

Table XIII. Measurement of effectiveness of bounded-exhaustive and reduced test suites, on mutant-killing for `insert`, `delete` and `search` of Search Tree (table reports mutants remaining live).

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

Table XIV. Sizes of bounded-exhaustive suites and suites with `repOK`-based reductions, for testing `remove`, `add` and `contains` operations of Red-Black Tree.

`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, 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 115 and for remove in scope 7,0,7,14 missed only 3 out of 138 compared to the bounded-exhaustive suite). About the missed mutants, this case does not differ too much from the already presented ones, the mutated parts are located deep in the code under test and the ability to capture those mutations, depends on exactly where the node to be deleted/added is placed in the tree, but this property is not taken into account by the coverage criteria applied to `repOK`.

Sc.	Op.(#Muts)	BE	Decision Cov.			Count. Dec. Cov.			Simple Path Cov.		
			10%	1%	OPC	10%	1%	OPC	10%	1%	OPC
4,0,4,4	remove(138)	43	78	78	77	66	66	66	43	43	43
	add(115)	27	47	79	79	79	79	79	33	33	33
	contains(32)	2	4	5	5	5	5	5	2	2	2
4,0,4,8	remove(138)	43	62	77	77	61	66	66	78	78	78
	add(115)	27	29	32	79	29	47	79	42	51	51
	contains(32)	2	2	3	5	2	4	5	4	4	4
5,0,5,5	remove(138)	39	77	78	77	64	64	64	64	64	64
	add(115)	27	32	79	79	79	79	79	45	45	45
	contains(32)	2	3	5	5	5	5	5	4	4	4
5,0,5,10	remove(138)	39	51	73	77	48	63	64	78	78	78
	add(115)	25	29	31	79	28	32	79	38	47	67
	contains(32)	2	2	2	5	2	3	5	4	4	4
6,0,6,6	remove(138)	37	62	78	77	42	62	62	67	67	67
	add(115)	25	31	47	79	44	79	79	51	51	51
	contains(32)	2	2	4	5	3	5	5	4	4	4
6,0,6,12	remove(138)	37	49	56	77	46	57	62	78	78	78
	add(115)	25	29	29	79	26	29	79	36	38	67
	contains(32)	2	2	2	5	2	2	5	4	4	4
7,0,7,7	remove(138)	37	56	77	77	37	62	62	72	72	72
	add(115)	25	29	32	79	29	79	79	42	51	51
	contains(32)	2	2	3	5	2	5	5	4	4	4
7,0,7,14	remove(138)	37	46	51	77	40	46	62	72	78	78
	add(115)	25	31	47	79	25	29	79	36	38	68
	contains(32)	2	2	2	5	2	2	5	4	4	4

Table XV. 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).

6. RELATED WORK

Various approaches deal with bounded exhaustive test generation, and how to tackle scalability issues. With respect to improving the bounded exhaustive testing time, Jagannath et al. [13] 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 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 the work presented in this paper, since it has as a purpose to reduce the size of the test suite. In fact, it is related to both techniques in this paper, since it reduces the generation time (by a structural manipulation) and also reduces the size of the obtained test suite. However, the approach is rather different to both presented techniques, since *(i)* smaller inputs are encoded as larger inputs, as opposed to the separate generation technique for disjoint substructures, that works in the opposite direction with respect to test granularity, and *(ii)* bounded exhaustiveness is preserved in structural test merging (although sets of small inputs are encoded as a single large input), whereas in the test suite reduction technique bounded exhaustiveness is dropped by selecting only some tests. The same differences apply to other works based on test granularity [23].

The technique for improving test generation by considering separately disjoint substructures is also related to other approaches improving the generation of bounded exhaustive suites, such as works on parallelisation of bounded exhaustive generation. Misailovic et al. [19] presented an approach for parallelising the bounded exhaustive test generation. Since, the separate generation for disjoint substructures can be independently performed, it is a straightforward approach for bounded exhaustive parallelisation. The disjoint generation technique is however significantly different from that presented by Misailovic et al. [19], since in the latter the candidate vectors encoding the possible instances are analysed in order to split the generation work, without taking into account semantic information about the structures being generated (as in the case of the disjoint generation technique, which requires information on the disjointness of substructures).

Besides the work presented by Bengolea et al. [4] (which this paper extends), 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 presented by Aguirre et al. [2] is strongly related to the work presented in this paper, especially because both approaches are based on the use of coverage criteria. However, the approach presented by Aguirre et al. [2] differs from the work in this paper in two aspects. First, it requires the user to provide the coverage criterion to perform the suite reduction, as opposed to the 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 the work in this paper concerns the reduction of bounded-exhaustive test suites to reduce the time for testing.

Other researchers have studied the effects of reducing test suites in finding bugs, e.g. Yu et al. [28]. The work in this paper is related, but a specific approach for test-suite reduction is proposed here (as opposed to studying the effects of test-suite reductions in general), and bounded-exhaustive test suites are specifically targeted.

For the case of bounded exhaustive test case generation of structurally complex, heap allocated inputs, various tools have been proposed. Among these, Java PathFinder [27], Alloy [12], CUTE [25] and Korat [5], which is used for the experiments in this paper (a thorough comparison between these tools for bounded exhaustive test generation is reported by Siddiqui et al. [24]), may be cited. An alternative tool for performing the experiments could have been used, although as is discussed by Siddiqui et al. [24], Korat is generally the most efficient, which justifies this selection. Although the experiments in this paper involved Korat, the technique applies to other tools that perform bounded exhaustive generation by filtering. Examples of such tools are Alloy [12], UDITA [11] (which also supports a generative approach) and AsmL [3].

7. CONCLUSIONS AND FURTHER WORK

Bounded exhaustive test generation is an effective testing technique in various contexts, such as that of testing complex heap allocated data structures. In many cases, however, both the time for test generation and for test execution grow exponentially with respect to the scope employed for bounded exhaustive generation. This fact, combined with the developer's need to use "larger" scopes in order to cover interesting cases or achieve a certain level of coverage (or, in general, improve the chances of finding bugs), makes in many cases either the generation or the use of the generated suites impractical. In this paper, two approaches have been presented to reduce the time employed in bounded exhaustive testing, one that aims at reducing the time for test generation by enabling the independent generation of disjoint substructures of the inputs, and the other aiming at reducing the testing time, by reducing the size of bounded exhaustive suites by filtering redundant cases (for a particular notion of redundancy, also presented in this work).

These techniques make use of the representation invariant of the code under test, usually implemented as a `repOK` routine. This `repOK` routine is used in these techniques in two different ways: in the case of *independent generation of substructures*, the representation invariant is used to factor out separate representation invariants for disjoint substructures of the inputs. In the case of *test suite reduction*, the representation invariant is used to define 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 `repOK` routine. As it is studied in this paper, these `repOK` based techniques may have a significant impact in bounded exhaustive testing. For the first technique, this is so because when inputs are composed of disjoint substructures, these substructures can be independently generated. This has various advantages, such as the parallelisation of the generation. Even if the generation of the substructures is performed sequentially, the problem of reconsidering every valid structure of the second substructure for each valid one of the first substructure, is avoided. The presented technique for test filtering is motivated by the observation that, if two inputs exercise the representation invariant code "in the same way", they might be considered "equivalent", and this fact exploited to define a black box coverage criterion for the code under test. As the experiments show, the resulting coverage criteria are meaningful in the context of bounded exhaustive testing, and suites are significantly reduced by filtering while maintaining the mutant killing ability of exhaustive suites. Especially for *counting decision coverage* in the biggest scopes, few mutants are missed only in some of the cases. More experimental evaluation could be done comparing these new black box criteria with testing of equivalence classes over the code under test, using statement coverage and even more sophisticated white box criteria. However keeping the equivalence classes independent of the code under test, as in the case of repOK-based reduction, has a valuable advantage: the obtained suite could be use to test any method of the structure on which it is generated as opposite of white box testing, where the equivalence classes are generated to test a particular method.

The introduced techniques rely on programs being equipped with their corresponding contract specifications. While writing contracts for code is not a wide-spread practice, recent efforts, most notably the work on JML, Code Contracts and similar frameworks, favour contract specification by integrating these as part of development environments, e.g., through libraries with special sentences to capture preconditions, postconditions, invariants, etc. Although the techniques presented in this paper require an imperative representation invariant, these are still relevant in contexts where declarative representation invariants, of the kind associated with JML and Code Contracts, are employed. Declarative languages for contracts are typically accompanied by run-time contract-checking environments, making these declarative contracts "executable"; the code corresponding to their run-time evaluation would correspond to what is is referred here as `repOK`. When reducing the bounded-exhaustive test suite's size, 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.

There are various lines for further work related to the techniques presented in this paper. The effectiveness of parallelising the generation of disjoint substructures has not been yet assessed. It is of course expected that this will improve the generation time compared to what is presented in this paper. Such an approach to the parallelisation of bounded exhaustive generation must also be compared with other works parallelising bounded exhaustive test generation, such as that presented by Misailovic et al. [19]. Some important technical issues are interesting sources for further lines of work. One has to do with analysing different sources of information in order to determine the disjointness of substructures of a given structure. The analysis of such sources, such as finitisation procedures, other user provided information and even the processing of the representation invariant, is an important problem to tackle, in particular if it is desirable to fully automate it. Another issue is the effective processing of the `repOK` in order to obtain "local" versions of it for the different disjoint substructures. This work is currently being done by a simple slicing mechanism based on *definition-use graphs*, but more sophisticated approaches may yield better "local" `repOK`'s.

ACKNOWLEDGEMENTS

This material is based upon work partially supported by Argentina's ANPCyT through grants PICT 2010-1690 and PICT 2012-1298, by the US National Science Foundation under Grant No. CCF-1012759, and by the MEALS project (EU FP7 MEALS - 295261).

REFERENCES

1. P. Abad, N. Aguirre, V. S. Bengolea, D. Ciolek, M. F. Frias, J. P. Galeotti, T. Maibaum, M. Moscato, N. Rosner and I. Vissani, *Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving*, in Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation ICST 2013, IEEE, 2013.
2. N. Aguirre, V. Bengolea, M. Frias and J. Galeotti, *Incorporating Coverage Criteria in Bounded Exhaustive Black Box Test Generation of Structural Inputs*, in Proc. of Intl. Conference on Tests and Proofs TAP 2011, LNCS 6706, Springer, 2011.
3. M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann and M. Veanes, *Model-Based Testing with AsmL .NET*, in Proceedings of the 1st European Conference on Model-Driven Software Engineering, 2003.
4. V. Bengolea, N. Aguirre, D. Marinov, M. Frias: *Using Coverage Criteria on RepOK to Reduce Bounded-Exhaustive Test Suites*, in Proc. of Intl. Conference on Tests and Proofs TAP 2012, LNCS 7305, Springer, 2012.
5. C. Boyapati, S. Khurshid and D. Marinov, *Korat: Automated Testing based on Java Predicates*, in Proc. of Intl. Symposium on Software Testing and Analysis ISSTA 2002, ACM Press, 2002.
6. P. Chalin, J.R. Kiniry, G.T. Leavens and E. Poll, *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*, in Proc. of Intl. Symposium on Formal Methods for Components and Objects FMCO 2005, LNCS 4111, Springer, 2005.
7. K. Claessen and J. Hughes, *QuickCheck: a lightweight tool for random testing of Haskell programs*, in Proceedings of the fifth ACM SIGPLAN international conference on Functional programming ICFP 2000, ACM, 2000.
8. D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan, *Software assurance by bounded exhaustive testing*, IEEE Transactions on Software Engineering 31 (4): 328-339, 2005.
9. G. J. Myers, *The Art of Software Testing*, 2nd. Ed., John Wiley & Sons, Inc, 2004.
10. C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering*, Second Edition, Prentice-Hall, 2002.
11. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak and D. Marinov, *Test generation through programming in UDITA*, in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE 2010, Cape Town, South Africa, ACM Press, 2010.

12. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
13. V. Jagannath, Y.Lee, B.Daniel and D. Marinov, *Reducing the Costs of Bounded-Exhaustive Testing*, in Proc. of Intl. Conference on Fundamental Approaches to Software Engineering FASE 2009, LNCS 5503, Springer, 2009.
14. P. Jalote, *An Integrated Approach to Software Engineering*, 3rd. Ed., Springer, 2005.
15. S. Khurshid and D. Marinov, *TestEra: Specification-Based Testing of Java Programs Using SAT*, Automated Software Engineering 11(4), Springer, 2004.
16. B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification and Object-Oriented Design*, Addison-Wesley, 2000.
17. Y.-S. Ma, J. Offutt and Y.-R. Kwon, *MuJava : An Automated Class Mutation System*, Journal of Soft ware Testing, Verification and Reliability, 15(2), Wiley, 2005.
18. A. Milicevic, S. Misailovic, D. Marinov and S. Khurshid, *Korat: A Tool for Generating Structurally Complex Test Inputs*, in Proc. of Intl. Conference on Software Engineering ICSE 2007, IEEE Press, 2007.
19. S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid and D. Marinov, *Parallel test generation and execution with Korat*, in Proc. of 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering, ESEC/SIGSOFT FSE 2007, ACM Press, 2007.
20. MuJava, `http://www.cs.gmu.edu/~offutt/mujava/`.
21. C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, *Feedback-Directed Random Test Generation*, in Proceedings of the 29th international conference on Software Engineering ICSE 2007, IEEE, 2007.
22. *Roops*, `http://code.google.com/p/roops/`.
23. G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri and B. Davia, *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.
24. J. Siddiqui and S. Khurshid, *An Empirical Study of Structural Constraint Solving Techniques*, in Proceedings of the 11th International Conference on Formal Engineering Methods ICFEM 2009, LNCS, Springer, 2009.
25. K. Sen, D. Marinov and G. Agha, *CUTE: a concolic unit testing engine for C*, in Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering, ESEC/SIGSOFT FSE 2005, ACM Press, 2005.
26. N. Tillmann and J. de Halleux, *Pex: White Box Test Generation for .NET*, in Proceedings of the Second International Conference on Tests and Proofs TAP 2008, LNCS, Springer, 2008.
27. W. Visser, C. Pasareanu, S. Khurshid, *Test input generation with java PathFinder*, in Proc. of Intl. Symposium on Software Testing and Analysis ISSTA 2004, ACM Press, 2004.
28. Y. Yu, J. Jones and M. Harrold, *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.
29. H. Zhu, P. Hall and J. May, *Software Unit Test Coverage and Adequacy*, ACM Computing Surveys 29(4), ACM Press, 1997.