Incorporating Coverage Criteria in Bounded Exhaustive Black Box Test Generation of Structural Inputs

Nazareno M. Aguirre[1], Valeria S. Bengolea[1], Marcelo F. Frias[2], and Juan P. Galeotti[3]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Río Cuarto, Córdoba, Argentina
{naguirre,vbengolea}@dc.exa.unrc.edu.ar
[2] Departamento de Ingeniería Informática, Instituto Tecnológico Buenos Aires and CONICET, Buenos Aires, Argentina
mfrias@itba.edu.ar
[3] Departamento de Computación, FCEyN, Universidad de Buenos Aires and CONICET, Buenos Aires, Argentina
jgaleotti@dc.uba.ar

Abstract. The automated generation of test cases for heap allocated, complex, structures is particularly difficult. Various state of the art tools tackle this problem by *bounded exhaustive* exploration of potential test cases, using constraint solving mechanisms based on techniques such as search, model checking, symbolic execution and combinations of these.

In this article we present a technique for improving the bounded exhaustive constraint based test case generation of structurally complex inputs, for "filtering" approaches. The technique works by guiding the search considering a given black box test criterion. Such a test criterion is incorporated in the constraint based mechanism so that the exploration of potential test cases can be pruned without missing coverable classes of inputs, corresponding to the test criterion.

We present the technique, together with some case studies illustrating its performance for some black box testing criteria. The experimental results associated with these case studies are shown in the context of Korat, a state of the art tool for constraint based test case generation, but the approach is applicable in other contexts using a filtering approach to test generation.

1 Introduction

Testing is a powerful and widely used technique for software quality assurance [6]. The technique essentially consists of executing a piece of code, whose quality needs to be assessed, under a number of particular inputs, or *test cases*. For these test cases to be adequate, they generally need to try the software under different circumstances. A variety of *test criteria* have been devised, which basically define what are the different situations that a set of test cases must exercise, or *cover* [17].

M. Gogolla and B. Wolff (Eds.): TAP 2011, LNCS 6706, pp. 15–32, 2011.

Generating test cases is generally a complex activity, in which the engineer in charge of the generation has to come up with inputs that satisfy, in many cases, complex constraints. The problem is particularly difficult when the inputs to be generated involve complex, heap allocated, structures, such as balanced trees, graphs, etc. Some tools [2,16,7,10,4] tackle this problem rather successfully, by *bounded exhaustive* exploration of potential test cases. More precisely, these tools work by generating all the inputs, within certain bounds (maximum number of objects of each of the classes involved in the structure), that satisfy a given constraint using some kind of constraint solver. Among the possible constraint solving techniques, model checking and other search related mechanisms have been implemented into state of the art tools.

In order to make the bounded exhaustive generation feasible, different mechanisms are implemented so that, in some cases, redundant structures are avoided, and such that parts of the state space corresponding to invalid structures are not explored. For instance, Korat implements a symmetry breaking mechanism together with an approach for avoiding the generation of invalid structures based on a sophisticated pruning technique; TestEra uses Alloy and its underlying symmetry breaking and optimisation mechanisms to improve the generation; UDITA implements a novel lazy evaluation mechanism, which in combination with symbolic execution greatly improve the test generation process.

In this work, we consider a complement to the so called *filtering approach* [4] to bounded exhaustive test generation, i.e., the process of exhaustively generating all possible structures (within the established bounds) and "filtering" to keep the valid or well formed ones. This complement takes into account a black box test criterion as part of the "generate and filter" process. Basically, in the same way that symmetric structures are avoided, we propose to also avoid the exploration of portions of the search space for test input candidates when such portions are guaranteed to provide test inputs corresponding to classes already covered by other test inputs previously generated. The result is somehow in between bounded exhaustive and "optimal" equivalence class coverage, and the actual "exhaustiveness" of the technique depends on the interaction of the test criterion (e.g., the adequacy of the predicates used for equivalence class coverage) and the generation procedure.

The motivation for this work lies in the fact that, by bounded exhaustive generation of test cases, in many cases, it becomes costly or infeasible to test a piece of code for all valid bounded inputs, even for small bounds, due to the large number of inputs obtained. Thus, a test criterion might be employed in order to "prune" the test generation, achieving a bounded exhaustive coverage of equivalence classes associated with the criterion. For instance, suppose that one counts with a black box test criterion for a given program to test. If one is interested in equivalence class coverage, it would be enough to generate a single test input per each (feasible) equivalence class. On the other hand, by bounded exhaustive generation one would be building all valid structures within the provided bounds. Instead, we propose to do a kind of exhaustive generation, but exploiting the possibility of pruning parts of the search space when one

is certain that all candidates in the pruned part correspond to classes already covered.

More precisely, our proposed technique works based on the following observation. The test generation mechanisms that follow a bounded exhaustive, filtering approach, generally contain a process for avoiding the generation of redundant structures. Independently of how such processes are implemented, they all correspond essentially to a pruning operation. For instance, the symmetry breaking formulas that TestEra incorporates in the Alloy model resulting from a program to be tested, instruct the underlying SAT solver to skip parts of the search space (in this context, assignments to propositional variables), thus constituting a pruning [7]. Similarly, UDITA prunes the search space to avoid isomorphic structures by incorporating isomorphism avoidance into the *object pool* abstraction and the operations for obtaining new objects from it in the construction of heap allocated structures [4]; Korat performs a similar pruning, imposing an ordering in the objects of the same type in the process of building the heap allocated structures [2]. Our approach proposes to take advantage of such pruning processes but in a different way; instead of just eliminating isomorphic (redundant) structures, we propose to take extra advantage of the pruning, and use it for skipping portions of the search space that would produce test cases covering classes that have already been covered by previous tests.

We present the approach by implementing it as a variant of the Korat algorithm/tool [10]. This variant is based on the use of a routine that we call `eqClass()`, that given a (valid) test input indicates which is the equivalence class it corresponds to, according to a test criterion. Basically, our variant of Korat, which we will refer to as Korat+[1], works as follows: when a candidate is found to be a valid test case, we invoke the `eqClass()` routine for this candidate, and look at what fields are observed to determine its equivalence class. We then try to "skip" the structures that coincide, for the observed fields, with the current candidate. Clearly, for any other candidate with the same values in these observed fields, its equivalence class would be the same as that of the current candidate.

We describe our approach in detail, via the mentioned variant of Korat. We provide examples and case studies together with their associated experimental results, using some black box test criteria. As it will be shown later on, our variant results in significant improvements, compared to Korat's search, for some of our case studies. As it is further explained later on, in some cases we were able to reduce the search space substantially, as well as to produce significantly less test cases, compared to bounded exhaustive generation. As we mentioned before, the technique results to be between bounded exhaustive, and "optimal" equivalence class coverage.

[1] Korat+ is only a variant of Korat that we present in this paper, for illustrating our technique. The name Korat+ is used only for reference purposes in the presentation of our approach. We fully acknowledge that the Korat tool and algorithm, as well as the name Korat, are the intellectual property of the authors of [2], and the technique we present, which could be applied in other contexts, should not be associated with the name Korat+.

2 Preliminaries

In this section we describe the Korat algorithm, which we use to present our technique. We also introduce a motivating example to drive the presentation.

2.1 The Korat Algorithm

Korat is an algorithm, and a tool implementing it, that allows for the generation of test cases composed of complex, heap allocated, structures [2]. Suppose, for instance, that we need to test a procedure that takes as a parameter a sorted singly linked list. Let us consider the following definition of the structure of sorted singly linked lists (a variant of `SinglyLinkedList`, as provided in Korat's distribution):

```
class SortedSinglyLinkedList {
    Node header;
    int size;
}
```

```
class Node {
    Integer elem;
    Node next;
}
```

A list is composed of a reference to a header node, and an integer value indicating the length of the list. The linked list of nodes starts with a header node with no element (a traditional dummy node); the actual contents of the list starts from the second node. The list should be acyclic, sorted (disregarding the header node), and the number of nodes in it minus one (the header) should coincide with the `size` value of the list. Korat can be used to generate such lists automatically, so that the procedure under analysis can be tested. Korat requires two routines accompanying the class associated with the input (in our example, `SortedSinglyLinkedList`). One of them is a boolean parameterless routine, called `repOk()` [8], that checks whether the structure satisfies its representation invariant. In our case, `repOk()` should check that the header is not null, and that no element is stored in it, that the list is acyclic and sorted, and the number of nodes in it minus one coincides with the value of the `size` field, as we explained before. The other routine that Korat requires is a *finitisation* procedure, that provides the bounds for the domains involved in the structure. This routine indicates the range for primitive type fields (e.g., that `size` in `SortedSinglyLinkedList` goes from 0 to 3), and the minimum/maximum number of objects of the classes involved in the structure (e.g., 1 list, 0 to 4 nodes, 1 to 3 integer objects).

Korat generates all possible valid structures within the provided bounds. By *valid* we mean that they satisfy the `repOk()` routine. For our example, this means that Korat will generate all acyclic sorted singly linked lists with dummy header, where the size coincides with the number of nodes in the linked list minus one, of size at most 3, containing integers from 1 to 3. In order to do so, Korat builds a tuple, where each entry corresponds to a value of a field of the involved objects. In our example, the tuple would have length 10, two values for the header and size of the lone list object, and the other 8 for the corresponding two fields of the four nodes that the list might at most contain. For instance, the following tuple

$$\langle 0, 0, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL} \rangle$$

would represent the empty list (where the first zero in the tuple is the reference to the first node object). Each entry in this tuple has a domain, which is defined by the finitisation procedure. Korat's actual algorithm works on what are called *candidate vectors*, vectors that represent the candidate tuples, but where the actual entries are replaced by indices into the respective domains. For instance, the candidate vector $\langle 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$ would correspond to the previously shown candidate tuple (each tuple entry has the first possible value in its domain, i.e., the value with index 0, except for the first entry, the head of the list, which points to the first node). Typically, most of the candidate vectors correspond to invalid structures, i.e., structures that do not satisfy the `repOk()`. Indeed, the space of candidates is in our example 3,200,000 ($5^5 \times 4^5$), but there exist only 8 singly sorted linked lists (up to isomorphism, as it will be explained later on), within the provided bounds.

Korat exhaustively explores the space of candidate vectors, using backtracking with a sophisticated pruning mechanism. More precisely, Korat works as follows: it starts with the initial candidate vector, with all indices in zero. It then executes `repOk()` on this candidate, monitoring the fields accessed in the execution, and storing these in a stack. Korat will then use this stack in order to backtrack over candidate vectors, as follows. If the current candidate satisfies `repOk()`, it is considered a valid test case (in this case, all reachable fields must be in the stack of accessed fields). If `repOk()` fails, then the candidate is discarded. In order to build the next candidate, Korat increments the last accessed field to its next value. If one or more of the last accessed fields are already in their corresponding maximum values, then these are resetted to 0, and the field accessed before them is incremented. If all fields are already at their maximum values, then the state space of candidate vectors has been exhaustively explored, and Korat terminates.

Notice that when `repOk()` fails, not all reachable fields might have been accessed, since its failure might be determined before exploring all reachable fields (for instance, in our example, if the first two nodes of the list are unordered, then `repOk()` fails without the need to explore the remaining part of the structure). Backtracking only on accessed fields is what enables Korat to prune large parts of the space of candidate vectors. It is sound since if the last accessed field is not modified, the output for `repOk()` would not change due to its determinism, (i.e., the parts of the structure visited by `repOk()` would remain the same, and therefore `repOk()` would fail again).

Besides the described search mechanism, with its incorporated search pruning, Korat also avoids generating isomorphic candidates [2]. Basically, two candidates are isomorphic if they only differ in the object identities of their constituents (i.e., if one of the candidates can be obtained from the other by changing the object identities). Most applications do not depend on the actual identities of objects (which represent the memory addresses or heap references of objects), and thus if one generated a structure, it is desirable to avoid generating its isomorphic structures, whose treatment would be redundant. Korat avoids generating isomorphic candidates by defining a lexicographic order between candidate vectors, and gen-

erating only the smallest in the order, among all isomorphic candidates. Basically, when considering the range of a class-typed field (i.e., its possible values) in the construction of candidates during the search, it is restricted to up to one "untouched" (i.e., not previously referenced in the structure) object of its corresponding domain. For example, suppose that in the construction of candidates one needs to consider different values for a given position i in the candidate vector. Suppose further that the ith position corresponds to a class domain D, and no fields of that domain have been accessed before i in the last invokation of `repOk()`. Then the only possible value for the i-th position is 0. More generally, if k objects of domain D have been accessed before in the last invokation of `repOk()`, these must be indexed 0 to $k-1$, and thus the ith position can go from 0 to k, but not beyond k. Korat's pruning and isomorphism elimination mechanisms allow the tool to reduce the search space significantly, in many cases. For our example, for instance, Korat explores only 319 out of the 3200000 possible cases, for linked lists with length 0 to 3, up to 4 nodes, and values ranging in integer objects from 1 to 3. For more details, we refer the reader to [2,10].

3 Incorporating Black Box Coverage to Bounded Exhaustive Search

In this section, we describe our proposal for improving a filtering approach to bounded exhaustive generation, by incorporating pruning associated with test criteria. Essentially, the approach is based on the observation that in many cases, the number of valid test cases, bounded by a value k, can be too large even for small bounds, and therefore evaluating the software under all these cases might be impractical. Then, our intention is to skip the generation of some test cases; the idea is to avoid generating test cases whose corresponding equivalence classes, for the test criterion under consideration, have already been covered. The idea is *not* to do "optimal" equivalence class coverage (one per equivalence class), but to approximate somehow to bounded exhaustive generation. That is, we would like to do a kind of bounded exhaustive generation, but with some pruning based on what the test criterion provides as information.

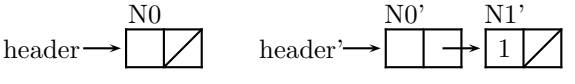
We present the approach by implementing it as a variant of the Korat algorithm, introduced in the previous section. In the same way that Korat requires an imperative predicate `repOk()`, we require a routine that we call `eqClass()`. This routine returns, given a valid candidate (i.e., a candidate satisfying `repOk()`), the equivalence class the candidate corresponds to, according to a test criterion. As for `repOk()`, this routine must be deterministic for exactly the same reason that `repOk()` must be deterministic. As opposed to Korat, which prunes (advances various candidates at once) only when `repOk()` fails, since if it does not fail all reachable fields must be in the stack of accessed fields, we prune the search space both when `repOk()` succeedes and when it fails: when `repOk()` fails, we advance various candidates at once using the fact that if none of the accessed fields is changed, then `repOk()` would fail again. When `repOk()`

succeedes, we execute `eqClass()` and monitor the accessed fields; we then advance various candidates at once to force a change in the last accessed field, since if none of the accessed fields changes the equivalence class would be the same of the previous valid candidate, which is already covered.

In order to better understand how this mechanism works, let us briefly expand our example. Suppose that we need a procedure `listAsSet` that, given a list `l` and a set `s`, both implemented over linked lists, determines whether `s` is the result of converting `l` to a set, i.e., disregarding repetitions and the order of elements in the list. From an implementation point of view, and taking into account the representation invariant of sets over singly linked lists, `s` should be the result of removing repetitions and sorting the list `l`. It is not difficult to find contexts in which a procedure of the kind of `listAsSet` is relevant. An obvious application of such a function would be an oracle for checking whether a list-to-set routine works as expected. If we want to generate test cases for `listAsSet`, we need to provide two objects, namely an arbitrary (acyclic) singly linked list of integers (the list `l`), and a strictly sorted acyclic singly linked list (the "set" `s`); the `repOk()` routine for this pair of objects checks first whether `l` is acyclic, and if so, it then checks whether `s` is acyclic and strictly sorted. Korat's candidate vectors will be composed of values for the fields of all objects of the two lists. Moreover, suppose that our test criterion takes into consideration all the combinations of four predicates:

- the first list is empty
- the first list has repeated elements
- the first list is sorted
- the second list is empty

The criterion is satisfied if at least one test case is produced for each of the satisfiable combinations of the truth values for the above predicates. Now, suppose that, in the search for valid candidates, Korat constructs the following pair of lists (for the linked list and the "set", respectively):



Let us refer to this pair of lists as `(l,s)`. Clearly, `(l,s)` satisfies `repOk()`. Let us analyse how Korat would proceed. Since `repOk()` is satisfied, Korat will move to the next candidate, which corresponds to advancing the last accessed field, i.e., `N1'.next`, assuming that `repOk()` checks the representation invariants of `l` and `s` in this order. Furthermore, because of the way `repOk()` works, Korat will produce all valid sets "greater than" (in the sense of the order in which Korat produces them) `s`, in combination with the empty `l`, before advancing `l`, i.e., producing a nonempty list.

Now let us analyse how our approach would proceed. According to the test criterion described before, this pair of lists corresponds to the equivalence class

$\langle T, F, T, F \rangle$ (i.e., the first list is empty, with no repeated elements and sorted, whereas the second one is nonempty). In order to determine the equivalence class for this test case, the parts of the structure that are examined are `header` (`N0`), `N0.next`, `header'` (`N0'`), `N0'.next`, in this order. Thus, if none of these fields are modified, the candidates produced would correspond to the same equivalence class as our current candidate `(1,s)`, which is already covered by this valid candidate. So, the approach "prunes" the search by attempting to advance the last accessed field, namely `N0'.next`, which is already at its maximum possible value (due to the rule of "at most one untouched object"). We move then to trying to advance `header'`, which again is at its maximum for the same reason as before, and thus we start considering greater values for `N0.next`. Notice how we avoided generating many (nonempty) sets, which in combination with the empty list would cover an already covered equivalence class. For example, if the finitisation procedure establishes that both lists have 0 to 4 nodes, and integers go from 1 to 3, then the described pruning, associated with our approach, constructs 679 candidates (320 of which are valid), skipping the construction of 14000 candidates (240 of which are valid, but cover already covered classes) that Korat would generate.

3.1 Soundness of the Approach

Let us argue about the soundness of the approach with respect to equivalence class coverage, i.e., that any valid test case in the pruned state space corresponds to a previously covered equivalence class. For comparison purposes, let us introduce a pseudo-code description of the standard Korat algorithm:

```
function korat() {
  Vector curr = initVector;
  Stack fields = new Stack();
  boolean ok;
  do {
    (ok, fields) = curr.repOk();
    if (ok) {
      reportValid(curr);
      fields.push(curr.reachFields - fields);
    }
    field = fields.pop();
    while (!fields.isEmpty() &&
           curr[field] >= nonIsoMax(curr, fields, field)) {
      curr[field] = 0;
      field = fields.pop();
    }
    if (!fields.isEmpty()) curr[field]++;
  } while (curr != lastVector && !fields.isEmpty())
}
```

In this pseudo-code description of the algorithm, we make an abuse of notation and make `repOk()`, which applies to candidate vectors, return both the

result of executing this function on the corresponding vector (a boolean, indicating whether the candidate is a valid one or not) and a stack with the fields accessed in the execution (`fields`). Notice how the backtracking is performed on the fields accessed by `repOk()`; also, when the current vector is valid, then all reachable fields are forced into the accessed fields, so that these are considered for backtracking and no candidates are missed. Finally, notice that an auxiliary function called `nonIsoMax`, which returns the maximum index possible for a given field, is used in order to determine the range of values for each field. This is crucial for the generation of nonisomorphic instances [2].

Our technique, which in this context we present as a variant of Korat referred to as Korat+, performs an extra pruning. It works by "popping out" more items from `fields`, the stack of accessed fields. In order to perform this pruning, the algorithm needs to compute the equivalence class for each valid candidate, monitoring the fields accessed in this computation (stored in `eqFields`). It then checks whether Korat's standard "next candidate" computation already advanced some of the fields accessed by the `eqClass()` routine, and if not it forces such an advance. The pseudo-code for our variant is the following:

```
function koratPlus() {
  Vector curr = initVector;
  Stack fields = new Stack();
  boolean ok;
  do {
    (ok, fields) = curr.repOk();
      if (ok) {
        reportValid(curr);
        fields.push(curr.Fields - fields);
        (eqClass, eqFields) = curr.eqClass();
        reportEqClass(eqClass);
      }
      List modified = new List();
      field = fields.pop();
      while (!fields.isEmpty() &&
             curr[field] >= nonIsoMax(curr, fields, field)) {
        curr[field] = 0;
        modified.add(field);
        field = fields.pop();
      }
      if (!fields.isEmpty()) {
        curr[field]++;
        modified.add(field);
      }
      // extra pruning
      if (ok &&
          (eqFields - modified == eqFields)) {
        for each field in modified {
          curr[field] = 0
        }
        boolean found = false;
```

```
        while (!fields.isEmpty() && !found) {
          field = fields.pop();
          if (eqFields.contains(field)) {
            found = true;
          }
          else {
            curr[field] = 0;
          }
        }
        if (found) {
          while (!fields.isEmpty() &&
                 curr[field] >= nonIsoMax(curr, fields, field)) {
            curr[field] = 0;
            field = fields.pop();
          }
          if (!fields.isEmpty()) {
            curr[field]++;
          }
        }
      }
  } while (!fields.isEmpty())
}
```

Guaranteeing the soundness of this pruning approach is relatively straightforward. First, notice that the backtracking order of the original Korat algorithm is preserved: Korat+ backtracks over `fields`, the fields accessed by `repOk()`. Our variant can only "pop" more items, but not modify the accessed fields (and thus the order of backtracking) in any other way.

Let us see that this new pruning can only skip valid candidates of already covered classes. Suppose that this new pruning stage is activated. Then, the previous candidate, which we will refer to as v_p, is a valid candidate, since `ok` is true; moreover, the standard Korat computation of the next candidate did not modify any of the fields accessed by `eqClass()`. This last pruning stage modifies the last field, according to `fields`, appearing in `eqFields`. Let v be a candidate vector pruned by this process. Assume further that v is a *valid* candidate. Since this candidate was pruned in this extra pruning, it corresponds to the pruned search space, which coincides in its values of the `eqFields` with v_p. Then, v corresponds to the same test equivalence class as v_p, due to the determinism of `eqClass()`. Therefore, the candidates pruned in the extra pruning stage correspond to the same equivalence class of v_p, which has already been covered by this test case.

4 Case Studies

We now describe some of the case studies we selected for assessing the technique. At the end of this section we will briefly analyse the experimental results associated with these case studies.

listAsSet. Our first case study corresponds to the `listAsSet` routine, and the black box test criterion described before, which requires covering all combinations of the predicates "first list is empty", "first list has repeated elements", "first list is sorted", and "second list is empty". The `repOk()` and `eqClass()` routines have been implemented as described in Section 3. This is a simple case study, with few equivalence classes, but it is still an interesting "toy" case study which serves the purpose of showing the benefits of the technique.

The experimental results are shown in the table below. The scope indicates the size ranges for the two lists (as separated scopes), the maximum number of nodes in each list (as separated scopes), and the number of different integer values allowed, respectively. For Korat and Korat with coverage pruning (Korat+), the table shows the number of explored vectors, together with how many of these are valid test cases (i.e., satisfying `repOk()`). We also indicate the time taken by both algorithms, and the number of classes covered (CC) for the corresponding scope (the covered classes are the same for Korat and Korat+, due to the soundness of the technique).

Scope	Korat	Korat+	Time Korat	Time Korat+	CC
0-2,0-2,3,3,3	1,121(91)	185(26)	0.331s	0.249s	8
0-4,0-4,3,3,3	1,485(91)	211(26)	0.277s	0.241s	8
0-4,0-4,4,4,3	14,679(320)	679(80)	0,422s	0,269s	10
0-5,0-5,5,5,4	1,274,977(5,456)	6,798(682)	2,39s	0,395s	10
0-5,0-5,5,5,5	6,692,357(24,211)	16,369(1,562)	10.961s	0.586s	10
0-7,0-7,7,7,6	-	1,453,804(111,974)	TIMEOUT	3,36s	10

Binomial Heap (Merge). Our second case study consists of generating test data for binomial heaps. A fundamental operation of binomial heaps is the merge of two heaps, which can be performed very efficiently for this structure. Assuming one is interested in testing such a routine, it is necessary to provide pairs of binomial heaps. The merging of two binomial heaps depends very much on how these are composed, and the degrees of their composing binomial trees. Considering equivalence class partitioning as the black box test criterion, the following predicates should provide a suitable coverage:

- the first heap is empty,
- the second heap is empty,
- the first heap has more elements than the second,
- both heaps have the same number of elements,
- the first heap has a larger degree than the second,
- both heaps have the same degree, and
- the heaps contain a tree with the same degree.

We have used the implementation of binomial heaps, with its corresponding `repOk()`, exactly as provided in the Korat distribution, replicated for the two binomial heaps. The domains for each of these have been defined disjoint, in the finitisation procedure. The experimental results regarding this case study are shown in the table below. The scope indicates the maximum number of elements both heaps might have. The keys in the nodes range from zero to this value

(repeated elements are allowed). The table shows the number of explored vectors, together with how many of these are valid test cases. We also indicate the number of equivalence classes covered, and the times taken by the two algorithms.

Scope	Korat	Korat+	Time Korat	Time Korat+	CC
2	348(36)	147(15)	0,42s	0,394s	6
3	5,389(784)	1,315(56)	0,656s	0,454s	10
4	150,448(14,400)	46,786(435)	1,436s	0,86s	10
5	3,125,314(876,096)	647,410(1,872)	16,347s	2,492s	10
6	274,808,123(57,790,404)	55,745,855(43,134)	1360,323s	178,072s	10

Directed Graphs. Our third case study corresponds to generating test cases for a routine manipulating a directed graph. The implementation of directed graphs is a standard object oriented implementation, consisting of a vector of vertices, each of which has a corresponding strictly sorted linked list, its adjacency list. Suppose that one is interested in generating case studies of varied arc "densities" and covering border cases; so, the combined graph characteristics considered for equivalence class partitioning could be the following:

- emptiness,
- density, and
- completeness.

The experimental results for this case study are shown in the table below. The scope indicates the exact number of nodes in the directed graph. As for the previous cases, the table shows the number of explored vectors, together with how many of these are valid test cases, the number of classes covered, etc. Notice that the number of valid cases grows too quickly, preventing us from reporting results for scopes higher than 3.

Scope	Korat	Korat+	Time Korat	Time Korat+	CC
2	1,624(382)	518(126)	0,343s	0,265s	4
3	372,861,255(47,672,840)	11,670,154(899,852)	1145,341s	37,854s	4

Weighted Directed Graphs. Our fourth case study extends the previous one, to generating test cases for *weighted* directed graphs. The graph implementation is an extension of the one described above, in which each entry in the adjacency list of a vertex has a corresponding weight.

Some typical algorithms on weighted directed graphs are calculations of transitive closure or minimal path information, as for instance using Floyd's algorithm. From the definition of minimal path some representative equivalence classes can be defined, based on properties of the graph:

- acyclicity,
- presence of negative weights, and
- connectedness of the graph.

They all play significant roles in the calculation of transitive closure or minimal path information. Thus, these are adequate predicates to consider for equivalence

class coverage. In order to also get cases of varied arc "densities" and cover border cases, we also take into account emptiness, density and completeness of the structure, as for the previous case study. The experimental results for this case study are shown in the table below. The scope indicates the exact number of nodes in the directed graph, and the range for weights. As for the previous cases, the table shows the number of explored vectors, together with how many of these are valid test cases, the number of classes covered, etc.

Scope	Korat	Korat+	Time Korat	Time Korat+	CC
2,-1-1	1,062(332)	984(256)	0.326s	0.324s	11
2,-2-2	2,272(1,542)	1,750(1,022)	0,632s	0,534s	11
3,-1-1	18,003,420(493,232)	17,815,155(304,982)	55.483s	52.759s	13
3,-2-2	33,122,848(15,612,660)	25,486,513(7,976,340)	278.562s	169.82s	13
3,-3-3	205,397,228(187,887,040)	103,127,315(85,617,142)	2812,665s	1333,942s	13

Search Tree (Delete). Our last case study is concerned with deletion in search trees. In this case, the test data to generate is composed of a combination of a search tree and a value to be deleted from it. The search tree implementation we considered is the one provided in the Korat distribution. The test case equivalence classes in this case correspond to the "position" of the value to be deleted in the tree; we have chosen the following cases:

- the value is not in the tree,
- the value is in the root,
- the value is in a leaf,
- the value is in a node with two (nonempty) subtrees,
- the value is in a node with a left subtree only, and
- the value is in a node with a right subtree only.

The experimental results for this case study are shown in the table below. The scope indicates the maximum number of nodes in the tree, the range for the size of the tree, and the number of keys allowed in the tree.

Scope	Korat	Korat+	Time Korat	Time Korat+	CC
3,0-3,3	534(45)	500(43)	0.272s	0.251s	8
3,0-3,4	1,152(148)	1,011(125)	0.255s	0.271s	8
3,0-3,6	4,290(822)	3,331(586)	0,423s	0,359s	8
3,0-3,8	12,144(2,760)	8,675(1,793)	0.661s	0.562s	8
5,0-5,8	477,888(29,416)	338,292(16,137)	1,607s	1,333s	9
6,0-6,9	4,597,299(167,814)	3,213,270(83,511)	7,529s	5,724s	9

4.1 Analysing the Assessment of our Case Studies

Let us briefly discuss now the results of our experimental analyses on our case studies. First, notice that we have chosen to report, for each of the case studies, the number of explored candidates, accompanied by the corresponding number

of valid candidates found. This is, in our opinion, the most reasonable measure to employ if one is interested in evaluating the level of pruning that our technique contributes to standard filtering. In our cases, these numbers reflect directly in running times, because our `eqClass()` routines, the most influencial (with respect to running time) part of the extra pruning section of our variant of Korat, do not increase in a noticeable way the running times of Korat for the scopes considered in these case studies. However, we only report the running time for generation. One would expect that this would also reflect in the time necessary for actually testing for the produced inputs. All the experiments were run on a 3.06GHz Intel Core 2 Duo with 4GB of RAM, and the reported data correspond to experiments that terminated within our timeout of 5 hours.

The performance of the technique, in this case implemented as a variant of Korat, depends greatly on the quality of `repOk()` and `eqClass()`, and how these relate to each other. For instance, in cases in which `eqClass()` needs to visit the whole structure in order to determine the equivalence class for the test case, there will be no extra pruning at all; this is due to the fact that the "next candidate" computation of Korat would have already advanced one of the fields observed by `eqClass()`, since it "observes everything". So, the technique provides better results when the test criterion under consideration is such that examining a small part of the structure one can determine a test case's equivalence class. This is exactly the case in our two first case studies, in which the technique exhibits a better profit.

Another important factor in the performance of our technique implemented as Korat+ compared to Korat is in the size of the "valid candidates" space over the search space. More precisely, when `repOK()` fails very often, i.e., when the conditions for valid structures are stronger, then Korat exploits its associated pruning mechanism. It is when `repOK()` succeedes more often than it fails when Korat+ contributes more to the pruning, since while Korat would advance to the next candidate with no pruning, our extra pruning mechanism would try to prune candidates corresponding to the just covered equivalence class. Notice that when for Korat the number of valid test cases is large in comparison with the number of explored candidates (`repOK()` succeedes more often), our extra pruning tends to contribute more to the pruning.

We have tried to foresee potential threats to the validity of our experimental results. We tried to be careful about the chosen case studies. Although our case studies correspond to relatively small pieces of code, they represent, in our opinion, rather natural testing situations in the context of the implementation of complex, heap allocated data structures (which is the main target for Korat). We have accompanied the presentation of each case study by a short justification of its appropriateness. We have included in our evaluation some case studies that have been successfully tackled by Korat, employing the same implementation available with Korat's distribution (for which `repOK()` routines are tailored to exploit Korat's search process).

One might argue that the equivalence classes used in these cases might prune too much, i.e., that these would show good pruning but would not be helpful

for finding bugs. We decided then to take the three case studies for which we achieved more pruning, and make an analysis of how good would the obtained test suites be for finding bugs. These case studies are *list as set*, *binomial heaps* and *search trees*. We took three programs, namely standard implementations of *listToSet*, *merge* and *deleteFromTree*, for these structures, and performed the following experiment. We used muJava [9] in order to generate all method mutants of these three programs, and employed the test cases produced by Korat, by Korat+ and optimal equivalence class coverage (i.e., "one per equivalence class"), to see how many mutants can be killed by each of these test suites. The mutants are those obtained by the application of 12 different method-level mutation operators, e.g., arithmetic, logical and relational operator replacements, etc. (see [11] for a complete list of method level mutation operators). Not all of these mutation operators were applicable to our programs (6 were applicable to *list as set*, 5 were applicable to *binomial heaps*, and 4 were applicable to *search trees*). The results obtained are shown in the tables at the end of this section. Each table shows the total number of mutants and how many remained live after testing using the corresponding test suite. Notice that the results for Korat correspond to optimal mutant killing, since their corresponding test suites are *bounded exhaustive* (i.e., Korat kills as many mutants as possible within the corresponding bounds). In order to obtain a test suite for optimal equivalence class coverage (one per equivalence class), we take the first test case of each equivalence class from the bounded exhaustive test suite produced by Korat. As these experiments show, we achieve better results compared to one per equivalence class, and as the bounds are increased we get closer to bounded exhaustive test suites. Our intuition of being somehow "in between" optimal equivalence class coverage and bounded exhaustive is supported by the results.

List as Set (49 mutants)			
Scope	Korat	Korat+	One Per Class
0-2,0-2,3,3,3	3	9	15
0-4,0-4,3,3,3	3	9	15
0-4,0-4,4,4,3	3	9	11
0-5,0-5,5,5,4	3	9	10
0-5,0-5,5,5,5	3	9	10

Binomial Heaps (117 mutants)			
Scope	Korat	Korat+	One Per Class
2	38	39	44
3	8	8	17
4	7	7	17
5	7	7	17
6	7	7	17

Search Trees (24 mutants)			
Scope	Korat	Korat+	One Per Class
3,0-3,3	2	2	2
3,0-3,4	2	2	2
3,0-3,6	2	2	2
3,0-3,8	2	2	2
5,0-5,8	0	0	2
6,0-6,9	0	0	2

5 Conclusions and Future Work

We have presented a technique for improving bounded exhaustive test case generation using a filtering approach, by incorporating black box test criteria and employing these for pruning the search of valid test inputs. The approach targets structurally complex inputs, and essentially consists of incorporating into the usual pruning processes present in test generation techniques, an extra pruning that skips parts of the search space when one is certain that only candidates of classes of inputs already covered would be found. We implemented this technique as a variant of Korat, a tool/algorithm that automatically generates test cases by a "generate and filter" mechanism [2] . We argued about the technique's correctness, and developed some case studies, whose associated experimental results enabled us to assess the benefits of the technique. The technique is somehow in between bounded exhaustive and "optimal" equivalence class coverage, and the actual "exhaustiveness" of the technique depends on the interaction of the test criterion (e.g., the adequacy of the predicates used for equivalence class coverage) and the generation procedure. This is reflected by the fact that, in our implementation, the performance depends on the quality of the `repOk()` and `eqClass()`, and how these relate to each other. In particular, when `eqClass()` roughly respects the order in which `repOk()` visits the fields of the structure, and in cases in which a relatively small part of it suffices to determining its equivalence class, the technique is more beneficial. We also found that standard Korat works well when the valid test cases are relatively few with respect to the number of general structures, i.e., when the restrictions for the structure to be valid are stronger. In these cases, `repOk()` fails often, and thus Korat's pruning improves the search significantly. On the contrary, when `repOk()` does not fail very often, Korat's pruning is not exercised much. These are the cases in which our technique shows more profit. For instance, structures such as directed acyclic graphs or linked lists show better results than structures such as red black or AVL trees.

Automatic test case generation is an active area of research. For the particular case of bounded exhaustive test case generation of structurally complex, heap allocated inputs, various tools have been proposed. Among these we may cite Java PathFinder [15], Alloy [5], CUTE [12] and obviously Korat. A thorough comparison between these tools, reported in [13], shows that Korat (seen as a kind of specialised solver) is generally the most efficient, justifying our selection of Korat for implementing the technique. Although our approach is implemented for Korat, the technique applies to other tools that perform bounded exhaustive generation by filtering. Examples of such tools would be Alloy [5], UDITA [4] (which also supports a generative approach) and AsmL [1].

As future work, we plan to develop a more significant evaluation of our technique, over larger source code than that used in the experiments presented in this paper (these experiments were limited to a number of case studies regarding heap allocated data structure implementations, and a few algorithms over these). We are also currently exploring the use of SAT based analysis for test case generation guided by test criteria, exploiting the scalability improvements

achieved in [3]. We plan to continue this line of work by exploring the parallelisation of the approach (e.g., by combining the pruning with mechanisms such as those in [14]), as well as by defining generic (i.e., not user provided) mechanisms for considering inputs to be similar. This would enable us to implement a similar technique, without the need for a user provided test criterion.

Acknowledgements

The authors would like to thank Darko Marinov and the anonymous referees for their valuable comments.

This work was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grant PICT 2006 No. 2484. The first author's participation was also supported through ANPCyT grant PICT PAE 2007 No. 2772.

References

1. Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Model-Based Testing with AsmL.NET. In: Proceedings of the 1st European Conference on Model-Driven Software Engineering (2003)
2. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated Testing based on Java Predicates. In: Proceedings of International Symposium on Software Testing and Analysis ISSTA 2002. ACM Press, New York (2002)
3. Galeotti, J.P., Rosner, N., López Pombo, C., Frias, M.: Analysis of invariants for efficient bounded verification. In: Proceedings of the 19th International Symposium on Software Testing and Analysis ISSTA 2010. ACM Press, Trento (2010)
4. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering ICSE 2010. ACM Press, Cape Town (2010)
5. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
6. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing. Wiley, Chichester (2001)
7. Khurshid, S., Marinov, D.: TestEra: Specification-Based Testing of Java Programs Using SAT. Automated Software Engineering 11(4) (2004)
8. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification and Object-Oriented Design. Addison-Wesley, Reading (2000)
9. Ma, Y.-S., Offutt, J., Kwon, Y.-R.: MuJava: An Automated Class Mutation System. Journal of Software Testing, Verification and Reliability 15(2) (2005)
10. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A Tool for Generating Structurally Complex Test Inputs. In: Proceedings of International Conference on Software Engineering ICSE 2007. IEEE Press, Los Alamitos (2007)
11. MuJava Home Page, `http://www.cs.gmu.edu/offutt/mujava/`
12. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Proceedings of the 5th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering ESEC/FSE 2005. ACM Press, New York (2005)

13. Siddiqui, J., Khurshid, S.: An Empirical Study of Structural Constraint Solving Techniques. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 88–106. Springer, Heidelberg (2009)
14. Siddiqui, J., Khurshid, S.: PKorat: Parallel Generation of Structurally Complex Test Inputs. In: Proceedings of the 2nd International Conference on Software Testing Verification and Validation ICST 2009. IEEE Computer Society, Los Alamitos (2009)
15. Visser, W., Pasareanu, C., Khurshid, S.: Test Input Generation with Java PathFinder. In: Proceedings of International Symposium on Software Testing and Analysis ISSTA 2004. ACM Press, New York (2004)
16. Xie, T., Marinov, D., Notkin, D.: Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering ASE 2004. IEEE Computer Society, Linz (2004)
17. Zhu, H., Hall, P., May, J.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys 29(4) (1997)