Efficient Bounded Exhaustive Input Generation from Program APIs

Mariano Politano[1,4]✉, Valeria Bengolea[1], Facundo Molina[3], Nazareno Aguirre[1,4], Marcelo F. Frias[2,4], and Pablo Ponzio[1,4]

[1] Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
[2] Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina
[3] IMDEA Software Institute, Madrid, Spain
[4] CONICET, Argentina

Abstract. Bounded exhaustive input generation (BEG) is an effective approach to reveal software faults. However, existing BEG approaches require a precise specification of the *valid* inputs, i.e., a `repOK`, that must be provided by the user. Writing `repOK`s for BEG is challenging and time consuming, and they are seldom available in software.

In this paper, we introduce BEAPI, an efficient approach that employs routines from the API of the software under test to perform BEG. Like API-based test generation approaches, BEAPI creates sequences of calls to methods from the API, and executes them to generate inputs. As opposed to existing BEG approaches, BEAPI does not require a `repOK` to be provided by the user. To make BEG from the API feasible, BEAPI implements three key pruning techniques: *(i)* discarding test sequences whose execution produces exceptions violating API usage rules, *(ii)* state matching to discard test sequences that produce inputs already created by previously explored test sequences, and *(iii)* the automated identification and use of a subset of methods from the API, called *builders*, that is sufficient to perform BEG.

Our experimental assessment shows that BEAPI's efficiency and scalability is competitive with existing BEG approaches, without the need for `repOK`s. We also show that BEAPI can assist the user in finding flaws in `repOK`s, by (automatically) comparing inputs generated by BEAPI with those generated from a `repOK`. Using this approach, we revealed several errors in `repOK`s taken from the assessment of related tools, demonstrating the difficulties of writing precise `repOK`s for BEG.

1 Introduction

Automated test generation approaches aim at assisting developers in crucial software testing tasks [2, 22], like automatically generating test cases or suites [6, 18, 10], and automatically finding and reporting failures [23, 19, 12, 20, 4, 13]. Many of these approaches involve random components, that avoid making a systematic exploration of the space of behaviors, but improve test generation efficiency [23, 19, 10]. While these approaches have been useful in finding a large number

of bugs in software, they might miss exploring certain faulty software behaviors due to their random nature. Alternative approaches aim at systematically exploring a very large number of executions of the software under test (SUT), with the goal of providing stronger guarantees about the absence of bugs [20, 4, 12, 14, 6, 18]. Some of these approaches are based on bounded exhaustive generation (BEG) [20, 4], which consists of generating all feasible inputs that can be constructed using bounded data domains. Common targets to BEG approaches have been implementations of complex, dynamic data structures with rich structural constraints (e.g., linked lists, trees, etc). The most widely-used and efficient BEG approaches for testing software [20, 4] require the user to provide a formal specification of the constraints that the inputs must satisfy –often a representation invariant of the input (`repOK`)–, and bounds on data domains [20, 4] –often called scopes. Thus, specification-based BEG approaches yield all inputs within the provided scopes that satisfy `repOK`.

Writing appropriate formal specifications for BEG is a challenging and time consuming task. The specifications must precisely capture the intended constraints of the inputs. Overconstrained specifications lead to missing the generation of valid inputs, which might make the subsequent testing stage miss the exploration of faulty behaviors of the SUT. Underconstrained specifications may lead to the generation of invalid inputs, which might produce false alarms while testing the SUT. Furthermore, sometimes the user needs to take into account the way the generation approach operates, and write the specifications in a very specific way for the approach to achieve good performance [4] (see Section 4). Finally, such precise formal specifications are seldom available in software, hindering the usability of specification-based BEG approaches.

Several studies show that BEG approaches are effective in revealing software failures [20, 16, 4, 33]. Furthermore, the *small scope hypothesis* [3], which states that most software faults can be revealed by executing the SUT on "small inputs", suggests that BEG approaches should discover most (if not all) faults in the SUT, if large enough scopes are used. The challenge that BEG approaches face is how to efficiently explore a huge search space, that often grows exponentially with respect to the scope. The search space often includes a very large number of invalid (not satisfying `repOK`) and isomorphic inputs [15, 28]. Thus, pruning parts of the search space involving invalid and redundant inputs is key to make BEG approaches scale up in practice [4].

In this paper, we propose a new approach for BEG, called BEAPI, that works by making calls to API methods of the SUT. Similarly to API-based test generation approaches [23, 19, 10], BEAPI generates sequences of calls to methods from the API (i.e., test sequences). The execution of each test sequence yielded by BEAPI generates an input in the resulting BEG set of objects. As usual in BEG, BEAPI requires the user to provide scopes for generation, which for BEAPI includes a maximum test sequence length. Brute force BEG from a user-provided scope would attempt to generate all feasible test sequences of methods form the API with up to a maximum sequence length. This is an intrinsically combinatorial process, that exhausts computational resources before completion even for

very small scopes (see Section 4). We propose several pruning techniques that are crucial for the efficiency of BEAPI, and allow it to scale up to significantly larger scopes. First, BEAPI executes test sequences and discards those that correspond to violations of API usage rules (e.g., throwing exceptions that indicate incorrect API usage, such as `IllegalArgumentException` in Java [17, 23]). Thus, as opposed to specification-based BEG approaches, BEAPI does not require a `repOK` that precisely describes valid inputs. In contrast, BEAPI requires minimum specification effort in most cases (including most of our case studies in Section 4), which consists of making API methods throw exceptions on invalid inputs (in the "defensive programming" style popularized by Liskov [17]). Second, BEAPI implements state matching [15, 28, 36] to discard test sequences that produce inputs already created by previously explored sequences. Third, BEAPI employs only a subset of the API methods to create test sequences: a set of methods automatically identified as builders [27]. Before test generation, BEAPI executes an automated builders identification approach [27] to find a smaller subset of the API that is sufficient to yield the resulting BEG set of inputs. Another advantage of BEAPI with respect to specification-based approaches is that it produces test sequences to create the corresponding inputs using methods from the API, making it easier to create tests from BEAPI's output [5].

We experimentally assess BEAPI, and show that its efficiency and scalability are comparable to those of the fastest BEG approach (Korat), without the need for `repOK`s. We also show that BEAPI can be of help in finding flaws in `repOK`s, by comparing the sets of inputs generated by BEAPI using the API against the sets of inputs generated by Korat from a `repOK`. Using this procedure, we found several flaws in `repOK`s employed in the experimental assessment of related tools, thus providing evidence on the difficulty of writing `repOK`s for BEG.

2 A Motivating Example

To illustrate the difficulties of writing formal specifications for BEG, consider `Apache`'s `NodeCachingLinkedList`'s (`NCL`) representation invariant shown in Figure 1 (taken from the ROOPS benchmark[5]). `NCL`s are composed of a main circular, doubly-linked list, used for data storage, and a cache of previously used nodes implemented as a singly linked list. Nodes removed from the main list are moved to the cache, where they are saved for future usage. When a node is required for an insertion operation, a cache node (if one exists) is reused (instead of allocating a new node). As usual, `repOK` returns true iff the input structure satisfies the intended `NCL` properties [17]. Lines 1 to 20 check that the main list is a circular doubly-linked list with a dummy head; lines 21 to 33 check that the cache is a null terminated singly linked list (and the consistency of size fields is verified in the process). This `repOK` is written in the way recommended by the authors of Korat [4]. It returns false as soon as it finds a violation of an intended property in the current input. Otherwise, it returns true at the end. This allows Korat to prune large portions of the search space, and improves its

[5] https://code.google.com/p/roops/

```
1  public boolean repOK() {
2    if (this.header == null) return false;
3    //  Missing constraint: the value of the sentinel node must be null
4    // if (this.header.value != null) return false;
5    if (this.header.next == null) return false;
6    if (this.header.previous == null) return false;
7    if (this.cacheSize > this.maximumCacheSize) return false;
8    if (this.size < 0) return false;
9    int cyclicSize = 0;
10   LinkedListNode n = this.header;
11   do {
12       cyclicSize++;
13       if (n.previous == null) return false;
14       if (n.previous.next != n) return false;
15       if (n.next == null) return false;
16       if (n.next.previous != n) return false;
17       if (n != null) n = n.next;
18   } while (n != this.header && n != null);
19   if (n == null) return false;
20   if (this.size != cyclicSize - 1) return false;
21   int acyclicSize = 0;
22   LinkedListNode m = this.firstCachedNode;
23   Set visited = new HashSet();
24   visited.add(this.firstCachedNode);
25   while (m != null) {
26       acyclicSize++;
27       if (m.previous != null) return false;
28       // Missing constraint: the value of cache nodes must be null
29       // if (m.value != null) return false;
30       m = m.next;
31       if (!visited.add(m)) return false;
32   }
33   if (this.cacheSize != acyclicSize) return false;
34   return true;
35 }
```

Fig. 1. `NodeCachingLinkedList`'s `repOK` from ROOPS

performance [4]. `repOK` suffers from underspecification: it does not state that the sentinel node and all cache nodes must have null values (lines 3-4 and 28-29, respectively). Mistakes like these are very common when writing specifications (see Section 4.3), and difficult to discover by manual inspection of `repOK`. These errors can have serious consequences for BEG. Executing Korat with `repOK` and a scope of up to 8 nodes produces 54.5 million `NCL` structures, while the actual number of valid `NCL` instances is 2.8 million. Clearly, this is a problem for Korat's performance, and for the subsequent testing of the SUT. In addition, the invalid instances generated might trigger false alarms in the SUT in many cases. We discovered these errors in `repOK` with the help of BEAPI: we automatically contrasted the structures generated using BEAPI and the `NCL`'s API, with those generated using Korat with `repOK`, for the same scope.

This example shows that writing sound and precise `repOK`s for BEG is difficult and time consuming. Fine-tuning `repOK`s to improve the performance of BEG (e.g., for Korat) is even harder. The main advantage of BEAPI is that it requires minimal specification effort to perform BEG. If API methods used for generation are correct, all generated structures are valid by construction. The programmer only needs to make sure that API methods throw exceptions when API usage

```
1  max.objects=3
2  int.range=0:2
3  # strings=str1,str2,str3
4  # omit.fields=NodeCachingLinkedList.DEFAULT_MAXIMUM_CACHE_SIZE
```

Fig. 2. BEAPI's scope definition for `NCL` (max. nodes 3)

rules are violated, in a defensive programming style [17]. In most cases, this requires checking very simple conditions on the inputs. In our example, the method to add an element to a `NCL` throws an `IllegalArgumentException` when is called with the `null` element (the implementation of the method takes care that the remaining `NCL` properties hold).

3 Bounded Exhaustive Generation from Program APIs

We now describe BEAPI's approach. We start with the definition of scope, then present BEAPI's optimizations, and we finally describe BEAPI's algorithm.

3.1 Scope Definition

The definition of scope in Korat involves providing bounded data domains for classes and fields of the SUT, since Korat explores the state space of feasible input candidates, and yields the set of inputs satisfying `repOK` as a result. Instead, BEAPI explores the search space of (bounded) test sequences that can be formed by making calls to the SUT's API. Thus, we have to provide data domains for the primitive types employed to make such calls, and a bound on the maximum size of the structures we want to keep, from those generated by such API calls. An example configuration file defining BEAPI's scope for the `NCL` case study is shown in Figure 2. The `max.objects` parameter specifies the maximum number of different objects (reachable from the root) that a structure is allowed to have. Test sequences that create a structure with a larger number of different objects (of any class) than `max.objects` will be discarded (and the structure too). In our example, this implies that BEAPI will not create `NCL`s with more than 3 nodes. Next, one has to specify the values that will be employed by BEAPI to invoke API routines that take primitive type parameters (e.g., elements to insert into the list). The `int.range` parameter allows one to specify a range of integers, which goes from 0 to 2 in Figure 2. One may also specify domains for other primitive types like floats, doubles and strings, by describing their values by extension. For example, line 3 shows how to define `str1`, `str2` and `str3` as the feasible values for String-typed parameters. Also, we can instruct BEAPI which fields to take into account for structure canonicalization, or which fields to omit (`omit.fields`). This allows the user to control the state matching process (see Section 3.2). For example, uncommenting line 4 would make BEAPI omit the `DEFAULT_MAXIMUM_CACHE_SIZE` in state matching, which in our example is a constant initialized to 20 in the class constructor. In this case, omitting the field does not change anything in terms of the different structures generated by

BEAPI, but in other cases omitting fields may have an impact. The configuration in Figure 2 is enough for BEAPI to generate NCLs with a maximum of 3 nodes, containing integers from 0 to 2 as values, which allowed us to mimic the structures generated by Korat for the same scope.

3.2 State Matching

In test generation with BEAPI, multiple test sequences often produce the same structure, e.g., inserting an element into a list and removing the element afterwards. BEAPI assumes that method executions are deterministic: any execution of a method with the same inputs yields the same results. For the generation of a bounded exhaustive set of structures, for each distinct structure `s` in the set, BEAPI only needs to save the first test sequence that generates `s`. All test sequences generated subsequently that also create `s` can be discarded. As BEAPI works by extending previously generated test sequences (Section 3.4), if we save many test sequences for the same structure, all these sequences would have to be extended with new routines in subsequent iterations of BEAPI, resulting in unnecessary computations. Hence, we implement state matching on BEAPI as follows. We store all the structures produced so far by BEAPI in a canonical form (see below). After executing the last routine `r(`p_1`,..,`p_k`)` of a newly generated test sequence `T`, we check whether any of `r`'s parameters hold a structure not seen before (not stored). If `T` does not create any new structure, it is discarded. Otherwise, `T` and the new structures it generates are stored by BEAPI.

We represent heap-allocated structures as labeled graphs. After the execution of a method, a (non-primitive typed) parameter p holds a reference to the root object r of a rooted heap (i.e. $p = r$), defined below.

Definition 1. *Let O be a set of objects, and P a set of primitive values (including null). Let F be the fields of all objects in O.*

- *A* heap *is a labeled graph $H = \langle O, E\rangle$ with $E = \{(o, f, v) | o \in O, f \in F, v \in O \cup P\}$.*
- *A* rooted heap *is a pair $RH = \langle r, H\rangle$ where $r \in O$, $H = \langle O, E\rangle$ is a heap, and for each $v' \in O \cup P$, v' is reachable from r through fields in F.*

The special case $p = null$ can be represented by a rooted heap with a dummy node and a dummy field pointing to *null*. In languages without explicit memory management (like Java), each object is identified by the memory address where is allocated. But changing the memory addresses of objects (while keeping the same graph structure) has no effect in the execution of a program. Heaps obtained by permutations of the memory addresses of their component objects are called *isomorphic heaps*. We avoid the generation of isomorphic heaps by employing a canonical representation for heaps [15, 4]. Rooted heaps can be efficiently canonicalized by an approach called *linearization* [15, 36], which transforms a rooted heap into a unique sequence of values.

Figure 3 shows the linearization algorithm used by BEAPI, a customized version that reports when objects exceed the scopes and supports ignoring object

```
1  int[] linearize(O root, Heap<O, E> heap, int scope, Regex omitFields) {
2    Map ids = new Map(); // maps nodes into their unique ids
3    return lin(root, heap, scope, ids, omitFields);
4  }
5  int[] lin(O root, Heap<O, E> heap, int scope, Map ids, Regex omitFields) {
6    if (ids.containsKey(root))
7      return singletonSequence(ids.get(root));
8    if (ids.size() == scope)
9      throw new ScopeExceededException();
10   int id = ids.size() + 1;
11   ids.put(root, id);
12   int[] seq = singletonSequence(id);
13   Edge[] fields = sortByField({ <root, f, o> in E }, omitFields);
14   foreach (<root, f, o> in fields) {
15     if (isPrimitive(o))
16       seq.add(uniqueRepresentation(o));
17     else
18       seq.append(lin(o, heap, scope, ids, omitFields));
19   }
20   return seq;
21 }
```

Fig. 3. Linearization algorithm

fields (for the original version see [36]). `linearize` starts a depth-first traversal of the heap from the root, by invoking `lin` in line 3. To canonicalize the heap, `lin` assigns different identifiers to the different objects it visits. Map `ids` stores the mapping between objects and unique object identifiers. When an object is visited for the first time, it is assigned a new unique identifier (lines 10-11), and a singleton sequence with the identifier is created to represent the object (line 12). Then, the object's fields, sorted in a predefined order (e.g., by name), are traversed and the linearization of each field value is constructed, and the result is appended to the sequence representing the current object (lines 13-19). A field storing a primitive value is represented by a singleton sequence with the primitive value (line 15-16). If a field references an object, a recursive call to `lin` converts the object into a sequence, which will be appended to the result (line 18). At the end of the loop, `seq` contains the canonical representation of the whole rooted heap starting at `root`, and is returned by `lin` (line 20). When an already visited object is traversed by a recursive call, the object must have an identifier already assigned in `ids` (line 6), and `lin` returns the singleton sequence with the object's unique identifier (lines 7). When more than `scope` objects are reachable from the rooted heap, `lin` returns an exception to report that the scope has been exceeded (lines 9-10). The exception will be employed later on by BEAPI to discard test sequences that create objects larger than allowed by the scope. `linearize` also takes as a parameter a regular expression `omitFields`, that matches the names of the fields that must be omitted during canonicalization (see Section 3.1). To omit such fields, we implemented `sortByField` (line 13) in such a way that it does not return the edges corresponding to fields whose names match `omitFields`. This in turn avoids saving the values of omitted fields in the sequence yielded by `linearize`. Finally, notice that linearization allows for efficient comparison of objects (rooted heaps): two objects are equal if and only if their corresponding sequences yielded by `linearize` are equal.

3.3 Builders Identification Approach

As the feasible combinations of methods grow exponentially with the number of methods, it is crucial to reduce the number of methods that BEAPI uses to produce test sequences. We employ an automated builders identification approach [27] to find a subset of API methods that are sufficient for the generation of the bounded exhaustive structure sets. We call such routines *builders*. The previous approach to identify a subset of sufficient builders from an API is based on a genetic algorithm, but is computationally expensive [27]. Here, we consider a simpler hill climbing approach (HC), that achieves better performance. HC may of course be less precise, as it may include some methods in the resulting set of builders that might not be needed to produce a bounded exhaustive set of structures. However, HC worked very well and consistently computed minimal sets of builders in our experiments (we checked that the set of builders computed by HC matched the set of builders we manually identified for each case study). Our goal here is to assess the impact of using builders for BEG from an API. Comparing the HC approach against existing techniques is left for future work.

Let `API`=$m_1, m_2, \ldots, m_n$ be the set of API methods. HC explores the search space of all subsets of methods from `API`. HC requires the user to provide a scope `s` (in the same way as in BEAPI). The fitness `f(sm)` of a given set `sm` of methods is the number of distinct structures (after canonicalization) that BEAPI generates using the set, for the given scope `s`. We also give priority in the fitness to sets of methods with less and simpler parameter types (see [27] for further details). The successors `succs(sm)` for a candidate `sm` are the sets `sm`$\cup\{m_i\}$, for each $m_i \in$ `API`. HC starts by computing the fitness of all singletons $\{c\}$ of constructor methods. The best of the singletons is set as the current candidate `curr`, and HC starts a typical iterative hill climbing process. At each iteration HC computes `f(succ)` for each `succ` $\in$ `succs(curr)`. Let `best` be the successor with the highest fitness value. Notice that `best` has exactly one more method than the best candidate of the previous iteration, `curr`. If `f(best) > f(curr)`, methods in `best` can be used to create a larger set of structures than those in `curr`. Thus, HC assigns `best` to `curr`, and continues with the next iteration. Otherwise, `f(best) <= f(curr)`, and `curr` already generates the largest possible set of structures (no method could be added that increases the number of generated structures from `curr`). At this point, `curr` is returned as the set of identified builders.

Notice that HC performs many invocations to BEAPI for builders identification. The key insight that makes builders identification feasible is that often builders identified for a relatively small scope are the same set of methods that are needed to create structures of any size. In other words, once the scope for builders computation is large enough, increasing the scope will yield the same set of builders as a result. This result resembles the small scope hypothesis for bug detection [3] (and transcoping [31]). A scope of 5 was enough for builders computation in all our case studies (we manually checked that the computed builders were the right ones in all cases). After builders are identified efficiently using a small scope, we can run BEAPI with the identified builders using a larger scope, for example, to generate bigger objects to exercise the SUT. In most of our case

```
1  BEAPI(List methods, int scope, Map<Type, List<Seq>> primitives, Regex omitFields) {
2    Map<Type, List<Seq>> currSeqs = new Map();
3    currSeqs.addAll({ T->L | T->L in primitives });
4    Set canonicalStrs = new Set();
5    for (int it=0; true; it++) {
6      Map<Type, List<Seq>> newSeqs = new Map();
7      boolean newStrs = false;
8      for (m(T1,...,Tn):Tr: methods) {
9        Map<Type, List<Seq>> seqsT1 = currSeqs.getSequencesForType(T1);
10       ...
11       Map<Type, List<Seq>> seqsTn = currSeqs.getSequencesForType(Tn);
12       for ((s1,...,sn): seqsT1 × ... ×seqsTn) {
13         Seq newSeq = createNewSeq(s1,...,sn,m);
14         o1,...,on,or,failure,exception = execute(newSeq);
15         if (failure) throw new ExecutionFailedException(newSeq);
16         if (exception) continue;
17         c1,...,cn,cr,outOfScope = makeCanonical(o1,...,on,or,scope,omitFields);
18         if (outOfScope) continue;
19         if (isReferenceType(T1) and !canonicalStrs.contains(c1)) {
20           canonicalStrs.add(c1);
21           newSeqs.addSeqForType(T1, newSeq);
22           newStrs = true;
23         }
24         ...
25         if (isReferenceType(Tr) and !canonicalStrs.contains(cr)) {
26           canonicalStrs.add(cr);
27           newSeqs.addSeqForType(Tr, newSeq);
28           newStrs = true;
29         }
30       }
31     }
32     if (!newStrs) break;
33     currSeqs.addAll(newSeqs);
34   }
35   return currSeqs.getAllSeqsAsList();
36 }
```

Fig. 4. BEAPI algorithm

studies, builders comprise a constructor and a single method to add elements to the structure. However, our automated builder identification approach showed that, for Red-Black Trees, a remove method was also required (for scopes greater than 3), since there are trees with a particular balance configuration (red and black coloring for the nodes) that cannot be constructed by just adding elements to the tree. In contrast, AVL trees, which are also balanced, do not require the remove method as a builder, and the class constructor and an add routine suffice. This shows that builders identification is non-trivial to perform manually, as it requires a very careful exploration of a very large number of structures and method combinations. Other structures that require more than two builders are binomial and Fibonacci heaps.

3.4 The BEAPI Approach

A pseudocode of BEAPI is shown in Figure 4. BEAPI takes as inputs a list of methods from an API, `methods` (the whole API, or previously identified builders); the scope for generation, `scope`; a list of test sequences to create values for each primitive type provided in the scope description, `primitives` (automat-

ically created from configuration options `int.range`, `strings`, etc., see Fig. 2); and a regular expression matching fields to be omitted in the canonicalization of structures, `omitFields`. Notice that methods from more than one class could be passed in `methods` if one wants to generate objects for several classes in the same execution of BEAPI, e.g., when methods from one class take objects from another class as parameters. BEAPI's map `currSeqs` stores, for each type, the list of test sequences that are known to generate structures of the type. `currSeqs` starts with all the primitive typed sequences in `primitives` (lines 2-3). At each iteration of the main loop (lines 5-34), BEAPI creates new sequences for each available method `m` (line 8), by exhaustively exploring all the possibilities for creating test sequences using `m` and inputs generated in previous iterations and stored in `currSeqs` (lines 9-30). The newly created test sequences that generate new structures in the current iteration are saved in map `newSeqs` (initialized empty in line 6); all the generated sequences are then added to `currSeqs` at the end of the iteration (line 33). If no new structures are produced at the current iteration (`newStrs` is false in line 32), BEAPI's main loop terminates and the list of all sequences in `currSeqs` is returned (line 35).

Let us now discuss the details of the for loop in lines 9-30. First, all sequences that can be used to construct inputs for `m` are retrieved in $\texttt{seqsT}_1$,...,$\texttt{seqsT}_n$. BEAPI explores each tuple $(\texttt{s}_1,...,\texttt{s}_n)$ of feasible inputs for `m`. Then, it executes `createNewSeq` (line 13), which constructs a new test sequence `newSeq` by performing the sequential composition of test sequences $\texttt{s}_1$,...,$\texttt{s}_n$ and routine `m`, and replacing `m`'s formal parameters by the variables that create the required objects in $\texttt{s}_1$,...,$\texttt{s}_n$. `newSeq` is then executed (line 14) and it either produces a failure (`failure` is set to true), raises an exception that represents an invalid usage of the API (`exception` is set to true), or its execution is successful and it creates new objects $\texttt{o}_1,\ldots,\texttt{o}_n,\texttt{o}_r$. In case of a failure, an exception is thrown and `newSeq` is presented to the user as a witness of the failure (line 15). If a different kind of exception is thrown, BEAPI assumes it corresponds to an API misuse (see below), discards the test sequence (line 16) and continues with the next candidate sequence. Otherwise, the execution of `newSeq` builds new objects $\texttt{o}_1,\ldots,\texttt{o}_n,\texttt{o}_r$ (or values of primitive types) that are canonicalized by `makeCanonical` (line 17) –by executing `linearize` from Figure 3 on each structure. If any of the structures produced by `newSeq` exceeds the scope, `makeCanonical` sets `outOfScope` to true, BEAPI discards `newSeq` and continues with the next one (line 18). If none of the above happens, `makeCanonical` returns canonical versions of $\texttt{o}_1,\ldots,\texttt{o}_n,\texttt{o}_r$ in variables $\texttt{c}_1,\ldots,\texttt{c}_n,\texttt{c}_r$, respectively. Afterwards, BEAPI performs state matching by checking that the canonical structure $\texttt{c}_1$ is of reference type and that it has not been created by any previous test sequence (line 19). Notice that `canonicalStrs` stores all of the already visited structures. If $\texttt{c}_1$ is a new structure, it is added to `canonicalStrs` (line 27), and the sequence that creates $\texttt{c}_1$, `newSeq`, is added to the set of test sequences producing structures of type $\texttt{T}_1$ (`newSeqs` in line 27). Also, `newStrs` is set to true to indicate that at least a new object has been created in the current iteration (line 22). This process is repeated for canonical objects $\texttt{c}_2,\ldots,\texttt{c}_n,\texttt{c}_r$ (lines 24-29).

BEAPI distinguishes failures from bad API usage based on the type of the exception (similarly to previous API based test generation techniques [23]). For example, `IllegalArgumentException` and `IllegalStateException` correspond to API misuses, and the remaining exceptions are considered failures by default. BEAPI's implementation allows the user to select the exceptions that correspond to failures and those that do not, by setting the corresponding configuration parameters. As mentioned in Section 2, BEAPI assumes that API methods throw exceptions when they fail to execute on invalid inputs. We argue that this is a common practice, called defensive programming [17], that should be followed by all programmers, as it results in more robust code and improves software testing in general [2] (besides helping automated test generation tools). We also argued in Section 2 that the specification effort required for defensive programming is much less than writing precise (and efficient) `repOK`s for BEG, and that this was true after manually inspecting the source code of our case studies. On the other hand, note that BEAPI can employ formal specifications to reveal bugs in the API, e.g., by executing `repOK` and check that it returns true on every generated object of the corresponding type (as in Randoop [23]). However, the specifications used for bug finding do not need to be very precise (e.g., the underspecified `NCL` `repOK` from Section 2 is fine for bug finding), or written in a particular way (as required by Korat). Other kinds of specifications that are weaker and simpler to write can also be used by BEAPI to reveal bugs, like violations of language specific contracts (e.g., `equals` is an equivalence relation in Java), metamorphic properties [7], user-provided assertions (`assert`), etc.

Another advantage of BEAPI is that, for each generated object, it yields a test sequence that can be executed to create the object. This is in contrast with specification based approaches (that generate a set of objects from `repOK`). Finding a sequence of invocations to API methods that create a specific structure is a difficult problem on its own, that can be rather costly computationally [5], or require significant effort to perform manually. Thus, often objects generated by specification based approaches are "hardwired" when used for testing a SUT (e.g., by using Java reflection), making tests very hard to understand and maintain, as they depend on the low-level implementation details of the structures [5].

4 Evaluation

In this section, we experimentally assess BEAPI against related approaches. The evaluation is organized around the following research questions:

RQ1 *Can BEG be performed efficiently using API routines?*
RQ2 *How much do the proposed optimizations impact the performance of BEG from the API?*
RQ3 *Can BEAPI help in finding discrepancies between `repOK` specifications and the API's object generation ability?*

As case studies, we employ data structures implementations from four benchmarks: three employed in the assessment of existing testing tools (Korat [4],

Kiasan [9], FAJITA [1]), and ROOPS. These benchmarks cover diverse implementations of complex data structures, which are a good target for BEG. We choose these as case studies because the implementations come equipped with `repOK`s, written by the authors of the benchmarks. The experiments were run on a workstation with an Intel Core i7-8700 CPU (3.2 Ghz) and 16Gb of RAM. We set a timeout of 60 minutes for each individual run. To replicate the experiments, we refer the reader to the paper's artifact [25].

4.1 RQ1: Efficiency of Bounded Exhaustive Generation from APIs

For RQ1 we assess whether or not BEAPI is fast enough to be a useful BEG approach, by comparing it to the fastest BEG approach, Korat [32]. The results of the comparison are summarized in Table 1. For each technique, we report generation times (in seconds), number of generated and explored structures, for increasingly large scopes. Due to space reasons, we show a representative sample of the results (we try to maintain the same proportion of good and bad cases for each technique in the data we report). We include the largest successful scope for each technique; the execution times for the largest scopes are in boldface in the table. In this way, should scalability issues arise, they can be easily identified. For the complete report of the results visit the paper's website [26]. To obtain proper performance results for BEAPI, we extensively tested the API methods of the classes to ensure they were correct for this experiment. We did not try to change the `repOK`s in any way because that would change the performance of Korat, and one of our goals here is evaluating the performance of Korat using `repOK`s written by different programmers. Differences in explored structures are expected, since the corresponding search spaces for Korat and BEAPI are different. However, for the same case study and scope, one would expect both approaches to generate the same number of valid structures. This is indeed the case in most experiments, with notable exceptions of two different kinds. Firstly, there are cases where `repOK` has errors; these cases are grayed out in the tables. Secondly, the slightly different notion of *scope* in each technique can cause discrepancies. This only happens for Red-Black Trees (`RBT`) and Fibonacci heaps (`FibHeap`), which are shown in boldface. In these cases certain structures of size n can only be generated from larger structures, with insertions followed by removals and then insertions again to trigger specific balance rearrangements. BEAPI discards generated sequences as soon as they exceed the maximum structure size, hence it cannot generate these structures.

In terms of performance, we have mixed results. In the Korat benchmark, Korat shows better performance in 4 out of 6 cases. In the FAJITA benchmark, BEAPI is better in 3 out of 4 cases. In the ROOPS benchmark, BEAPI is better in 5 out of 7 cases. In the Kiasan benchmark, Korat is faster in 6 of the 7 cases. We observe that BEAPI shows a better performance in structures with more restrictive constraints such as `RBT` and Binary Search Trees (`BST`); often these cases have a smaller number of valid structures. Cases where the number of valid structures grows faster with respect to the scope, such as doubly-linked lists (`DLList`), are better suited for Korat. More structures means BEAPI has

Table 1. Efficiency assessment of BEAPI against Korat

	Class	S	Time		Generated		Explored	
			Korat	BEAPI	Korat	BEAPI	Korat	BEAPI
KORAT	DLList	6	0.24	7.11	55987	55987	521904	335930
		7	2.31	**108.08**	960800	960800	9875550	6725609
		9	**1333.88**	TO	435848050		5325611829	
	FibHeap	6	1.26	5.95	573223	54159	1641562	379125
		7	32.87	**115.44**	17858246	898394	54268866	7187167
		8	**1415.77**	TO	654214491		2105008180	
	BinHeap	7	0.26	25.32	107416	107416	261788	859337
		8	0.85	**163.39**	603744	603744	1323194	5433706
		11	**2558.32**	TO	2835325296		2985116257	
	BST	10	131.18	49.10	223191	223191	216680909	2231922
		11	**1137.17**	199.46	974427	974427	1679669258	10718710
		12	TO	**1341.86**		4302645		51631754
	SLList	7	5.76	17.87	137257	137257	2055596	960807
		8	8.16	**256.49**	2396745	2396745	40701876	19173969
		9	**190.45**	TO	48427561		919451065	
	RBT	11	40.54	33.42	51242	39942	53141999	878743
		12	220.77	79.45	146073	112237	276868584	2693710
		13	**1277.67**	**689.06**	428381	314852	1454153331	8186175
FAJITA	BinTree	10	73.73	51.34	223191	223191	218675679	2231922
		11	**634.114**	265.57	974427	974427	1689480455	10718710
		12	TO	**1578.72**		4302645		51631754
	AVL	10	163.50	1.92	7393	7393	349178307	73942
		11	**1271.23**	5.80	20267	20267	2504382415	222950
		13	TO	**45.45**		145206		1887693
	RBT	11	58.74	19.72	51242	39942	75814869	878743
		12	318.57	63.16	146073	112237	385422689	2693710
		13	**1779.83**	**206.66**	428381	314852	1957228527	8186175
	BinHeap	7	.77	44.452	107416	107416	1447594	859337
		8	5.96	**97.08**	603744	603744	13329584	5433706
		10	**1174.91**	TO	117157172		2064639445	
ROOPS	AVL	5	3.54	0.05	1107	62	12277946	317
		6	**213.63**	.009	3969	157	701862289	950
		13	TO	**46.71**		145206		1887693
	NCL	6	0.65	2.27	800667	11196	805921	134364
		7	8.797	33.89	2739128	160132	16443824	2241862
		8	**205.596**	**769.63**	381367044	2739136	381381493	43826192
	BinTree	3	0.173	0.02	65376	15	65596	50
		4	**37.546**	0.05	121853251	51	121855507	210
		12	TO	**966.41**		4302645		51631754
	LList	7	0.51	12.62	137257	137257	1410799	960807
		8	7.64	**295.94**	2396745	2396745	26952027	19173969
		9	**176.69**	TO	48427561		591734656	
	RBT	11	69.87	31.02	51242	39942	75814869	878743
		12	361.88	81.03	146073	112237	385422689	2693710
		13	**2007.29**	**697.06**	428381	314852	1957228527	8186175
	FibHeap	4	1.851	0.13	131444	335	5681553	1683
		5	**346.275**	0.70	21629930	4381	1295961583	26297
		7	TO	**129.01**		898394		7187167
	BinHeap	6	1.04	1.31	7602	7602	3202245	53222
		7	17.47	13.06	107416	107416	64592184	859337
		8	**448.48**	**96.94**	603744	603744	1483194820	5433706
KIASAN	BST	11	12.184	204.83	974427	974427	62669069	10718710
		12	65.305	**1235.67**	4302645	4302645	308229505	51631754
		14	**1751.4**	TO	86211885		7438853941	
	DLL	7	0.614	18.09	137257	137257	2326622	960807
		8	9.824	**257.42**	2396745	2396745	45449534	19173969
		9	**245.787**	TO	48427561		1015587001	
	RBT	7	10.76	0.78	911	561	44832139	7866
		8	**283.33**	1.57	2489	1657	1044561963	26526
		12	TO	**84.51**		112237		2693710
	DisjSetFast	6	0.198	0.89	52165	544	117456	22890
		7	1.209	**8.26**	1545157	4397	3398383	246288
		9	**1402.376**	TO	2201735557		4715569321	
	StackList	6	0.128	4.35	55987	55987	56008	335930
		7	0.517	**83.06**	960800	960800	960828	6725609
		9	**212.919**	TO	435848050		435848095	
	BHeap	7	0.654	53.78	3206861	458123	3221407	3665089
		8	8.98	**1221.59**	64014472	8001809	64124432	72016409
		9	**202.804**	TO	1447959627		1449279657	
	TreeMap	5	.55	24.95	40526	34276	162375	1028287
		6	2.85	**866.71**	1207261	1098397	3381725	46132686
		8	**1980.70**	TO	1626500673		2671020961	

to create more test sequences in each successive iteration, which makes its performance suffer more in such cases. As expected, the way `repOK`s are written has a significant impact in Korat's performance. For example, for binomial heaps (`BinHeap`) Korat reaches scope 8 with `Roops`' `repOK`, scope 10 with `FAJITA`'s `repOK`, and scope 11 with `Korat`'s `repOK` (all equivalent in terms of generated structures). In most cases, `repOK`s from the `Korat` benchmark result in better performance, as these are fine-tuned for usage with Korat. Case studies with errors in `repOK`s are grayed out in the table, and discussed further in Section 4.3. Notice that errors in `repOK`s can severely affect `Korat`'s performance.

4.2 RQ2: Impact of BEAPI's Optimizations

Table 2. Execution times (sec) of BEAPI under different configurations.

ROOPS					
Class	S	SM/BLD	SM	BLD	NoOPT
AVL	3	.02	.04	.34	-
	4	.03	.07	102.16	-
	5	.05	.11	-	-
	13	46.71	657.17	-	-
NCL	3	.04	1.31	1.37	7.96
	4	.10	9.59	52.17	-
	5	.34	40.54	-	-
	8	769.63	-	-	-
BinTree	3	.02	.04	.23	33.84
	4	.05	.08	85.32	-
	5	.11	.16	-	-
	12	966.41	2281.42	-	-
LList	3	.03	.09	.26	-
	4	.07	.48	115.27	-
	5	.18	118.75	-	-
	8	295.94	-	-	-
RBT	3	.04	.04	39.11	-
	4	.11	.09	-	-
	5	.22	.14	-	-
	12	81.03	2379.44	-	-
FibHeap	3	.04	.09	.94	-
	4	.13	.20		-
	5	.70	1.13		-
	7	129.01	243.36	-	-
BinHeap	3	.05	.11	2.03	18.38
	4	.09	.34	-	-
	5	.26	.96	-	-
	8	96.94	220.18	-	-

Real World					
Class	S	SM/BLD	SM	BLD	NoOPT
NCL	3	.10	.47	-	-
	4	.41	3.48	-	-
	5	3.33	-	-	-
	6	73.78	-	-	-
TSet	3	.03	.07	56.82	-
	11	21.52	86.06	-	-
	12	69.98	276.85	-	-
	13	226.66	887.83	-	-
TMap	3	.11	.25	-	-
	4	.75	2.36	-	-
	5	15.97	57.64	-	-
	6	839.87	2901.37	-	-
LList	3	.02	.13	.64	-
	6	.96	258.85	-	-
	7	12.98	-	-	-
	8	286.21	-	-	-
HMap	3	.10	11.49	-	-
	4	.55	-	-	-
	5	5.33	-	-	-
	6	119.87	-	-	-

In RQ2 we assess the impact each of BEAPI's proposed optimizations has in BEG. For this, we assess the performance of four different BEAPI configurations: SM/BLD is BEAPI with state matching (SM) and builder identification (BLD) enabled; SM is BEAPI with only state matching (SM) enabled; BLD is BEAPI with only builders (BLD) identification enabled; NoOPT has both optimizations disabled. The left part of Table 2 summarizes the results of this experiment for the `ROOPS` benchmark; the right part reports preliminary results on five "real world" implementations of data structures: `LinkedList` (21 API methods), `TreeSet` (22 API methods), `TreeMap` (32 methods) and `HashMap` (29

methods) from `java.util`, and `NCL` from Apache Collections (20 methods). As most real world implementations, these data structures do not come equipped with `repOK`s, hence we only employed them in this RQ.

The brute force approach (NoOPT) performs poorly even for the easiest case studies and very small scopes. These scopes are too small and often not enough if one wants to generate high quality test suites. State matching is the most impactful optimization, greatly improving by itself the performance and scalability all around (compare NoOPT and SM results). As expected, builders identification is much more relevant in cases where the number of methods in the API is large (more than 10), and remarkably in the real world data structures (with 20 or more API methods). SM/BLD is more than an order of magnitude faster than SM in `AVL` and `RBT`, and it reaches one more scope in `NCL` and `LList`. The remaining classes of `ROOPS` have just a few methods, and the impact of using builders is relatively small. The conclusions drawn from `ROOPS` apply to the other three benchmarks (we omit their results here for space reasons, visit the paper's website for a complete report [26]). In the real world data structures, using precomputed builders allowed SM/BLD to scale to significantly larger scopes in all cases but `TreeMap` and `TreeSet`, where it significantly improves running times. Overall, the proposed optimizations have a crucial impact in BEAPI's performance and scalability, and both should be enabled to obtain good results.

On the cost of builders identification. Due to space reasons we report builders identification times in the paper's website [26]. For the conclusions of this section, it is sufficient to say that scope 5 was employed for builders identification in all cases, and that the maximum runtime of the approach was 65 seconds in the four benchmarks (`ROOPS' SLL`, 11 methods), and 132 seconds in the real world data structures (`TreeMap`, 32 methods). We manually checked that the identified methods included a set of sufficient builders in all cases. Notice that BEG is often performed for increasingly larger scopes, and the identified builders can be reused across executions. Thus, builder identification times are amortized across different executions, which makes it difficult to calculate how much builder identification times add to BEAPI running times in each case. So we did not include builder identification times in BEAPI running times in any of the experiments. Notice that, for the larger scopes, which arguably are the most important, builders identification time is negligible in relation to generation times.

4.3 RQ3: Analysis of Specifications using BEAPI

RQ3 addresses whether BEAPI can be useful in assisting the user in finding flaws in repOKs, by comparing the set of objects that can be generated using the API and the set of objects generated from the `repOK`. We devised the following automated procedure. First, we run BEAPI to generate a set `SA` of structures from the API, and Korat to generate a set `SR` from `repOK`, using the same scope for both tools. Second, we canonicalize the structures in both `SA` and `SR` using linearization (Section 3.2). Third, we compare sets `SA` and `SR` for equality. Differences in this comparison point out a mismatch between `repOK` and the API. There are three possible outputs for this automated procedure. If $\texttt{SA} \subset \texttt{SR}$, it is possible

Table 3. Summary of flaws found in `repOKs` using BEAPI

Bench.	Class	Error Description	Type
Korat	RBTree	Color of root should not be red	under
Roops	NCL	Key values in the cache should be set to null	under
		Key value of the dummy node in the main list should be null	under
	BinTree	Parent of root node should be null	under
	RBT	Color of root should not be red	under
	AVL	Height computation is wrong (leaves are assigned the wrong value)	error
		Repeated key values should not be allowed	under
	FibHeap	Left and right fields of nodes should not be null	under
		Min node should always contain the minimum value in the heap	under
		If a node has no child its degree should be zero	under
		Child nodes should have smaller keys than their parents	under
		Parent fields of all nodes are forced to be null	over
		Heap with min node set to null is rejected	over
Kiasan	DisjSetFast	The rank of the root can be invalid	under
	BinaryHeap	The first position of an array (dummy) may contain an element	under
Fajita	AVL	Height computation is wrong (leaves are assigned the wrong value)	error

that the API generates a subset of the valid structures, that `repOK` suffers from underspecification (missing constraints), or both. In this case, the structures in `SR` that do not belong to `SA` are witnesses of the problem, and the user has to manually analyze them to find out where the error is. Here, we report the (manually confirmed) underspecification errors in `repOKs` that are witnessed by the aforementioned structures. In contrast, when $\texttt{SR} \subset \texttt{SA}$, it can be the case that the API generates a superset of the valid structures, that `repOK` suffers from overspecification (`repOK` is too strong), or both. The structures in `SA` that do not belong to `SR` might point out to the root of the error, and again they have to be manually analyzed by the user. We report the (manually confirmed) overspecification errors in `repOKs` that are witnessed by these structures. Finally, it can be the case that there are structures in `SR` that do not belong to `SA`, and there are structures (distinct than the former ones) in `SA` that do not belong to `SR`. These might be due to faults in the API, flaws in the `repOK`, or both. We report the manually confirmed flaws in `repOKs` witnessed by such structures simply as errors (`repOK` describes a different set of structures than the one it should). Notice that differences in the scope definitions for the approaches might make sets `SA` and `SR` differ. This was only the case in the `RBT` and `FibHeap` structures, where BEAPI generated a smaller set of structures for the same scope than Korat due to balance constraints (as explained in Section 4.1). However, these "false positives" can be easily revealed, since all the structures generated by Korat were always included in the structures generated by BEAPI if a larger scope was used for the latter approach. Using this insight we manually discarded the "false positives" due to scope differences in `RBT` and `FibHeap`.

The results of this experiment are summarized in Table 3. We found out flaws in 9 out of 26 `repOKs` using the approach described above. The high number of

flaws discovered evidences that problems in `repOK`s are hard to find manually, and that BEAPI can be of great help for this task.

5 Related Work

BEG approaches have been shown effective in achieving high code coverage and finding faults, as reported in various research papers [20, 16, 4, 33]. Our goal here is not to assess yet again the effectiveness of BEG suites, but to introduce an approach that is straightforward to use in today's software because it does not require the manual work of writing formal specifications of the properties of the inputs (e.g., `repOK`s). Different languages have been proposed to formally describe structural constraints for BEG, including Alloy's relational logic (in the so-called declarative style), employed by the TestEra tool [20]; and source code in an imperative programming language (in the so-called operational style), as used by Korat [4]. The declarative style has the advantage of being more concise and simpler for people familiar with it, however this knowledge is not common among developers. The operational style can be more verbose, but as specifications and source code are written in the same language this style is most of the time preferred by developers. UDITA [11] and HyTeK [29] propose to employ a mix of the operational and the declarative styles to write the specifications, as parts of the constraints are often easier to write in one style or the other. With precise specifications both approaches can be used for BEG. Still, to use these approaches developers have to be familiar with both specification styles, and take the time and effort required to write the specifications. Model checkers like Java Pathfinder [34] (JPF) can also perform BEG, but the user has to manually provide a "driver" for the generation: a program that the model checker can use to generate the structures that will be fed to the SUT afterwards. Writing a BEG driver often involves invoking API routines in combination with JPF's nondeterministic operators, hence the developer must familiarize with such operators and put in some manual effort to use this approach. Furthermore, JPF runs over a customized virtual machine in place of Java's standard JVM, so there is a significant overhead in running JPF compared to the use of the standard JVM (employed by BEAPI). The results of a previous study [32] show that JPF is significantly slower than Korat for BEG. Therein, Korat has been shown to be the fastest and most scalable BEG approach at the time of publication [32]. This in part can be explained by its smart pruning of the search space of invalid structures and the elimination of isomorphic structures. In contrast, BEAPI does not require a `repOK` and works by making calls to the API.

An alternative kind of BEG consists of generating all inputs to cover all feasible (bounded) program paths, instead of generating all feasible bounded inputs. This is the approach of systematic dynamic test generation, a variant of symbolic execution [14]. This approach is implemented by many tools [13, 12, 24, 8], and has been successfully used to produce test suites with high code coverage, reveal real program faults, and for proving memory safety of programs.

Kiasan [9] and FAJITA [1] are also white-box test case generation approaches that require formal specifications and aim for coverage of the SUT.

Linearization has been employed to eliminate isomorphic structures in traditional model checkers [15, 28], and also in software model checkers [35]. A previous study experimented with state matching in JPF and proposed several approaches for pruning the search space for program inputs using linearization, for both concrete and symbolic execution [35]. As stated before, concrete execution in JPF requires the user to provide a driver. The symbolic approach attempts to find inputs to cover paths of the SUT; we perform BEG instead. Linearization has also been employed for test suite minimization [36].

6 Conclusions

Software quality assurance can be greatly improved thanks to modern software analysis techniques, among which automated test generation techniques play an outstanding role [6, 18, 10, 23, 19, 12, 20, 4, 13]. Random and search-based approaches have shown great success in automatically generating test suites with very good coverage and mutation metrics, but their random nature does now allow these techniques to precisely characterize the families of software behaviors that the generated tests cover. Systematic techniques such as those based on model checking, symbolic execution or bounded exhaustive generation, cover a precise set of behaviors, and thus can provide specific correctness guarantees.

In this paper, we presented BEAPI, a technique that aims at facilitating the application of a systematic technique, bounded exhaustive input generation, by producing structures solely from a component's API, without the need for a formal specification of the properties of the structures. BEAPI can generate bounded exhaustive suites from components with implicit invariants, and reduces the burden of providing formal specifications, and tailoring the specifications for improved generation. Thanks to a number of optimizations, including an automated identification of builder routines and a canonicalization/state matching mechanism, BEAPI can generate bounded exhaustive suites with a performance comparable to that of the fastest specification-based technique Korat [4]. We have also identified the characteristics of a component that may make it more suitable for a specification-based generation, or an API-based generation.

Finally, we have shown how specification based approaches and BEAPI can complement each other, depicting how BEAPI can be used to assess `repOK` implementations. Using this approach, we found a number of subtle errors in `repOK` specifications taken from the literature. Thus, techniques that require `repOK` specifications (e.g, [30]), as well as techniques that require bounded-exhaustive suites (e.g., [21]) can benefit from our presented generation technique.

Acknowledgements This work was partially supported by ANPCyT PICTs 2017-2622, 2019-2050, 2020-2896, an Amazon Research Award, and by EU's Marie Sklodowska-Curie grant No. 101008233 (MISSION). Facundo Molina's work is also supported by Microsoft Research, through a LA PhD Award.

References

1. Abad, P., Aguirre, N., Bengolea, V.S., Ciolek, D.A., Frias, M.F., Galeotti, J.P., Maibaum, T., Moscato, M.M., Rosner, N., Vissani, I.: Improving test generation under rich contracts by tight bounds and incremental SAT solving. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013. pp. 21–30. IEEE Computer Society (2013)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
3. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the "small scope hypothesis". Tech. rep., MIT CSAIL (10 2002)
4. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. In: Frankl, P.G. (ed.) Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002. pp. 123–133. ACM (2002)
5. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: Combining symbolic execution and search-based testing for programs with complex heap inputs. In: Bultan, T., Sen, K. (eds.) Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017. pp. 90–101. ACM (2017)
6. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008)
7. Chen, T.Y., Kuo, F.C., Liu, H., Poon, P.L., Towey, D., Tse, T.H., Zhou, Z.Q.: Metamorphic testing: A review of challenges and opportunities. ACM Comput. Surv. **51**(1) (jan 2018)
8. Christakis, M., Godefroid, P.: Proving memory safety of the ANI windows image parser using compositional exhaustive testing. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8931, pp. 373–392. Springer (2015)
9. Deng, X., Robby, Hatcliff, J.: Kiasan: A verification and test-case generation framework for java based on symbolic execution. In: Leveraging Applications of Formal Methods, Second International Symposium, ISoLA 2006, Paphos, Cyprus, 15-19 November 2006. p. 137. IEEE Computer Society (2006)
10. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. pp. 416–419. ACM (2011)
11. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. pp. 225–234. ACM (2010)

12. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 213–223. ACM (2005)
13. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)
14. Godefroid, P., Sen, K.: Combining model checking and testing. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 613–649. Springer (2018)
15. Iosif, R.: Symmetry reduction criteria for software model checking. In: Bosnacki, D., Leue, S. (eds.) Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2318, pp. 22–41. Springer (2002)
16. Khurshid, S., Marinov, D.: Checking java implementation of a naming architecture using testera. Electron. Notes Theor. Comput. Sci. **55**(3), 322–342 (2001)
17. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edn. (2000)
18. Luckow, K.S., Pasareanu, C.S.: Symbolic pathfinder v7. ACM SIGSOFT Softw. Eng. Notes **39**(1), 1–5 (2014)
19. Ma, L., Artho, C., Zhang, C., Sato, H., Gmeiner, J., Ramler, R.: GRT: an automated test generator using orchestrated program analysis. In: Cohen, M.B., Grunske, L., Whalen, M. (eds.) 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. pp. 842–847. IEEE Computer Society (2015)
20. Marinov, D., Khurshid, S.: Testera: A novel framework for automated testing of java programs. In: 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA. p. 22. IEEE Computer Society (2001)
21. Molina, F., Ponzio, P., Aguirre, N., Frias, M.: EvoSpex: An evolutionary algorithm for learning postconditions. In: Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering ICSE 2021, Virtual (originally Madrid, Spain), 23-29 May 2021 (2021)
22. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley Publishing, 3rd edn. (2011)
23. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007. pp. 75–84. IEEE Computer Society (2007)
24. Pham, L.H., Le, Q.L., Phan, Q., Sun, J.: Concolic testing heap-manipulating programs. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 442–461. Springer (2019)
25. Politano, M., Bengolea, V., Molina, F., Aguirre, N., Frias, M.F., Ponzio, P.: *Efficient Bounded Exhaustive Input Generation from Program APIs* paper's artifact. https://doi.org/10.5281/zenodo.7574758
26. Politano, M., Bengolea, V., Molina, F., Aguirre, N., Frias, M.F., Ponzio, P.: *Efficient Bounded Exhaustive Input Generation from Program APIs* paper's website. https://sites.google.com/view/bounded-exhaustive-api

27. Ponzio, P., Bengolea, V.S., Politano, M., Aguirre, N., Frias, M.F.: Automatically identifying sufficient object builders from module apis. In: Hähnle, R., van der Aalst, W.M.P. (eds.) Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11424, pp. 427–444. Springer (2019)
28. Robby, Dwyer, M.B., Hatcliff, J., Iosif, R.: Space-reduction strategies for model checking dynamic software. Electron. Notes Theor. Comput. Sci. **89**(3), 499–517 (2003)
29. Rosner, N., Bengolea, V., Ponzio, P., Khalek, S.A., Aguirre, N., Frias, M.F., Khurshid, S.: Bounded exhaustive test input generation from hybrid invariants. SIGPLAN Not. **49**(10), 655–674 (oct 2014)
30. Rosner, N., Geldenhuys, J., Aguirre, N., Visser, W., Frias, M.F.: BLISS: improved symbolic execution by bounded lazy initialization with SAT support. IEEE Trans. Software Eng. **41**(7), 639–660 (2015)
31. Rosner, N., Pombo, C.G.L., Aguirre, N., Jaoua, A., Mili, A., Frias, M.F.: Parallel bounded verification of alloy models by transcoping. In: Cohen, E., Rybalchenko, A. (eds.) Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8164, pp. 88–107. Springer (2013)
32. Siddiqui, J.H., Khurshid, S.: An empirical study of structural constraint solving techniques. In: Breitman, K.K., Cavalcanti, A. (eds.) Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5885, pp. 88–106. Springer (2009)
33. Sullivan, K.J., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: Software assurance by bounded exhaustive testing. In: Avrunin, G.S., Rothermel, G. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004. pp. 133–142. ACM (2004)
34. Visser, W., Mehlitz, P.C.: Model checking programs with java pathfinder. In: Godefroid, P. (ed.) Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3639, p. 27. Springer (2005)
35. Visser, W., Pasareanu, C.S., Pelánek, R.: Test input generation for java containers using state matching. In: Pollock, L.L., Pezzè, M. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006. pp. 37–48. ACM (2006)
36. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: 19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria. pp. 196–205. IEEE Computer Society (2004)