Efficient Bounded Exhaustive Input Generation from Program APIs

Anonymous Authors

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

In this paper, we introduce BEAPI, an efficient approach that employs routines from the API of the software under test to perform BE input generation. Like API-based test generation approaches, BEAPI creates sequences of calls to methods from the API (test sequences), and executes them to generate inputs. As opposed to existing BE approaches, BEAPI doesn't require a `repOK` to be provided by the user. To make BE generation from the API feasible, we propose three key pruning techniques: (i) BEAPI discards test sequences whose execution produces exceptions violating API usage rules, (ii) BEAPI implements state matching to discard test sequences that produce inputs already created by previously explored test sequences, (iii) BEAPI only employs a subset of methods from the API called builders –an automatically identified subset of routines that is sufficient to generate all valid BE inputs.

Our experimental assessment shows that BEAPI's efficiency and scalability is competitive with existing BE 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`. In this way, we revealed several errors in `repOK`s taken from the assessment of related tools, demonstrating the difficulties of writing precise `repOK`s for BE generation.

1 Introduction

Automated test generation approaches aim at assisting developers in crucial software testing tasks [2, 22], like automatically generating or making it easier to create tests 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 very 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]. One of these approaches is

bounded exhaustive (BE) generation [20, 4], which consists in generating all feasible structures that can be constructed using bounded data domains. Common targets to BE 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 BE approaches for testing software are black-box [20, 4]. They require the user to provide a formal specification of the constraints that the structures must satisfy –most frequently a representation invariant of the structure (`repOK`)–, and the bounds on data domains [20, 4] –often called scopes. Thus, black-box BE approaches generate all the structures within the provided scopes that satisfy `repOK`.

Several studies show that BE approaches are effective in revealing software failures [20, 16, 4, 31]. Furthermore, the so-called *small scope hypothesis* [3] – which states that most software failures can be revealed by executing the SUT on "small inputs"– suggests that, if large enough scopes are used, BE approaches should be capable of revealing most (if not all the) faults in the SUT. The challenge that BE approaches face is how to efficiently explore a huge search space, that in the worst case grows exponentially w.r.t. to the scopes. The search space often includes a very large number of invalid (not satisfying `repOK`) and isomorphic structures [15, 26]. Thus, pruning parts of the search space involving invalid and redundant structures is key to make BE approaches scale up in practice [4]. Writing appropriate formal specifications for BE generation is a challenging and time consuming task. The specifications must precisely capture the intended set of constraints on the structures. Overconstrained specifications lead to missing the generation of a portion of the valid structures, 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 structures (i.e., structures not satisfying the intended constraints), which might produce false alarms while testing the SUT. Furthermore, sometimes the user has to take into account the way the generation approach operates, and write the specifications in a very specific way, such that the approach can achieve good performance [4] (also see Section 4). Finally, such precise formal specifications are seldom available in software, hindering the usability of black-box BE approaches.

In this paper, we propose a new approach for BE generation, called BEAPI, that works by making calls to API routines of the SUT. Like API-based test generation approaches [23, 19, 10], BEAPI creates sequences of calls to methods from the API, often called test sequences [23], and executes them to generate structures. As usual with BE generation, the user must provide BEAPI with the scopes for generation. As opposed to black-box BE approaches, BEAPI does not require a formal specification of the properties of the structures. Brute force generation would attempt to generate all the feasible test sequences from routines up to the maximum length, and return all the structures generated by the test sequences as a result. However, this is an intrinsically combinatorial problem, and this procedure exhausts the computational resources before completing 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 larger scopes. First, BEAPI executes test sequences and discards those that produce exceptions violating API usage rules (e.g. `IllegalArgumentException` and `IllegalStateException` in Java [17, 23]). Second, BEAPI implements state matching [15, 26, 35] to discard test sequences that produce structures already created by previously explored test sequences. Third, BEAPI employs only a subset of the API routines to create test sequences: a set of routines automatically identified as builders [25]. That is, before generation BEAPI executes an automated builders identification approach to find a smaller subset of the API that is sufficient to build a BE set of structures.

Besides not requiring formal specifications, another advantage of BEAPI w.r.t. black-box approaches is that, for each generated structure, BEAPI produces a test sequence to construct the structure using the API. This makes debugging easier, as it's much easier for programmers to understand a test sequence that creates an structure, than the low level details of the structure [5].

We experimentally assess BEAPI, and show that its efficiency and scalability is competitive with the fastest BE approach (Korat), without the need for specifications. We also show that BEAPI is useful to find flaws in representation invariants of classes (`repOK`s), by automatically comparing sets of inputs generated with BEAPI from those generated using a `repOK`. Using BEAPI in this way, we found several flaws in `repOK`s taken from the experimental assessment of related tools, providing additional evidence that writing `repOK`s for BE generation is a difficult task.

2 A Motivating Example

To illustrate the difficulties of writing formal specifications for BE generation, consider the representation invariant (`repOK`) of `Apache`'s `NodeCachingLinkedList`[1] (`NCL` for short) shown in Figure 1. This is a `repOK` from the ROOPS benchmark[2]. `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. Thus, when a node is required for an insertion operation, a cache node (if one exists) is reused (instead of allocating a new node). The goal is to avoid garbage collection overhead for applications that perform a large number of insertions and removals on the list. `repOK` returns true iff the input structure satisfies the structural properties of `NCL`. 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). Notice that, `repOK` is written in the way recommended for the Korat BE generation approach: it returns false as soon as it finds a violation of an intended property in the current input [4]. Otherwise, it returns true at the end. Such `repOK`s allow Korat to prune large portions of the search space, and greatly improve its efficiency [4].

[1] https://commons.apache.org/
[2] 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

This `repOK` suffers from underspecification: constraints stating that the sentinel and the cache nodes must have null values are missing (lines 3-4 and 28-29, respectively). Mistakes like these are very common when writing specifications (see Section 4.3), and difficult to discover manually. This might have serious consequences for BE generation. 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. This significantly hurts the performance of Korat and of the subsequent testing of the SUT. In addition, the invalid instances generated might trigger false alarms in the SUT in many cases. Notice that, we've found the errors in `repOK` with the help of BEAPI. To achieve this, we generated structures with BEAPI using `NCL`'s API, and with Korat using the `repOK`, for the same scope (see Section 4.3 for details). An automated comparison among the sets of structures generated by the approaches reported that a very large number of `NCL`'s with sentinel and cache nodes with null values were generated by Korat but not by BEAPI, hence pointing out to the problems in `repOK`.

This example shows that writing sound and precise `repOK`s for BE generation is difficult and time consuming. Fine-tunning `repOK` to improve the performance of BE generation (e.g., for Korat) is even harder. The main advantage of BEAPI

is that it does not require any specification for BE input generation. If the API routines used for generation are correct, all generated structures are valid by construction. Weaker specifications can be used as oracles for testing the SUT, like fully-automated oracles that come "for free" in the code as exceptions, violations of language specific contracts (e.g., `equals` is an equivalence relation in Java), (automatically computed) metamorphic properties [7], user provided assertions in property-based tests [32], or any combination of these.

3 Bounded Exhaustive Generation from Program APIs

In this section we describe our approach, BEAPI. We start with the definition of the scopes in Section 3.1, then we present BEAPI optimizations (Sections 3.2 and 3.3), and finally BEAPI's algorithm (Section 3.4).

3.1 Scope definition

```
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)

The definition of the scopes in Korat involves providing bounded data domains for classes and fields of `NCL`, since Korat explores the state space of feasible `NCL`s candidates, and yields the set of structures satisfying `repOK` as a result. Instead, BEAPI explores the search space of (bounded) test sequences that can be formed by making calls to `NCL`'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 structures we want to keep from those generated by such API calls. An example configuration file defining BEAPI's scopes 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 containing a larger number of different objects (of any class) than `max.objects` will be discarded (and the structure too). In our example it 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, etc.). The `int.range` parameter allows to specify a range of integers, which goes from 0 to 2 in Figure 2. One could also specify domains for other primitive types like floats, doubles and strings by describing their values by extension. For example, the figure defines `str1`, `str2` and `str3` as the feasible values for String-typed parameters. Also, we can inform BEAPI which are the relevant field to consider for structure canonicalization, or which fields from the structures to omit (`omit.field`). 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 be important. The configuration in Figure 2 is enough for BEAPI to generate `NCL`s 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 scopes (some lines are commented out because they are not needed for this). The configuration files used in our experimental section are similar to the one shown here.

3.2 State matching

In test generation with BEAPI often many test sequences produce the same structure, e.g., inserting an element in a list and removing the element afterwards. BEAPI assumes that method executions are deterministic: any execution of a routine with the same inputs yields the same results. We observe that, for each distinct structure `s` we only need to save the first test sequence that generates `s` (and the structure itself). All test sequences generated subsequently that also create `s` can be discarded. Notice that, 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 many 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(`$\mathtt{p}_1$`,..,`$\mathtt{p}_k$`)` 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 like Java or C#, each object is identified by the memory address where it is located. Changing the memory addresses where objects are allocated has no effect from the program point of view, as the programmer doesn't have control of the low-level representation of the memory (unlike other languages like C). 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, 35], which transforms a rooted heap into a unique sequence of values.

A custom version of the linearization algorithm from [35], modified to inform when objects exceed the scopes and to support ignoring object fields, is shown in Figure 3. `linearize` starts a depth-first traversal of the heap from the root (line 3). `linearize` assigns different object identifiers to each visited objects. When an object is visited for the first time, it is assigned a new unique identifier (lines 10-11) and a singleton sequence `seq` with the object identifier es created to represent the object (line 12). `ids` stores the mapping between objects and unique object identifiers. 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 appended to the sequence representing the object, `seq` (lines 13-19). If a field stores a primitive value (line 15), a singleton sequence representing the value is to `seq` (line 16). If the field references an object, a recursive call to `linearize` will transform it into a sequence, which will be added to `seq` (line 18). At the end of the loop, `seq` contains the canonical representation of the whole rooted heap starting at `root`, and it's returned by the algorithm (line 20). When an already visited object is found by a recursive call, i.e., it has an identifier already assigned in `ids`, the algorithm returns the singleton sequence with the object's unique identifier (lines 6-7). If more than `scope` objects are reachable from the rooted heap, `linearize` returns an exception to inform that the scopes have been exceeded (lines 9-10). This behavior is particular to our approach, as the exception will be employed by BEAPI to discard test sequences that create objects larger than allowed by the scopes. Notice that, linearization allows for efficient comparison of objects (rooted heaps): two objects are equals if and only if their corresponding sequences yielded by `linearize` are equal.

`linearize` also takes as a parameter a regular expression `omitfields` that matches the names of the fields that must be omitted during object canonicalization (see Section 3.1). It uses `omitfields` to match against field names in `sortByField` (line 13). `sortByField` does not return the edges corresponding to the names that match `omitfields`. This in turn avoids saving the values of omitted fields in the sequence yielded by `linearize`.

3.3 Builders identification approach

As the feasible combinations of routines grow exponentially with the number of available routines, it is crucial to reduce as much as possible the number of routines that BEAPI uses to produce test sequences. Hence, we employ an automated builders identification approach [25] to find a subset of the API routines that are sufficient for the generation of the whole set of BE structures within the scopes. We call such routines builders. Existing approaches to identify a subset of builders from an API are based on a genetic algorithm, but are computationally expensive [25]. Here, we consider a simpler hill climbing approach (HC) that is faster, although in theory it could be less precise, in the sense that it

```
1  int[] linearize(Node root, Heap <O,E>, int scope, Regex omitfields) {
2    Map ids = new Map(); // maps nodes into their unique ids
3    return lin(root, <O,E>, int scope);
4  }
5  int[] lin(Node root, Heap <O,E>, int scope) {
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, <O,E>));
19   }
20   return seq;
21 }
```

Fig. 3. Linearization algorithm

may yield a larger than needed set of builders in some cases (that is, it might include some methods that can be omitted and we could still obtain a BE set of structures with the remaining methods). HC worked very well and always computed a minimal set of builders in our experiments. In any case, the focus here is on the impact of using builder routines for BE generation from an API, not on builder computation approaches. Comparing The hill climbing approach against existing techniques is left for future work.

Let `API`=$m_1, m_2, \ldots, m_n$ be the set of available 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 BEAPI). The fitness of a given set of methods `sm`, `f(sm)`, is the number of distinct structures that BEAPI generates using the set, for scope `s`. We also give priority in the fitness to sets of methods with less and simpler parameter types, but omit the details here (see [25]). The successors for a candidate `sm`, `succs(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)`, the set of methods `best` can be used to create a larger set of BE structures than `curr`. Thus, HC assigns `best` to `curr`, and continues to 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`). Hence, the algorithm ends and returns `curr` as the identified builders.

HC performs many invocations to BEAPI for builders identification. Thus, the only way to make HC fast we use the transcoping insight [29]. We run HC using a small scope, but this scope is enough for the identified builders to be the "real

builders", that is, if we increase the scope the identified builders remain the same (and often they remain the same in the unbounded case). Thus, 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. Providing BEAPI with builders significantly reduces the number of combinations of routines that BEAPI has to explore, and it significantly impacts its performance and scalability (see Section 4).

In most of our case studies, builders comprise one constructor and a single method to add elements to the structure. However, builder identification is not trivial, as exemplified by the Red-Black Trees data structure. For Red-Black Trees we also need a remove method for BE generation and scopes greater than 3, since there exist structures with a particular balance configuration (red and black coloring for the nodes) that cannot be constructed by only adding elements to the tree. On the other hand, 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 tricky 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 heaps and Fibonacci heaps.

3.4 The BEAPI approach

A pseudocode of BEAPI is presented in Figure 4. BEAPI takes as inputs a list of routines, `routines`, from an API (the whole API, or previously identified builders); the scope for generation, `scope` (`max.scope` from Figure 2); a list of test sequences, `primitives`, to create values provided in the scope description for each primitive typed parameters (constructed automatically from configuration options `int.range`, `strings`, etc., from Figure 2); and a regular expression describing fields to be omitted in the canonicalization of structures, `omitf` (configuration option `omit.fields`). BEAPI's map `currSeqs` stores, for each type, the list of test sequences that are known to generate structures of the type.

At the beginning, `currSeqs` starts with all the primitive typed sequences in `primitives` (lines 2-3). At each iteration of its main loop (lines 5-34), BEAPI creates new sequences by employing each routine `m` in `routines` (line 8) and exhaustively explores all new forms of creating test sequences using `m` and inputs from `currSeqs` (lines 9-30). The test sequences that create new structures in the current iteration are saved in map `newSeqs` (initialized empty in line 6), and then added to `currSeqs` at the end of the iteration (line 33). When 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).

Lets us now discuss the details of how test sequences are created for a routine `m` in the for loop in lines 9-30. First, all sequences that can be used to construct inputs for `m` (with the right types) are retrieved in $\texttt{seqsT}_1$,...,$\texttt{seqsT}_n$. BEAPI explores each tuple $(\texttt{s}_1,...,\texttt{s}_n)$ of feasible inputs for `m` (in the cartesian product of $\texttt{seqsT}_1$,...,$\texttt{seqsT}_n$ in lines 12-29). Then, it executes `createNewSeq`

```
1  BEAPI(List routines, int scope, Map<Type, List<Seq>> primitives, Regex omitf) {
2    Map<Type, List<Seq>> currSeqs = new Map();
3    currSeqs.addAll(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: routines) {
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) return newSeq;
16         if (exception) continue;
17         c1,...,cn,cr,outOfScope = makeCanonical(o1,...,on,or,scope,omitf);
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

(line 13), which constructs a new test sequence `newSeq` by performing the sequential composition of test sequences s_1,...,s_n and routine `m`, and replacing `m`'s formal parameters by the variables that create the required objects in s_1,...,s_n. `newSeq` is then executed (line 14) and it either (i) produces a failure (and boolean variable `failure` is set to true), (ii) raises an exception (that is stored in variable `exception`), or (iii) it's execution is successful and it creates new objects $o_1,...,o_n,o_r$. In case of a failure, `newSeq` is presented to the user as a witness and BEAPI terminates (line 15). If an exception that does not represent a failure is thrown, e.g., from an invalid usage of the API, the test sequence is discarded (lines 16) and BEAPI continues with the next candidate sequence. Otherwise, the execution of `newSeq` builds new objects $o_1,...,o_n,o_r$ (or values of primitive types) that are canonicalized by `makeCanonical` (line 17) –it applies the `linearize` algorithm of Figure 3 to each structure. If any of the structures produced by `newSeq` exceeds the scopes, `makeCanonical` sets `outOfScope` to true, and BEAPI discards `newSeq` and continues with the next one (line 18). This ensures BEAPI never creates objects larger than the given scopes. If none of the above happens, `makeCanonical` returns canonical versions of $o_1,...,o_n,o_r$ in variables $c_1,...,c_n,c_r$, respectively. Afterwards, BEAPI performs state match-

ing by checking that the canonical structure c_1 is of reference type and that it has not been created by any previous test sequence (line 19). The set `canonicalStrs` stores all the already visited structures. If c_1 is a new structure, it's added to `canonicalStrs` (line 27), and the sequence that creates c_1, `newSeq`, is added to the set of test sequences producing structures of type T_1 (`newSeqs` in line 27). Also, `newStrs` is set to true to indicate that at least an object has been created in the current iteration (line 22). This process is repeated for canonical objects $c_2, \ldots, c_n, c_r$ (lines 24-29).

BEAPI distinguishes failures from other kind of exceptions (like API misuses) based on the type of the exception. For example, `IllegalArgumentException` and `IllegalStateException` correspond to API misuses, whereas the remaining exceptions are considered failures by default. However, in BEAPI's prototypical implementation the user can choose the kind of exceptions that are classified as failures and those that are not, by setting the corresponding configuration parameters. This is the approach that previous API based test generation techniques follow [23]. Additionally, invariant properties (e.g., weak invariants) can be manually provided by the user, whose violations are considered failures (e.g., by annotating a method with @CheckRep as in Randoop [23]).

A distinguishing characteristic of BEAPI in comparison to specification-based BE approaches is the output that is produced. Specification based approaches return a set of objects that satisfy `repOK` and are within the provided bounds. A problem that arises in this context is that these objects may not necessarily be reproducible using the API of the corresponding component (e.g., as illustrated by our motivating example of Section 2). But even when the `repOK` is precise, finding a sequence of invocations to API routines 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. In addition, tests resulting from a direct encoding of the objects generated by specification-based approaches, for example by using Java reflection to "hardwire" the value of each object field, are difficult to understand and depend on low-level implementation details of the structure. This often makes the tests brittle and more difficult to maintain since changes to the implementation of the structure might break the tests. In contrast, the test sequences generated by BEAPI are easier to understand, and can often tolerate implementation changes in the underlying structure [5].

4 Evaluation

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

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

In RQ1 we evaluate whether BEAPI is fast enough to perform BE generation for increasingly large scopes. For this, we compare BEAPI (with optimizations enabled) against the fastest BE generation approach, Korat [30]. In other words, we would like to know if BEAPI is efficient enough to be a useful alternative to Korat. In RQ2 we assess the individual impact of each of BEAPI's optimizations for BE generation. Thus, we run four different configurations of BEAPI in all case studies for increasingly large scopes. We call SM/BLD to BEAPI with state matching (SM) and builder identification (BLD) enabled; SM to BEAPI with only state matching (SM) enabled ; BLD to BEAPI with only builders (BLD) identification enabled; NoOPT has both optimizations disabled. RQ3 assesses how useful BEAPI can be in assisting the user in finding flaws in specifications. Thus, we analyzed the repOKs of our case studies with the help BEAPI to see if any errors in repOKs could be revealed.

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 BE generation. We choose these benchmarks because the implementations come equipped with `repOK`s written by the authors. The experiments were run in 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. For the detailed experimental results, and instructions to replicate the experiments visit the paper's website[3].

4.1 RQ1: Efficiency of Bounded Exhaustive Generation from APIs

The results of the comparison of BEAPI against Korat are summarized in Table 1. To obtain proper performance results for `BEAPI`, i.e., such that it does not miss the generation of valid structures nor it generates invalid structures, we extensively tested and fixed all the faults we've found in the API routines before running this experiment. We report the generation times (in seconds), the number of generated and explored structures for each technique, for the different subjects and increasingly large scopes. Due to space reasons, we show a representative sample of the results. We always include the largest successful scope for each technique; the execution times of the largest scopes are reported in boldface in the table. In this way, should scalability issues arise, they can be easily identified. Visit the paper's website for a full report of the results.

Differences in the numbers of structures explored by the techniques are expected, since the search spaces are different. For the same case study and scope, one would expect both approaches to generate the same number of structures. This is indeed the case in most experiments, with notable exceptions of two different kinds. Firstly, there are cases where the `repOK` has errors; these cases are grayed out in the tables. Secondly, the slightly different notion of *scope* in each technique can cause discrepancies. The case studies where this happens, RBT and FibHeap, are shown in boldface. In these cases certain structures of

[3] https://sites.google.com/view/bounded-exhaustive-api/

Table 1. Efficiency assessment of BEAPI against Korat

	Class	S	Time		Generated		Explored	
			Korat	BEAPI	Korat	BEAPI	Korat	BEAPI
KORAT	DDList	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
KIASAN	**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
	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	
	DDL	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	

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 given maximum structure size, and therefore it cannot generate these kind of 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 Red-Black Trees (`RBT`) and Binary Search Trees (`BST`), as these cases have the smaller number of valid structures. Cases where the number of valid structures grows faster w.r.t. the scopes, such as doubly-linked lists, are better suited for Korat. More structures means that BEAPI has to create more test sequences for each of the next iterations, which makes the performance of BEAPI suffer more in such cases.

The results also point out that, as expected, the way `repOK`s are written has a significant influence 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` (and these `repOK`s are equivalent in terms of generated structures). In most cases, `repOK`s from the `Korat` benchmark result in better performance for Korat. Case studies with errors in `repOK`s (grayed out in the table) are discussed further in Section 4.3. Notice that errors in `repOK`s can seriously affect `Korat`'s performance.

4.2 RQ2: Impact of BEAPI's Optimizations

The left part of Table 2 summarizes the results of executing the different BEAPI configurations in 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 (20 methods). As most real world data structures, these case studies do not come equipped with `repOK`s, hence we could only assess them as part of this RQ.

The brute force approach (NoOPT) performs poorly even for the easiest case studies and very small scopes. These scopes are not good enough for producing 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 (at least more than 10), and remarkably in the real world data structures (number of API methods is greater or equal than 20). In cases from the `ROOPS` benchmark, when there are more than 10 routines, using precomputed builders significantly contributes to the efficiency of the generation. For example, (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 routines, and therefore the impact of using builders is small. The conclusions for `ROOPS`

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	-	-	-

also apply to the other three benchmarks, hence we omit their results here for space reasons (they can be found in the paper's website). In the real world data structures, precomputed builders allow SM/BLD to scale to significantly larger scopes in all cases but `TreeMap` and `TreeSet`, where they significantly improves the running times (w.r.t. SM). Overall, the optimizations have a crucial impact in BEAPI's performance, and for BEAPI to achieve good scalability w.r.t. the scopes both optimizations must be enabled.

On the cost of builders identification. For space reasons we do not report detailed builders identification times here (these can be found in the paper's website). For the conclusions of this section, it is sufficient to say that scope 5 was enough for builders identification in all cases, and the maximum runtime of the approach was 65 seconds for the benchmarks (`ROOPS' SLL`, 11 API methods), and 132 seconds for the real world data structures (`java.util.TreeMap`, 32 methods). We manually checked that the identified routines included a set of sufficient builders in all cases. Notice that, BE generation 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, and hence it is difficult to calculate how much builder identification times contribute to BEAPI running times in each case. So we do not incorporate builder identification times to BEAPI running times in any of the experiments. Notice also that, for larger scopes, which are very important for bug finding, builders identification time is negligible in relation to BE generation time.

4.3 RQ3: Analysis of Specifications using BEAPI

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

For this experiment, we analyzed the repOKs provided with the benchmarks with the help of BEAPI. To achieve this, we devised the following automated procedure. We ran BEAPI to generate a BE set of structures `SA` from the API, and Korat to generate a BE set `SR` from `repOK`, using the same scope for both tools. We (automatically) canonicalize the structures in both `SA` and `SR` using the linearization approach of Section 3.2. Then, we compare sets `SA` and `SR` for equality. Differences in this comparison point out a mismatch between `repOK` and the API. If $\texttt{SA} \subset \texttt{SR}$, it is possible 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` can be witnesses of the error, and we manually analyzed them to find out the cause of the error. 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, the `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 we manually analyzed them. 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 manually confirmed flaws in `repOKs` witnessed by such structures as errors (`repOK` describes a different set structures than it should). Notice that, differences in the

scope descriptions of the approaches might make these sets differ; we manually discarded such "false positives" when we found them.

The results of the experiment are summarized in Table 3. We've found out flaws in 9 out of 26 `repOK`s 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

BE generation approaches have been shown effective in achieving high code coverage and finding faults, as reported in various research papers [20, 16, 4, 31]. Our goal here is not to assess yet again the effectiveness of BE 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.

Different languages have been proposed to formally describe structural constraints for BE generation, including Alloy's relational logic (in the so-called declarative style), employed by tool TestEra [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 [27] 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 BE generation. 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 [33] (JPF) can also perform BE generation, 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 BE 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 an previous study [30] show that JPF is significantly slower and less scalable than Korat for BE generation. Therein, Korat has been shown to be the fastest and most scalable BE generation approach at the time of publication [30]. This in part can be explained by its smart pruning of the search space of invalid structures and the elimination of isomorphic structures. The approach presented here, BEAPI, does not require formal specifications and works using only the routines defined in the API of the SUT. Writing precise specifications for BE generation is not easy, as evidenced by our analysis of existing specifications (Section 4.3).

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

Linearization has been employed to eliminate isomorphic structures in traditional model checkers [15, 26], and also in software model checkers [34]. 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 [34]. 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 BE generation instead. Linearization has also been employed for test suite minimization [35].

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 BE input generation, cover a precise set of behaviors, and thus can provide specific guarantees of correctness.

In this paper, we presented BEAPI, a technique that aims at facilitating the application of a systematic technique, BE input generation, by producing structures solely from a component's API, without the need for a formal specification 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, [28]), as well as techniques that require bounded-exhaustive suites (e.g., [21]) can benefit from our presented generation technique.

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). https://doi.org/10.1109/ICST.2013.46, https://doi.org/10.1109/ICST.2013.46
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016). https://doi.org/DOI: 10.1017/9781316771273
3. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the "small scope hypothesis". Tech. rep. (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). https://doi.org/10.1145/566172.566191, https://doi.org/10.1145/566172.566191
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). https://doi.org/10.1145/3092703.3092715, https://doi.org/10.1145/3092703.3092715
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), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
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). https://doi.org/10.1145/3143561, https://doi.org/10.1145/3143561
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). https://doi.org/10.1007/978-3-662-46081-8"˙21, https://doi.org/10.1007/978-3-662-46081-8_21
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). https://doi.org/10.1109/ISoLA.2006.60, https://doi.org/10.1109/ISoLA.2006.60
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). https://doi.org/10.1145/2025113.2025179, https://doi.org/10.1145/2025113.2025179

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). https://doi.org/10.1145/1806799.1806835, https://doi.org/10.1145/1806799.1806835
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). https://doi.org/10.1145/1065010.1065036, https://doi.org/10.1145/1065010.1065036
13. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012). https://doi.org/10.1145/2093548.2093564, https://doi.org/10.1145/2093548.2093564
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). https://doi.org/10.1007/978-3-319-10575-8“˙19, https://doi.org/10.1007/978-3-319-10575-8_19
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). https://doi.org/10.1007/3-540-46017-9“˙5, https://doi.org/10.1007/3-540-46017-9_5
16. Khurshid, S., Marinov, D.: Checking java implementation of a naming architecture using testera. Electron. Notes Theor. Comput. Sci. **55**(3), 322–342 (2001). https://doi.org/10.1016/S1571-0661(04)00260-9, https://doi.org/10.1016/S1571-0661(04)00260-9
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). https://doi.org/10.1145/2557833.2560571, https://doi.org/10.1145/2557833.2560571
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). https://doi.org/10.1109/ASE.2015.102, https://doi.org/10.1109/ASE.2015.102
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). https://doi.org/10.1109/ASE.2001.989787, https://doi.org/10.1109/ASE.2001.989787

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). https://doi.org/10.1109/ICSE.2007.37, https://doi.org/10.1109/ICSE.2007.37
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). https://doi.org/10.1007/978-3-030-30942-8"˙27, https://doi.org/10.1007/978-3-030-30942-8_27
25. 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). https://doi.org/10.1007/978-3-030-16722-6"˙25, https://doi.org/10.1007/978-3-030-16722-6_25
26. 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). https://doi.org/10.1016/S1571-0661(05)80009-X, https://doi.org/10.1016/S1571-0661(05)80009-X
27. 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). https://doi.org/10.1145/2714064.2660232, https://doi.org/10.1145/2714064.2660232
28. 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). https://doi.org/10.1109/TSE.2015.2389225, https://doi.org/10.1109/TSE.2015.2389225
29. 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). https://doi.org/10.1007/978-3-642-54108-7"˙5, https://doi.org/10.1007/978-3-642-54108-7_5
30. 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). https://doi.org/10.1007/978-3-642-10373-5“¨5, https://doi.org/10.1007/978-3-642-10373-5_5
31. 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). https://doi.org/10.1145/1007512.1007531, https://doi.org/10.1145/1007512.1007531
32. Tillmann, N., Schulte, W.: Parameterized unit tests. SIGSOFT Softw. Eng. Notes **30**(5), 253–262 (sep 2005). https://doi.org/10.1145/1095430.1081749, https://doi.org/10.1145/1095430.1081749
33. 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). https://doi.org/10.1007/11537328“¨5, https://doi.org/10.1007/11537328_5
34. 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). https://doi.org/10.1145/1146238.1146243, https://doi.org/10.1145/1146238.1146243
35. 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). https://doi.org/10.1109/ASE.2004.10056, http://doi.ieeecomputersociety.org/10.1109/ASE.2004.10056