Science of Computer Programming 238 (2024) 103153

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original software publication

# BEAPI: A tool for bounded exhaustive input generation from APIs

Mariano Politano [a,*], Valeria Bengolea [a], Facundo Molina [b], Nazareno Aguirre [a,c], Marcelo Frias [d], Pablo Ponzio [a,c]

[a] *University of Rio Cuarto, Argentina*
[b] *IMDEA Software, Spain*
[c] *National Council for Scientific and Technical Research (CONICET), Argentina*
[d] *University of Texas at El Paso, USA*

## ARTICLE INFO

*Keywords:*
Software testing
Automated test generation
Bounded exhaustive testing

## ABSTRACT

Bounded exhaustive testing is a very effective technique for bug finding, which proposes to test a given program under all valid bounded inputs, for a bound provided by the developer. Existing bounded exhaustive testing techniques require the developer to provide a precise specification of the valid inputs. Such specifications are rarely present as part of the software under test, and writing them can be costly and challenging.

To address this situation we propose BEAPI, a tool that given a Java class under test, generates a bounded exhaustive set of objects of the class solely employing the methods of the class, without the need for a specification. BEAPI creates sequences of calls to methods from the class' public API, and executes them to generate inputs. BEAPI implements very effective pruning techniques that allow it to generate inputs efficiently.

We experimentally assessed BEAPI in several case studies from the literature, and showed that it performs comparably to the best existing specification-based bounded exhaustive generation tool (Korat), without requiring a specification of the valid inputs.

* Corresponding author.
*E-mail address:* naguirre@dc.exa.unrc.edu.ar (N. Aguirre).

https://doi.org/10.1016/j.scico.2024.103153
Received 2 November 2023; Received in revised form 30 April 2024; Accepted 27 May 2024

Available online 5 June 2024

## Code metadata

| Code metadata description | |
|---|---|
| Current code version | `v1.0.1` |
| Permanent link to code/repository used for this code version | https://github.com/ScienceofComputerProgramming/SCICO-D-23-00353 |
| Permanent link to Reproducible Capsule | https://zenodo.org/records/11088251 |
| Legal Code License | MIT License |
| Code versioning system used | `git` |
| Software code languages, tools, and services used | `Java 1.8, gradle, docker` |
| Compilation requirements, operating environments and dependencies | `docker` |
| If available, link to developer documentation/manual | https://github.com/mpolitano/bounded-exhaustive-api/blob/master/README.md |
| Support email for questions | mpolitano@dc.exa.unrc.edu.ar |

## 1. Introduction

Testing is the one of the most widely adopted techniques to assess functional quality in software. Testing inherently requires one to select a sample of software inputs or software execution scenarios, where the actual software behavior is contrasted with the expected behavior. Different criteria for this selection exist, which are known to have an important impact on the effectiveness of software assessment. One such criterion is the so called *bounded exhaustive testing* [1,2]. Bounded exhaustive testing consists of assessing the behavior of the software under test (SUT) by executing it on *all* valid bounded inputs, for a user-provided bound on input size. The rationale for bounded exhaustive testing is expressed in the *small scope hypothesis* [3], which states that the majority of software defects can be revealed by relatively small inputs. In effect, bounded exhaustive testing has been shown to be very effective for finding software defects [1,4,2,5].

The combinatorial nature of bounded exhaustive testing requires, for its practical application, the use of mechanisms for *automated test generation*. State-of-the-art bounded exhaustive test generation approaches are *specification-based* [1,2,6,7], i.e., they require the developer to provide, besides bounds for the involved data domains, a formal specification of the constraints that the inputs of the SUT must satisfy, in terms of operational [2], declarative [1], or hybrid [6,8] specifications. Such specifications are rarely found in practice as part of the SUT, and writing them is non-trivial, challenging and time consuming. This is due to the fact that specifications must very precisely capture the constraints that the valid inputs must satisfy. Over-constrained specifications lead the bounded exhaustive approaches to miss the generation of inputs that could trigger defective behavior. On the other hand, under-constrained specifications lead to the generation of invalid inputs, which can produce false positives when testing the SUT, and to larger than actually necessary numbers of test inputs, that affect the efficiency of the testing process (see Section 2). Furthermore, some techniques require specifications to be written in a very specific way in order to take advantage of the pruning mechanisms they implement [2]; failing to capture specifications in the right way for the corresponding tool can seriously impact the performance of test generation too.

To address the above described difficulties in the use of bounded exhaustive testing, we present a novel tool for bounded exhaustive input generation, called BEAPI [9]. In contrast with existing approaches, BEAPI solely employs the public methods in the API of a class in order to perform bounded exhaustive input generation for the class, without the need for a formal specification of valid inputs. Besides the SUT, BEAPI only requires the developer to provide bounds for the maximum length for input-generating method sequences, or alternatively a maximum size (number of constituent lower-level objects) for each class in the SUT, and ranges for primitive types, the latter to be used as inputs for the primitive-typed parameters of the API methods. BEAPI works by generating test sequences [10], that is, sequences of invocations to methods of the API. In its most basic form, it generates all test sequences that can be created by combining up to $k$ methods of the API, with $k$ being the maximum sequence length, part of the bounds provided by the developer. The generated method sequences are executed by BEAPI to yield a bounded exhaustive set of objects. However, the scalability of this basic approach is very limited, as it can only generate inputs for very small bounds (see Section 4).

To achieve an efficiency comparable with existing approaches, BEAPI implements various optimizations over the “brute force” generation approach described above [9]. Firstly, it exploits the idea of feedback-directed test generation [10] to discard method sequences whose executions produce exceptions, so that only non-failing method executions are extended with additional method invocations in the construction of new method sequences. Secondly, it stores the set of generated objects associated with the accumulated method sequences, so that new method sequences that do not contribute new objects are identified as redundant, and also discarded (not further extended). This latter optimization is a form of *state matching* in the sense of [11]. Finally, we exploit the fact that, often, only a small subset of the set of public methods in the API of a class can be employed to build all feasible objects of a class. For instance, in collection implementations, it is usually the case that a constructor and an insertion method suffice to build all feasible collection objects. We call such subsets of the API the *builder methods*, or *builders*. BEAPI takes advantage of automatically detected builders [9] (or, alternatively, these may be provided by the developer) to avoid generating a significant number of redundant test sequences, improving test generation performance.

The remainder of this paper is organized as follows. First, Section 2 illustrates the difficulties of writing precise formal specifications, and motivates the advantages of using BEAPI. Then, Section 3 describes the architecture and the main features of BEAPI. Finally, Section 4 summarizes the results of an experimental assessment of BEAPI in comparison with Korat [2], the best performing specification-based bounded exhaustive generation tool, showing a comparable performance [9]. Finally, Section 5 ends with the conclusions of our work and discusses lines of future work.

## 2. Motivating example

In this section, we motivate our proposed technique, and illustrate the difficulties of writing precise formal specifications for bounded exhaustive generation, by means of an example. Let us consider the NodeCachingLinkedList (NCL) implementation from Apache Commons [12]. NCL is a Java implementation of a collection that consists of a main circular, doubly-linked list, and a secondary singly-linked list that acts as a cache of previously used nodes. Thus, nodes removed from the main list are stored in the cache for future usage. When a node is needed for an insertion operation, a node from the cache is reused (if one exists) without the need of allocating a new node. The goal is to avoid the overhead of wasteful object creation and frequent garbage collection in applications that update lists very often through large numbers of insertions and removals.

In order to use Korat for bounded exhaustive generation for NCL, an operational predicate in Java characterizing valid NCL objects must be provided [2]. An example of such a predicate is shown in Fig. 1. The `repOK` predicate in the Figure was taken from the roops benchmark [13]. This `repOK` checks if a NCL structure is a valid structure or not. The first part of `repOK` (lines 2 to 25) checks that the main list is a well-formed circular, doubly-linked list with a sentinel head. The second part (lines 26 to 35) checks that the cache list is a typical null-terminated, acyclic singly-linked list.

There are two main issues in `repOK` that negatively affect bounded exhaustive generation. The first is shown in lines 4 and 5 of Fig. 1. The lines are commented in the Figure, since these are missing in the original `repOK` from roops. These lines check that the value of the sentinel node of the main list (`this.header`) must always be null. The second issue is depicted in the commented statement in line 34 of Fig. 1. This statement, which is also missing in the original `repOK` from roops, checks that the values of the nodes in the cache list must all be set to null. This `repOK` shows a typical case of under-specification, where the developer fails to fully specify the constraints on valid inputs. These issues may seem minor, but they have important consequences: running Korat to perform bounded exhaustive generation for NCL with up to 6 nodes using this weakly specified `repOK` yields 800,667 objects. In contrast, the valid NCL objects are in fact 11,116, less than 2% of the objects generated using `repOK`.

Having so many invalid objects substantially reduces Korat's performance and scalability. More importantly, it degrades the performance of the subsequent testing of the SUT. Furthermore, in some cases invalid objects might cause false positives. That is, failures in testing that arise due to employing invalid objects to test the SUT, but do not represent real bugs in the SUT. False positives are a significant burden for the developer, as he/she needs to manually debug and find the cause of the failure, and discard the tests responsible for the false positives.

Our example shows that it is difficult to write precise specifications. Even in this case, where the `repOK` originates from the roops benchmark intended to be used in the assessment of automated analysis tools, and specifications are written by experts, inaccurate specification issues are found. Furthermore, we have identified many similar errors in specifications from other benchmarks, as reported in [9].

Another aspect to take into account while writing specifications for test generation using Korat (as well as using other bounded exhaustive test generation tools), that makes the task of the specifier even harder, is that the way specifications are written can significantly affect Korat's performance. The `repOK` in Fig. 1 actually follows the guidelines from Korat's authors [2]: it fails (returns false) as soon as a constraint violation in the structure is found. This often allows Korat to prune a very large portion of invalid structures from its search space and run significantly faster [2]. However, fine-tuning the specification to achieve good performance with Korat is hard, as evidenced by the variability in performance presented by Korat when using specifications written by different developers for the same structure [9].

As opposed to specification-based approaches, our tool BEAPI does not require formal specifications of the properties of the inputs to perform bounded exhaustive generation. It directly employs the public methods from the API of the SUT, thus relieving the developers from the effort of having to write the specifications, while at the same time guaranteeing that all generated objects are true positives, i.e., correspond to feasible objects that can be created by using the corresponding class' API. This makes BEAPI significantly easier to use in comparison with existing approaches. Fig. 2 shows some sample tests generated using BEAPI, in the form of sequences of calls to public methods to the class' API, in this case NCL's API.

## 3. Software framework

Fig. 3 shows an overview of the BEAPI tool. BEAPI takes as inputs the target class (or classes) for test case generation, configuration files defining the bounds (also called *scopes* in this context), and a file with the signatures of the methods of the target class that will be used for generation (see the discussion on builder methods below). As outputs, the tool yields a bounded exhaustive set of objects, a JUnit [14] test suite with the method sequences produced by BEAPI to create each object in the result set, and a separate test suite with tests revealing errors (if these have been found) in the methods used for generation.

BEAPI is a command line tool that runs on a Docker container. BEAPI's source code and documentation are available online in a github repository [15]. For instructions on how to install BEAPI refer to the documentation. The tool is implemented on top of Randoop's infrastructure [16], replacing random test sequence generation by bounded exhaustive generation. The script `run-beapi.sh`, located in the project's root directory, executes bounded exhaustive generation with BEAPI. Its syntax is shown below:

```
# ./run-beapi.sh -cp=<classpath> -c=<target class>
              -l=<primitives scope> -b=<objects scope>
              -m=<methods> -s=<objects file>
```

The next sections describe the parameters of the script in detail.

```
public boolean repOK() {
  if (this.header == null) return false;
  // ERROR: Missing lines:
  // if (this.header.value != null)
  //      return false;
  if (this.header.next == null)
      return false;
  if (this.header.previous == null)
      return false;
  if (this.cacheSize > this.maximumCacheSize)
      return false;
  if (this.size < 0) return false;
  int cyclicSize = 0;
  LinkedListNode n = this.header;
  do {
      cyclicSize++;
      if (n.previous == null) return false;
      if (n.previous.next != n) return false;
      if (n.next == null) return false;
      if (n.next.previous != n) return false;
      if (n != null) n = n.next;
  } while (n != this.header && n != null);
  if (n == null) return false;
  if (this.size != cyclicSize − 1)
      return false;
  int acyclicSize = 0;
  LinkedListNode m = this.firstCachedNode;
  Set visited = new HashSet();
  visited.add(this.firstCachedNode);
  while (m != null) {
      acyclicSize++;
      if (m.previous != null) return false;
      // ERROR: Missing line:
      // if (m.value != null) return false;
      m = m.next;
      if (!visited.add(m)) return false;
  }
  if (this.cacheSize != acyclicSize)
      return false;
  return true;
}
```

**Fig. 1.** NCL's specification from ROOPS.

```
@Test
public void test014() {
  // Creates a list with one element (1),
  // and an empty cache
  NodeCachingLinkedList list0 =
      new NodeCachingLinkedList();
  boolean b2 = list0.addLast(1);
}
...
@Test
public void test025() {
  // Creates a list with two elements,
  // (0 and 1) and an empty cache
  NodeCachingLinkedList list0 = new
      NodeCachingLinkedList();
  boolean b2 = list0.addLast(0);
  boolean b4 = list0.addLast(1);
}
...
@Test
public void test261() {
  // Creates a list with one element (1),
  // and a single node in the cache
  NodeCachingLinkedList list0 = new
      NodeCachingLinkedList();
  boolean b2 = list0.addLast(0);
  boolean b4 list0.addLast(1);
  list0.removeFirst();
}
...
```

**Fig. 2.** A few tests for NCL generated with BEAPI.

### 3.1. BEAPI's inputs

*Target class.* The source code of the target class (or classes) has to be compiled first, and the `-cp` option must be set to the location of the Java bytecode (.class files). The `-c` option specifies the fully qualified name of the target class. The source code for the `NCL` example is located in the `examples` folder in BEAPI's repository. Its fully qualified class name is:

```
org.apache.commons.collections4.list.NodeCachingLinkedList
```

To generate tests for several related target classes, the `-c` flag can be used multiple times. In other words, pass `-c` once for each class.

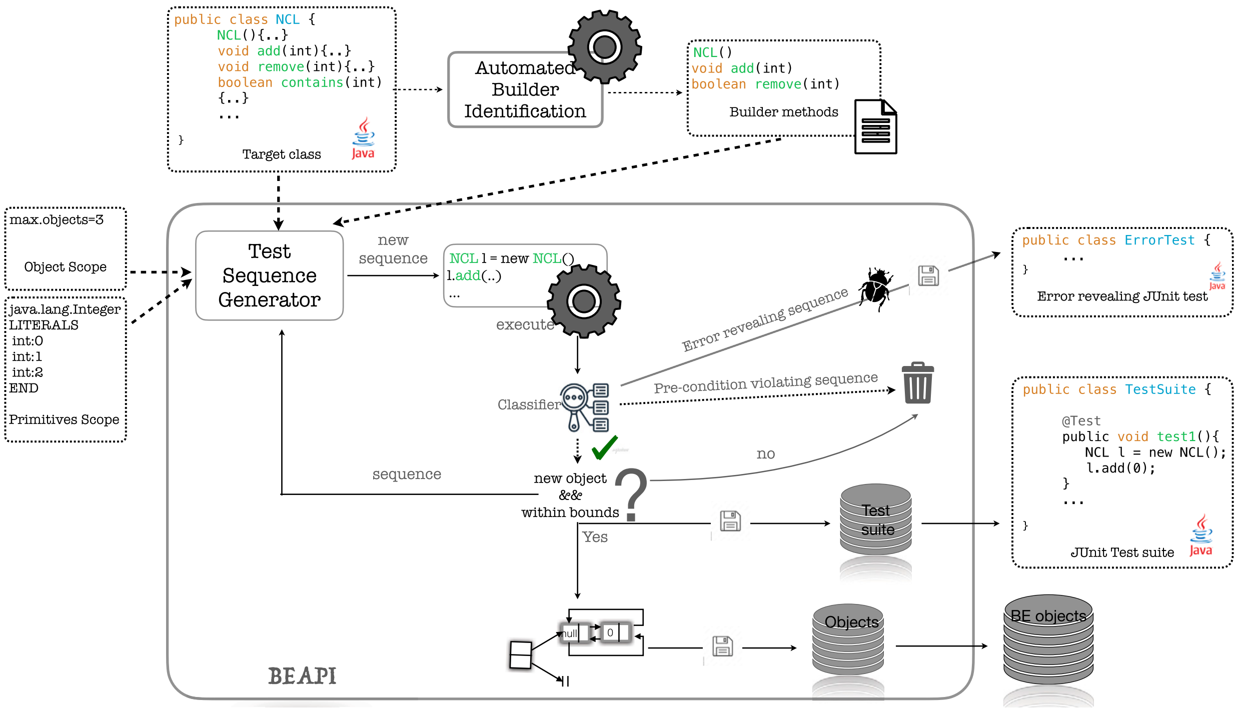
*Scopes.* The scopes are defined via two configuration files, that must be provided to the tool using the `-l` and `-b` options.

Fig. 4 shows the contents of a file that defines the scopes for reference types. The file path must be passed as a parameter using the `-b` option. `max.objects` defines the maximum allowed size for objects, which in this case is set to 3. Thus, method sequences that create lists with more than 3 nodes will be discarded (together with the objects created by the execution of the sequence). The `omit.fields` parameter is a Java regular expression that indicates the fields that must be omitted when canonicalizing objects, which affects the state matching mechanism of BEAPI. We refer the reader to [9] for details. This parameter can be left unset, without any significant impact, in most cases.

Fig. 5 shows a sample file defining the primitive values that BEAPI can use for test sequence generation. The `-l` option should be used to pass the file path. When an API method takes a primitive-typed parameter, BEAPI will invoke the method once with each primitive value defined in the file. One may also specify primitive values for other primitive types like floats, doubles and strings, by describing their values by extension. The format of this file is inherited from Randoop's [16].

*Builder methods.* As the feasible combinations of test sequences grow exponentially with the number of methods, it is crucial to reduce the set of methods that BEAPI uses for generation as much as possible. Thus, BEAPI is parameterized to receive the subset of API methods that it will use for generating tests. The `-m` option expects a file containing the signatures of the builder methods that BEAPI will employ for generation. If `-m` is left unset, BEAPI uses the whole API.

To avoid missing the generation of relevant objects, we must always provide BEAPI with a *sufficient* set of builder methods [17]. That is, methods from the API that by themselves are enough to produce all the feasible objects for the target class. A sufficient set of builders can be computed automatically from the API of a class using an external tool [9]. Precomputed builder methods are available for all case studies in the empirical evaluation of the paper introducing the `BEAPI` approach [9]. For instance, the contents of the file with the signatures of the builders for NCL is shown below:



```
public class NCL {
      NCL(){..}
      void add(int){..}
      void remove(int){..}
      boolean contains(int)
      {..}
      ...
}
```

```
NCL()
void add(int)
boolean remove(int)
```

```
max.objects=3
```

```
java.lang.Integer
LITERALS
 int:0
 int:1
 int:2
END
```

```
NCL l = new NCL()
l.add(..)
...
```

```
public class ErrorTest {
      ...
}
```

```
public class TestSuite {

     @Test
     public void test1(){
        NCL l = new NCL();
        l.add(0);
     }
     ...
}
```

**Fig. 3.** Overview of the BEAPI approach.

```
max.objects=3
omit.fields=DEFAULT_MAXIMUM_CACHE_SIZE
```

**Fig. 4.** BEAPI's scopes for reference types in our `NCL` example.

```
START CLASSLITERALS
CLASSNAME
java.lang.Integer
LITERALS
int:0
int:1
int:2
END CLASSLITERALS
```

**Fig. 5.** Primitive values used by BEAPI in our `NCL` example.

```
ncl.NodeCachingLinkedList.<init>\(\)
ncl.NodeCachingLinkedList.addLast\(int\)
ncl.NodeCachingLinkedList.removeIndex\(int\)
```

### *3.2. An overview of BEAPI's approach*

BEAPI employs the builder methods of the target class and the primitive values provided as scopes to generate sequences of builder methods, that constitute test sequences and lead to test objects. This task is performed by the test sequence generator module in Fig. 3. Briefly, the module first generates all the feasible test sequences of length 1 (with just one method invocation). Recall that the primitive values provided as scopes are used to instantiate parameters of the corresponding types when invoking methods from the API. Then, the sequences are executed and the ones whose execution create “new” objects (different from objects generated by previous sequences) are fed back to the test sequence generator. These sequences are then extended with one extra method invocation, and all the sequences of length 2 (the sequential composition of two method invocations) are generated by the module. The sequences are executed, and the ones yielding new objects are employed by the module to produce the sequences of length 3, and so on. When no new objects within the provided scopes are produced by the sequences of length $k$, the test sequence generator terminates and BEAPI returns the generated test suite and objects.

Let us now focus on the execution of the test sequences, and what BEAPI does with each sequence in response to the different outcomes. The classifier module in Fig. 3 observes the outcome of the execution of the current sequence, which can be one of the following:

1. the sequence reveals an error in a method;
2. the sequence violates a precondition of a method;

**Table 1**
Some results from our experimental assessment of BEAPI against Korat.

| Class | Scope | Time (seconds) | |
|---|---|---|---|
| | | Korat | BEAPI |
| Korat's SLList | 7 | 5.76 | 17.87 |
| | 8 | 8.16 | **256.49** |
| | 9 | **190.45** | TO |
| Korat's BinTree | 10 | 131.18 | 49.10 |
| | 11 | **1137.17** | 199.46 |
| | 12 | TO | **1341.86** |
| Roops's NCL | 6 | 0.65 | 2.27 |
| | 7 | 8.797 | 33.89 |
| | 8 | **205.596** | **769.63** |
| Korat's Red-Black Tree | 11 | 40.54 | 33.42 |
| | 12 | 220.77 | 79.45 |
| | 13 | **1277.67** | **689.06** |
| Kiasan's Red-Black Tree | 7 | 10.76 | 0.78 |
| | 8 | **283.33** | 1.57 |
| | 12 | TO | **84.51** |

3. the sequence creates an object larger than allowed by the scopes or the objects it creates were already generated by a previous test sequence; or
4. the sequence generates at least one new object that is within the scopes.

Cases (3) and (4) correspond to test sequences that finish their execution successfully (without raising exceptions). The objects stored by variables at the end of the execution of the sequence are canonicalized [9], and the tool checks if they were already generated or they are larger than allowed by the scopes (case 3). In such cases, the test sequence and the objects are discarded. Otherwise, if the sequence produces at least one new object that has not been seen before (case 4), the sequence is returned back to the test sequence generator, it is stored as a test in the resulting test suite, and the objects it creates are stored to be returned in the resulting bounded exhaustive set of objects.

Cases (1) and (2) correspond to test sequences that do not terminate correctly and throw exceptions. As proposed by Randoop [16], the kind of the exception yielded by the execution separates what BEAPI considers a bug in a method from a precondition violation. By default, `IllegalArgumentException` and `IllegalStateException` correspond to precondition violations in BEAPI, while the remaining exceptions are considered error revealing exceptions. However, the user can configure how exceptions are treated by BEAPI via parameters of the tool. Of course, test sequences that violate preconditions are discarded by BEAPI (case 1), and those that reveal bugs are reported and returned to the user (case 2). Similarly to Randoop, the user can provide additional specifications such as a `repOK`, to enhance BEAPI's bug finding capabilities.

BEAPI can be configured to enable or disable the state matching optimization, as well as the use of builders [18]. When the optimizations are disabled, BEAPI can be very inefficient, and may only generate inputs for very small scopes (about 3) within reasonable time [9]. On the other hand, with optimizations enabled, BEAPI's performance is comparable to the performance of Korat (see Section 4).

### *3.3. BEAPI's outputs*

The result of executing the script is a JUnit test suite. Fig. 2 shows an excerpt of the JUnit test suite generated by BEAPI for the NCL class, using the configuration files shown in this section. Each test in the suite is a sequence of methods that creates an object in the resulting bounded exhaustive set of objects. The user can instruct BEAPI to serialize the objects to a file via the `-s` option.

## 4. Experimental evaluation

In this section, we summarize some of the results from the experimental evaluation performed in [9], where we assessed the performance of BEAPI for bounded exhaustive generation of structurally complex inputs. In the evaluation, we compared BEAPI against the fastest bounded exhaustive generator to date, Korat [19]. As mentioned before, Korat is a specification-based approach, requires `repOK`s to be provided by developers and suffers from the limitations discussed in Section 2.

Table 1 shows part of the results from the experimental assessment. The Table reports generation times (in seconds) for both tools and a few scopes. The conclusion from the whole evaluation [9], which can be observed in the results presented in Table 1 is that the performance of BEAPI is comparable to that of Korat. In some cases, Korat is faster and/or scales better (`Korat's SLList` and `Roops' NCL`), while in other cases it is the other way around (`Korat's BinTree` and `Korat's Red-Black Tree`), and BEAPI is faster and/or scales better. The last case we included here illustrates how Korat's performance strongly depends on how the specification of the valid inputs is structured. For the same `Red-Black Tree` structure, the generation performed with the `repOK` provided with Korat (`Korat's Red-Black Tree`) is much faster and scales to larger scopes, compared to the generation performed with a `repOK` written by Kiasan's developers, used in the assessment of their own tool (`Kiasan's Red-Black Tree`) [20]. This further emphasizes our point that writing specifications for bounded exhaustive generation is non-trivial.

Finally, we would like to highlight the crucial role of BEAPI's optimizations. For example, for NCL, BEAPI with both optimizations enabled reaches a scope of 8. With optimizations disabled, it can only terminate successfully for scopes up to 3 within 60 minutes [9].

## 5. Conclusions and future work

We presented BEAPI, a tool for bounded exhaustive test input generation that solely relies on the public API of the SUT for test generation. By directly exercising the API of the software under test, BEAPI becomes straightforward to use, and enables bounded exhaustive generation for software lacking formal specifications of valid inputs. At the same time, the tool necessarily generates true positive inputs, in the sense that every produced input is indeed constructible from the SUT's public methods.

Despite the fact that bounded exhaustive test generation in intrinsically combinatorial, the generation process can be made efficient thanks to various optimizations to BEAPI's bounded exhaustive generation algorithm, that we have proposed and implemented. These optimizations include feedback-driven method sequence generation, redundant state elimination based on state matching, and the identification and subsequent use of builder methods, to reduce method sequence explosion. We experimentally showed that the proposed optimizations are crucial for the efficiency and scalability of the tool, and that they allow BEAPI to achieve an efficiency that is comparable to that of the most efficient existing specification-based bounded exhaustive approaches.

As future work, we plan to evaluate BEAPI against other non-exhaustive state-of-the-art test generators (e.g., Randoop), using modern benchmarks from the literature, to better assess the capabilities of BEAPI (and bounded exhaustive generation in general) against other automated testing techniques.

## CRediT authorship contribution statement

**Mariano Politano:** Writing – review & editing, Writing – original draft, Validation, Software, Investigation. **Valeria Bengolea:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Investigation. **Facundo Molina:** Writing – review & editing, Writing – original draft, Validation, Investigation. **Nazareno Aguirre:** Writing – review & editing, Writing – original draft, Supervision, Investigation. **Marcelo Frias:** Writing – review & editing, Writing – original draft, Supervision, Investigation. **Pablo Ponzio:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Investigation.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Nazareno Aguirre reports financial support was provided by National University of Río Cuarto, through grant C588-1. Nazareno Aguirre reports a relationship with National Scientific and Technical Research Council that includes: employment. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

[1] D. Marinov, S. Khurshid, 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, IEEE Computer Society, 2001, p. 22.

[2] C. Boyapati, S. Khurshid, D. Marinov, Korat: automated testing based on Java predicates, in: P.G. Frankl (Ed.), Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002, ACM, 2002, pp. 123–133.

[3] A. Andoni, D. Daniliuc, S. Khurshid, D. Marinov, Evaluating the "small scope hypothesis", Tech. Rep., 10 2002.

[4] S. Khurshid, D. Marinov, Checking Java implementation of a naming architecture using testera, Electron. Notes Theor. Comput. Sci. 55 (3) (2001) 322–342, https://doi.org/10.1016/S1571-0661(04)00260-9.

[5] K.J. Sullivan, J. Yang, D. Coppit, S. Khurshid, D. Jackson, Software assurance by bounded exhaustive testing, in: G.S. Avrunin, G. Rothermel (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004, ACM, 2004, pp. 133–142.

[6] N. Rosner, V.S. Bengolea, P. Ponzio, S.A. Khalek, N. Aguirre, M.F. Frias, S. Khurshid, Bounded exhaustive test input generation from hybrid invariants, in: A.P. Black, T.D. Millstein (Eds.), Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, Part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, ACM, 2014, pp. 655–674.

[7] N.A. Awar, K. Jain, C.J. Rossbach, M. Gligoric, Programming and execution models for parallel bounded exhaustive testing, Proc. ACM Program. Lang. 5 (OOPSLA) (2021) 1–28, https://doi.org/10.1145/3485543.

[8] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, D. Marinov, Test generation through programming in UDITA, in: J. Kramer, J. Bishop, P.T. Devanbu, S. Uchitel (Eds.), Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010, ACM, 2010, pp. 225–234.

[9] M. Politano, V.S. Bengolea, F. Molina, N. Aguirre, M.F. Frias, P. Ponzio, Efficient bounded exhaustive input generation from program APIs, in: L. Lambers, S. Uchitel (Eds.), Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings, Springer, 2023, pp. 111–132.

[10] C. Pacheco, S.K. Lahiri, T. Ball, Finding errors in net with feedback-directed random testing, in: B.G. Ryder, A. Zeller (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, ACM, 2008, pp. 87–96.

[11] W. Visser, C.S. Pasareanu, R. Pelánek, Test input generation for Java containers using state matching, in: L.L. Pollock, M. Pezzè (Eds.), Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006, ACM, 2006, pp. 37–48.

[12] Java Apache, Commons, https://commons.apache.org/.

[13] ROOPS Benchmark, https://code.google.com/archive/p/roops/.

[14] Junit, https://junit.org/.

[15] BEAPI tool, https://github.com/mpolitano/bounded-exhaustive-api/.

[16] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, IEEE Computer Society, 2007, pp. 75–84.

[17] P. Ponzio, V.S. Bengolea, M. Politano, N. Aguirre, M.F. Frias, Automatically identifying sufficient object builders from module apis, in: R. Hähnle, W.M.P. van der Aalst (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, in: Lecture Notes in Computer Science, vol. 11424, Springer, 2019, pp. 427–444.

[18] BEAPI Reproducible, Empirical evaluation, https://github.com/mpolitano/bounded-exhaustive-api-testgen/.

[19] J.H. Siddiqui, S. Khurshid, An empirical study of structural constraint solving techniques, in: K.K. Breitman, A. Cavalcanti (Eds.), 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings, in: Lecture Notes in Computer Science, vol. 5885, Springer, 2009, pp. 88–106.

[20] X. Deng, Robby, J. Hatcliff, 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, IEEE Computer Society, 2006, p. 137.