Korat-API: A Framework to Enhance Korat to Better Support Testing and Reliability Techniques

Nima Dini, Cagdas Yelen, Zakaria Alrmaih, Amresh Kulkarni, and Sarfraz Khurshid
The University of Texas at Austin (USA)
{nima.dini,cagdas,zakotm}@utexas.edu, amresh.kulkarni@gmail.com, khurshid@utexas.edu

ABSTRACT

Logical constraints play an important role in software testing and reliability. For example, constraints written by users allow automating test case generation and systematic bug finding, and constraints computed using data-flow of a program allow studying its reliability. The key to practical usefulness of constraints is the effectiveness and efficiency of constraint solvers that determine constraint feasibility and produce solutions.

Our focus in this paper is the well-studied solver Korat for constraints written as imperative predicates. We introduce Korat-API, a novel framework that introduces an API for building constraint solving problems, and algorithms for solving them to enhance Korat. Our goal is two-fold: (1) to facilitate the use of Korat as an optimized constraint solver when used as a standalone tool or as a backend engine; (2) to optimize Korat for more efficient constraint solving by exploiting problem-specific information. We describe the API and algorithms that embody our framework. We also present an experimental evaluation using a suite of complex constraint solving problems to show the potential usefulness of Korat-API.

KEYWORDS

Constraint solving, Imperative constraints, Korat

ACM Reference Format:
Nima Dini, Cagdas Yelen, Zakaria Alrmaih, Amresh Kulkarni, and Sarfraz Khurshid. 2018. Korat-API: A Framework to Enhance Korat to Better Support Testing and Reliability Techniques. In *Proceedings of SAC 2018: Symposium on Applied Computing , Pau, France, April 9–13, 2018 (SAC 2018),* 10 pages. https://doi.org/10.1145/3167132.3167339

1 INTRODUCTION

Constraint solvers [6, 9, 14] now play an important role in the development of reliable systems. For example, efficient constraint solving lies at the heart of various modern approaches to systematic testing [6, 33], symbolic execution [23, 26, 27], automated theorem proving [10], and software reliability techniques [17, 18].

Our focus is the Korat framework [6], which introduced the first approach for bounded exhaustive testing using constraints written as imperative predicates, termed *repOK*() methods [31]. The Korat user writes properties of desired inputs as the *repOK*() method, which checks those properties and returns true if and only if its input satisfies them. The user also writes a *finitization* that bounds the input size. Korat implements a backtracking search with efficient pruning and isomorphism breaking to systematically explore the space of all possible inputs up to the bound, and enumerates all inputs that satisfy the desired properties; each generated input serves as a desired input for testing the program under test.

The Korat approach has been effective in various applications for systematic bug finding [32]; most recently, it was used as a backend tool for model counting, i.e., computing the number of solutions, in the context of a technique for software reliability [17, 18]. Moreover, several projects have built on the basic Korat approach to further optimize it, e.g., using parallel techniques [7, 11, 34, 38], incremental techniques [11], memoization [12], and static analysis [40].

A key property of Korat that makes it particularly attractive for automated testing is that it solves predicates written in imperative code and thus does not require the use of some specialized language for writing constraints, which has semantics and syntax possibly very different from the widely used imperative programming languages that the developers are commonly familiar with. While this key property is a particular strength of Korat, it also leads to a basic limitation: the constraints that are *logical* in nature must be written in a language that is *imperative*, which can create a potential mismatch between "what was intended" and "how it was written". The logical nature of the constraints can largely be lost due to the required use of standard imperative constructs, e.g., sequencing of statements, and related execution semantics, e.g., short-circuiting of conditional expressions. For example, assume the user wants the inputs to satisfy properties p and q that can each be checked *independently*, i.e., regardless of whether the other property holds, and writes the *repOK*() predicate body simply as "*return p*() *and q*();" where predicate $p()$ checks property p and predicate $q()$ checks property q. Purely as a consequence of the way the user has written the *repOK*(), an artificial dependence, specifically q is checked only if p is true, i.e., $p()$ has become a pre-condition of $q()$, between the two properties has been introduced.

Preserving the logical structure of the constraints is valuable for two key reasons. One, it facilitates re-use of constraints written as imperative predicates and understanding them. Two, perhaps more importantly, it allows the backend search to be improved using optimization techniques which are able to explore different orders of checking properties [30]. The execution-driven nature of Korat's backtracking search makes it inherently dependent on the way the *repOK*() is written. The same properties written differently can lead to significantly different performance of Korat. Thus, for optimal performance, the user has to pay particular attention to how to write *repOK*().

Our key insight is that we can enhance the applicability and usefulness of Korat by defining an API that allows building constraint solving problems for Korat and enables directly conveying (1) high level structure of the desired input properties being written as *repOK*() predicates; and (2) specific search goals, e.g., based on fine-grained test requirements. We design our API specifically in view of some projects on parallel or incremental techniques that enhance Korat [11, 34, 38] or use it as a backend solver [17, 18].

This paper introduces our API and supporting algorithms, which we term collectively as the *Korat-API* framework. We describe how Korat-API uses the core Korat engine and demonstrate its benefits in two application contexts – as a standalone test generator and as a backend constraint solver. In the context of using Korat as a standalone test generator, we focus on exploiting *higher-level* logical structure of the constraints to optimize Korat. Specifically, we demonstrate that Korat-API allows describing logical conjunctions of properties and *re-ordering* constraints for faster solving. In the context of using Korat as a backend constraint solver, we focus on exploiting *lower-level* constraints, i.e., fine-grained requirements on field-values, to optimize Korat; we derive motivation from a recent approach [17, 18] for reliability that leveraged the standard Korat tool-set and invoked it directly for model counting based on path conditions created by symbolic execution [27]. We demonstrate that Korat-API can help in constructing such constraint solving problems for model counting and solving them more efficiently.

This paper makes the following contributions [1, 30]:

- **Korat API**. We introduce the idea of providing an API for constructing constraint solving problems for the Korat solver for imperative predicates;
- **Translation to imperative solving problems**. We describe the core algorithms that support our API for constructing the constraint solving problems based on imperative constraints for standard Korat. We also present algorithms for more efficient solving by exploiting higher-level constraint structure using constraint re-ordering, as well as lower-level requirements on field-values using more effective pruning.
- **Evaluation**. We present an experimental evaluation using our prototype Korat-API. The experimental results show that using our API can provide substantial speedup for test generation and model counting by exploiting the higher-level structure and lower-level requirements.

We believe our work introduces a promising approach for making the ability of Korat to efficiently solve imperative predicates more widely applicable, possibly even in new application contexts where Korat has not been used before. Our work takes inspiration from the Kodkod backend [41] for Alloy [24], a first-order declarative language with transitive closure, which is particularly suitable for creating software models. Kodkod provides a Java API that allows other applications to create constraint solving problems in Alloy and solved using off-the-shelf propositional satisfiability (SAT) solvers. Thus, the Kodkod API provides a programmatic interface for creating constraints in first-order logic and solving them using SAT. Kodkod has enabled the Alloy tool-set and SAT solvers to be utilized in various applications and tools [28]. We hope our work opens the possibility of utilizing Korat's search capabilities likewise.

2 TWO ILLUSTRATIVE EXAMPLES

2.1 Higher-level constraint structure

Consider using Korat to generate binary search trees with parent pointers as test inputs. Figure 1 shows the Java code that defines the constraint solving problem for generating binary search trees with parent pointers using standard Korat [6]. The classes *BST* and *Node* introduce the types necessary to model the tree structure. Each tree has a *root* node and caches the number of nodes in the *size* field. Each node has a *left* child, a *right* child, a *parent*, and an integer element (*elem*). The predicate *repOK*() uses three helper methods (omitted here) to check the desired properties. The method *acyclic*() checks if the input has a valid tree structure, i.e., has no (undirected) cycles, and if the total number of nodes reachable from *root* is equal to the *size* field. The method *parentOK* checks that all parent pointers have the correct values with respect to the *left* and *right* field values. The method *searchOK* checks if the elements in the tree nodes are in the correct binary search order. The finitization method specifies that *elem* can take integer values from range [0, *num* − 1], while *size* is fixed to *num* (to generate structures with exactly *num* nodes). Further, *root*, *left*, *right*, and *parent* fields are either *null* or point to one of *n* unique nodes that Korat search will create upon initialization.

To run Korat, the user compiles the Java code that describes the constraint solving problem (Figure 1) and invokes Korat using the command line. To illustrate, "java -cp .:korat.jar korat.Korat --class BST --args 3" instructs Korat to enumerate all trees with 3 nodes where each node element is 1, 2, or 3. Figure 2 graphically illustrates all trees that Korat generates for this example invocation.

Figure 3 shows how Korat-API can be used to define a constraint solving problem that is analogous to the problem represented by Figure 1 and its corresponding Korat invocation above. Line 2 initializes the constraint solving problem. Lines 5–6 declare the basic types. Lines 9–14 declare the object fields. Lines 17–31 define the *repOK*() predicate and its helper methods. Note the invocation of *land*() that creates a *logical* conjunction of two formulas that can be evaluated in any order, and explicitly provides a *higher-level* logical structure to be exploited during constraint solving. Lines 34–36 define the overall value domains. Lines 39–44 define for each field, its value domains. Lines 47–50 invoke the default Korat solver to compute the solutions, and print them.

2.2 Lower-level requirements on field values

Consider using Korat as a backend model counter to compute the number of solutions for a constraint derived from path conditions that arise in symbolic execution, e.g., in the spirit of recent work on software reliability [17, 18]. To illustrate, consider the following path condition, which is termed *heap-PC* to emphasize the nature of the constraints that are on the fields of heap-allocated objects:

```
header != null && header.next != null &&
  header.next.next == null && size == 2
```

where *header* and *size* are fields in class *List* and *next* is in *Node*.

Figure 4 shows example Java code that the user can write to create the constraint solving problem that represents solving this path condition in conjunction with the list class invariant. The body

```java
public class BST {
  Node root;
  int size;

  public static class Node {
    Node left, right, parent;
    int elem;
  }

  public boolean repOK() {
    return acyclic() && parentOK() && searchOK();
  }

  public static IFinitization finBST(int num) {
    IFinitization f =
      FinitizationFactory.create(BST.class);
    IObjSet nodes=f.createObjSet(Node.class,true);
    nodes.addClassDomain(
        f.createClassDomain(Node.class, num));
    IIntSet elems = f.createIntSet(0, num-1);
    IIntSet size = f.createIntSet(num, num);
    f.set("size", size);
    f.set("root", nodes);
    f.set("Node.right", nodes);
    f.set("Node.elem", elems);
    f.set("Node.left", nodes);
    f.set("Node.parent", nodes);

    return f;
  } ...
}
```

Figure 1: Constraint solving problem for generating binary search trees with parent pointers using standard Korat [6]

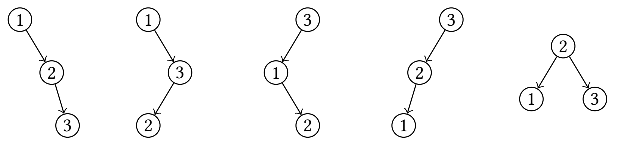

Figure 2: All trees with 3 nodes with integer elements 1, 2, and 3 generated by Korat

of the *repOK*() method returns true if and only if the path condition holds (in addition to the class invariant).

The following Korat invocation solves this problem and reports the count of the solutions: "java -cp .:korat.jar korat.Korat --class List --args 3". Executing this command explores 19 candidates and finds 1 valid instance.

Figure 5 shows how Korat-API can be used to define a constraint solving problem that is analogous to the problem represented by Figure 4 and its corresponding Korat invocation above. Line 2 initializes the constraint solving problem. Lines 3-5 (omitted) declare the basic types, object fields, and helper predicates. Lines 7–10 define the class invariant as the *repOK*() predicate. Lines 13–14 define the overall value domains. Lines 17–19 define for each field, its value domains. Lines 22-25 invoke Korat-API methods that allow specifying *lower-level* requirements on values of object fields. The *excludeNull*(*p*) invocation declares that the object field that is identified by input position *p* in the *candidate vector* that Korat

```java
static void BSTExample() {
  KoratAPI ka = new KoratAPI();

  // Declare types.
  Type bst = ka.Type("BST");
  Type node = ka.Type("Node");

  // Declare fields.
  Field root = ka.Field("root", bst, node);
  Field size = ka.Field("size", bst, ka.INT);
  Field left = ka.Field("left", node, node);
  Field right = ka.Field("right", node, node);
  Field parent = ka.Field("parent", node, node);
  Field elem = ka.Field("elem", node, ka.INT);

  // Define helper predicates.
  Predicate acyclic = ka.Predicate("acyclic()",
    bst, "boolean acyclic() { ... }");
  Predicate parentOK = ka.Predicate("parentOK()",
    bst, "boolean parentOK() { ... }");
  Predicate searchOK = ka.Predicate("searchOK()",
    bst, "boolean searchOK() { ... }");

  // Define repOK.
  Formula f1 = ka.PredicateInvocation(acyclic);
  Formula f2 = ka.PredicateInvocation(parentOK);
  Formula f3 = ka.PredicateInvocation(searchOK);
  Formula f4 = f2.land(f3); // Logical and.
  Formula f = f1.and(f4); // Short-circuiting and.

  Predicate repOK = ka.Predicate("repOK", bst, f);

  // Define domains of values.
  Domain nodes = ka.ObjectDomain(node, 3);
  Domain elems = ka.IntDomain(1, 3);
  Domain sizes = ka.IntDomain(3, 3);

  // Set field value domains.
  ka.setDomain(root, nodes, true);
  ka.setDomain(size, sizes);
  ka.setDomain(left, nodes, true);
  ka.setDomain(right, nodes, true);
  ka.setDomain(parent, nodes, true);
  ka.setDomain(elem, elems);

  // Solve the constraint and print the solutions.
  Iterator<Solution> it =
    ka.solve(repOK, ka.DEFAULT_SOLVER);
  while (it.hasNext()) {
    System.out.println(it.next());
  }
}
```

Figure 3: Constraint solving problem for generating binary search trees with parent pointers using Korat-API

search internally maintains takes non-null values. The $fix(p, d)$ invocation declares that the object field that is identified by input position *p* in the candidate vector takes the fixed value that is identified by input index *d* in the domain of values for that field. These invocations allow our algorithms to more effectively prune the exploration space and provide faster constraint solving. Lines 28–29 use Korat-API for computing the solution count, which returns 2 explored candidates and 1 valid instance, i.e., 17 fewer explored candidates than the traditional approach (described earlier).

```
public class List {
  Entry header;
  int size;

  static class Entry {
    Entry next;
  }

  boolean repOK() {
    // Check class invariant.
    if (!acyclic() || !sizeOK()) return false;

    // Check heap-PC constraint.
    return header != null && header.next != null
      && header.next.next == null && size == 2;
  }

  static IFinitization finList(int num) {
    IFinitization f =
        FinitizationFactory.create(List.class);
    IObjSet e0=f.createObjSet(Entry.class, true);
    e0.addClassDomain(f.createClassDomain(
                                Entry.class, num));
    IIntSet e1 = f.createIntSet(0, num);
    f.set("header", e0);
    f.set("Entry.next", e0);
    f.set("size", e1);
    return f;
  } ...
}
```

Figure 4: Constraint solving problem to count number of solutions for `header != null && header.next != null && header.next.next == null && size == 2`, using Korat [6]

```
static void HeapPCExample() {
  KoratAPI ka = new KoratAPI();
  // Declare types ...
  // Declare fields ...
  // Define helper predicates ...
  // Define class invariant.
  Formula f1 = ka.PredicateInvocation(acyclicOK);
  Formula f2 = ka.PredicateInvocation(sizeOK);
  Formula f = f1.and(f2);
  Predicate inv = ka.Predicate("repOK", list, f);

  // Define domains of values.
  Domain entries = ka.ObjectDomain(entry, 3);
  Domain ints = ka.IntDomain(0, 3);

  // Set field value domains.
  ka.setDomain(header, entries, true);
  ka.setDomain(next, entries, true);
  ka.setDomain(size, ints);

  // Lower-level requirements on object fields.
  ka.excludeNull(0); // header != null
  ka.excludeNull(2); // header.next != null
  ka.fix(3, 0); // header.next.next == null
  ka.fix(1, 2); // size == 2

  // Solve the constraint and count the solutions.
  System.out.println(
    ka.count(inv, ka.DEFAULT_SOLVER));
}
```

Figure 5: Constraint solving problem to count number of solutions for `header != null && header.next != null && header.next.next == null && size == 2`, using Korat-API

3 KORAT-API

This section describes our Korat-API framework, which has two key elements: (1) a programmatic API for defining constraint solving problems; and (2) two forms of algorithms – (a) reducing problems defined using our API to constraint solving problems for *enhanced* Korat (Section 3.1), and (b) enhancing the standard Korat search to provide more efficient solving. Thus, the usage of Korat-API has two key steps: (1) defining the constraint solving problem using the API; and (2) solving it using our framework. The first step may be performed by a human user or an application; the second step is mechanical and embodied by our algorithms that are implemented in our prototype tool.

Korat-API enables two forms of optimizations for solving imperative constraints. To exploit higher-level logical structure of the constraints, specifically logical conjunction, Korat-API explores different constraint *orders* for smaller sizes, to identify a sequential order that is likely optimal to solve the constraints for the target size (Section 3.2). To exploit lower-level requirements on field values, Korat-API modifies the Korat search to make its backtracking *aware* of these requirements and enhance pruning (Section 3.3).

To build a constraint solving problem, there are four key parts to define: (1) types and fields; (2) the constraints; (3) the bounds on the state space; and (4) the analysis to perform (e.g., solve, count, etc.). Each constraint solving problem is defined in a new Java class that contains a *main* method, which typically starts by initializing a new object of class *KoratAPI*, which provides the core methods for declaring the four parts of the problem.

3.1 Korat-API program translation

Algorithm 1 describes the translation of a Korat-API program to an enhanced Korat problem. Line 1 declares the target class meta-object. Line 2 initializes the set of declared types to *main*. Lines 3-4 create the repOK and finitization methods, based on user-defined formulas and bounds, and add them (in textual representation) to the main class. Lines 6-11 iterate over the list of user-defined fields, add them as static inner classes within appropriate types, and mark the corresponding types as declared. Next, Lines 13-15 add the user-defined predicates to the main class meta-object. Finally, Line 16 feeds the textual representation of the auto-generated main class to an in-memory Java compiler, and returns the compiled Class.

3.2 `repOK` constraint prioritization

Korat search is inherently sequential [34], as it uses repOK execution to guide the search. Therefore, two semantically equivalent repOK implementations may lead to very different performances. While Korat experts, who are familiar with Korat nuances, can write efficient repOK methods, a regular Java developer may suffer from a poorly written repOK logic. This section describes how Korat-API can automatically prioritize logical constraints of a repOK predicate.

Recall from Section 2.1 that a logical conjunction of two formulas, specified by a *land* invocation in Korat-API, can be evaluated in any order. Algorithm 2 explains how Korat-API automates the process of logical constraint prioritization. It takes as inputs a `formula` and a `size` of *field domains* (specified by the user through a sequence of Korat-API method invocations). Lines 1-4 gets all the predicate

Algorithm 1: Translate a program written in Korat-API to an enhanced Korat problem

```
Input: Loaded Korat API object API.
Output: A compiled Java class including repOK and finitization.
main ← API.mainJavaClass()
declaredSet ← {main}
main.addMethod(API.getRepOK())
main.addMethod(API.getFinitization())
for field in API.fields() do
    from ← getJavaClass(field.getFrom())
    to ← getJavaClass(field.getTo())
    if $to \notin declaredSet \wedge from \neq to$ then
        from.addStaticInnerClass(to)
        declaredSet ← {to} ∪ declaredSet
    end
end
for predicate in API.predicates() do
    main.addMethod(predicate)
end
return CompileInMemory.compile(main.stringify())
```

Algorithm 2: Prioritize logical `repOK` constraints to find the optimal sequential order of constraints for the given size

```
Input: Korat-API formula and size.
Output: Permutation with the smallest space explored.
1  permutations ← genPermutations(formula)
2  bestPermutation ← ∅
3  minNumberExplored ← ∞
4  solutionCount ← −1
5  for perm ∈ permutations do
6      newFormula ← perm.removeFirstPredicate()
7      for predicate ∈ perm do
8          newFormula ← newFormula.and(predicate)
9      end
10     solution ← KoratAPI.solve(newFormula, size)
11     count ← solution.numberOfValidSolutions()
12     if solutionCount == −1 then
13         solutionCount ← count
14     else
15         assert solutionCount == count
16     end
17     explored ← solution.numberOfCandidatesExplored()
18     if explored < minNumberExplored then
19         explored ← minNumberExplored
20         bestPermutation ← perm
21     end
22     return bestPermutation
23 end
```

permutations and initializes the local variables. Lines 6-9 build a formula, that consist of the conjunction of all predicates in a permutation. Line 10 makes a backend call to Korat that returns a solution object including the total number of valid and explored candidates. Line 15 checks whether Korat finds the same number of valid solutions for each permutation. This may be violated if a conjunction was mistakenly specified as logical by the user. Lines 17- 21 find the permutation with the smallest number of explored candidates, which will be returned in Line 22.

The key advantage of this technique is how one can prioritize constraints on a smaller size of a problem, and apply the resulting order to generate tests for a larger size of the problem. Although this is a heuristic and may not hold, our experiments in Section 4.2 show that in practice this heuristic helps for well-studied data structure invariants. To illustrate a case where the heuristic fails, consider a data structure where the concrete representation depends on the structure size, e.g., a linked structure for small size but an array-based structure for larger sizes; indeed, in such a case trying to find an optimal order using small size may not help at all with generating a larger input.

3.3 Finer-grained finitization

Recall that Korat search is bounded exhaustive, and it explores all non-isomorphic candidates [6]. A property of traditional Korat is that the finitization bound is coarse-grained, i.e., it is at the field declaration level. Specifically, once a field domain is defined, say $[null, N0, N1, N2]$ for *Node.left*, it is fixed and the same domain is used for all left fields for all Node objects. While this behavior is desirable for test generation in general, using Korat as a backend constraint solver (e.g., model-counting), requires supporting finer-grained level of control over finitization. Korat-API facilitates that by introducing two new API calls, namely *excludeNull* and *fix*.

The *excludeNull(p)* invocation makes Korat aware that the object corresponding to index *p* of a candidate vector, cannot take *null* (or 0 at the level of candidate vector) as a value. Hence, our *enhanced* Korat assigns the initial value of that field to the first non-null value from its field domain, i.e., sets the corresponding candidate vector element to 1. Further, it will ensure whenever Korat search attempts to reset that field, due to backtracking, it will reset it to value 1 instead of 0.

The *fix(p, d)* invocation assigns the object field identified by the input position *p* in the candidate vector to the fixed value identified by the input index d from its field domain. Further, it excludes index *p* of candidate vector from Korat search, i.e., the search will neither backtrack, nor consider any different values for that field, resulting in a more efficient search.

These two API extensions enforce tighter bounds on individual field instances, resulting in a targeted Korat search for model-counting purposes. This finer-grained level of control allows efficient solving of constraints like $acyclic(input)\&\&input.next{=}null$ $\&\&input{\neq}null$, which were extensively studied in prior work [18], using the default Korat. Note that the *traditional* way of using Korat in this context, defines the new repOK as a conjunction of the old repOK with a *heap-PC*: $newRepOK() \leftarrow repOK()\ \&\&\ heap\text{-}PC$. However, using Korat search with *finer-grained* finitization, the repOK stays the same, while the *heap-PC* is defined as a set of *excludeNull(p)* and *fix(p, d)* invocations in Korat-API.

Our Korat-API technique supports three categories of clauses for a given *heap-PC*:

(1) $Reference \neq null$, e.g., $N1.left \neq null$.
(2) $Reference == null$, e.g., $header.next.next == null$.
(3) $Primitive == IntegerLiteral$, e.g., $N2.key == 5$.

Table 1: Objects of Study

Subject	`repOK`
disjoint-set (*Disj-Set*)	*size()* && *prop1()* ∧ *prop2()* ∧ *unique()*
doubly-linked-list (*DLL*)	*circular()* && *sorted()* ∧ *link()* ∧ *size()*
height-balanced-bst (*HBST*)	*acyclic()* && *bst()* ∧ *parents()* ∧ *balanced()*
n-queens (*Queens*)	*row()* ∧ *diagonal()*
red-black-tree (*RBT*)	*acyclic()* && *parents()* ∧ *bst()* ∧ *colors()*
singly-linked-list (*SLL*)	*acyclic()* && *unique()* ∧ *sorted()* ∧ *size()*

Table 2: Imperative Properties Used in `repOK`

Property	Description
acyclic()	No directed cycles exists in the data structure.
balanced()	Both `left` and `right` sub-trees are balanced and their height difference is ≤ 1.
bst()	Binary search property holds.
circular()	Data structure is circular.
colors()	Proper assignment of nodes' `colors` (RED or BLACK).
diagonal()	No two queens share the same diagonal.
link()	Current element is the `previous` of the `next` element.
parents()	Proper assignment of `parent` pointers.
prop1()	Each node's `parent` has a non-negative value, smaller than the `size` field.
prop2()	Rank of the `parent` is greater than `child`'s rank, and number of roots is not greater than number of nodes with $rank == 0$.
row()	No two queens share the same row.
size()	Number of nodes is cached in the `size` field.
sorted()	Nodes are maintained in sorted order.
unique()	Data structure does not contain duplicate elements.

4 EVALUATION

To evaluate Korat-API, we answer the following research questions:
RQ1: What is the speedup in *test generation* time and reduction in the number of explored candidates due to constraint prioritization?
RQ2: What is the speedup in test generation when prioritizing constraints on a smaller size?
RQ3: What is the speedup in *model counting* time and reduction in the number of explored candidates due to finer-grained finitization?

Execution platform: We obtained all data on a machine with 4-core 2.20GHz Intel Xeon CPU with 16GB of RAM, running Ubuntu 16.04 LTS. We used Oracle Java 1.8.0_121.

We first discuss the subjects used in our study and then answer our research questions.

4.1 Subjects

We used a variety of data structures, plus a classic puzzle (*n-queens*) to evaluate Korat-API. Table 1 shows subjects and their *repOK* logic: (1) *disjoint-set (Disj-Set)*; (2) *sorted doubly-linked-list (DLL)*; (3) *height-balanced binary search tree (HBST)*; (4) *n-queens (Queens)*, a traditional constraint solving problem [3]; (5) *red-black-tree (RBT)*; and (6) *sorted singly-linked-list* with *unique* elements *(SLL)*. All subjects are adapted from Korat open-source repository [29]. Prior studies used similar subjects in their evaluations [6, 12, 34, 38].

For each subject, we wrote the body of *repOK* method as conjunction of other predicates, as specified in Table 1. Note the two different notations used for conjunction: (1) "&&" denotes sequential conjunction and indicates the left-hand-side is a precondition for the right-hand-side, and (2) "∧" denotes logical conjunction and indicates that executing the two predicates in any order gives the same result.

Table 2 shows the list and descriptions of structural properties we implemented, as boolean predicates (methods), in Java language. Each predicate is used in the body of at least one *repOK* method shown in Table 1. For the sake of brevity in presentation, we used the same property names, when referred to the same concept used in different subjects, e.g., the *size()* used in *repOK* of *disjoint-set* is implemented independently of the *size()* used in *doubly-linked-list*.

We run our experiments for small sizes as necessary for bounded exhaustive generation, which requires exploring very large state spaces. For instance, to find all *red-black-tree* structures with 13 nodes, Korat search considers 1,049,391,426 candidates out of a space of more than 2^{213} candidate structures, taking 26 minutes in total and generating 2708 valid structures.

4.2 Results and Answers

4.2.1 RQ1: Speedup in test generation time and reduction in the number of explored candidates. To answer **RQ1**, we implemented each subject in Java, using Korat-API, similar to the *BST* example shown in Figure 3. We selected the type of conjunction between two predicates by using the appropriate method invocations (`and` for "&&", and `land` for "∧").

Recall from Section 3.1 that Korat-API can automatically create an *enhanced* Korat problem, and from Section 3.2, that it can prioritize the logical constraints of a *repOK* predicate.

Table 3 shows the results for constraint prioritization. The first two columns show subjects and input bounds (as number of nodes). The next two columns (3-4) show the worst permutation on top, followed by the execution time (in milliseconds), and number of candidates Korat explored. Columns 5-6 top for each subject, show the best permutation. Column 5 shows speedup in execution time when the best permutation is used, as opposed to the worst permutation. Column 6 shows the reduction (percent) in the number of candidates Korat explored for the best permutation compared to the worst permutation.

For test generation, we considered a time budget of 5 minutes, and structures of up to 10 nodes, for running Korat search on each enumerated permutation. The largest sizes used for `disjoint-set` and `n-queens` were 6 and 9 respectively; other subjects scaled to size 10. For each subject, we observed that the best, and the worst permutations of predicates remained the same across all finitizations. This can be helpful because one can automatically learn the best permutation using Korat-API on a smaller size, and apply the order to generate tests for a bigger size.

Our results show that both speedup and the explored space reduction (percent) increased for larger sizes, achieving speedups in the range 3.86× for `disjoint-set` to 285.71× for `singly-linked-list`, while saving 79.86% and 99.58% of the explored space respectively.

Table 3: Korat Speedup (Time) and Reduction (Explored Space) for Two Extreme Permutations of `repOK` Predicates

	Size	The Worst Permutation		The Best Permutation	
		prop1()∧prop2()∧unique()		*unique()∧prop2()∧prop1()*	
		Time [*ms*]	Explored [#]	Speedup [×]	Reduction [%]
disjoint-set	4	60	26,840	2.07	65.88
	5	2414	1,837,917	3.17	74.23
	6	96,158	187,044,149	3.86	79.86
		sorted()∧link()∧size()		*size()∧link()∧sorted()*	
		Time [*ms*]	Explored [#]	Speedup [×]	Reduction [%]
doubly-linked-list	4	21	5186	3.50	90.92
	5	70	27,996	4.12	92.68
	6	164	144,187	4.21	93.69
	7	529	716,925	4.68	94.40
	8	1968	3,470,206	6.72	94.95
	9	9582	16,447,236	12.99	95.40
	10	48,866	76,645,405	15.97	95.78
		parents()∧bst()∧balanced()		*balanced()∧bst()∧parents()*	
		Time [*ms*]	Explored [#]	Speedup [×]	Reduction [%]
height-balanced-bst	4	9	1099	1.50	55.51
	5	24	7205	1.85	74.45
	6	176	49,985	5.87	89.88
	7	477	362,011	6.54	92.30
	8	2787	2,698,488	7.58	95.79
	9	13,231	20,480,122	30.30	98.08
	10	107,350	157,135,472	88.50	99.13
		diagonal()∧row()		*row()∧diagonal()*	
		Time [*ms*]	Explored [#]	Speedup [×]	Reduction [%]
n-queens	4	11	136	1.22	8.82
	5	16	1429	1.33	42.27
	6	56	19,876	3.50	68.88
	7	423	336,799	5.72	84.57
	8	2539	6,702,256	8.49	92.76
	9	50,476	152,239,945	26.46	96.72
		parents()∧bst()∧colors()		*colors()∧bst()∧parents()*	
		Time [*ms*]	Explored [#]	Speedup [×]	Reduction [%]
red-black-tree	4	13	1251	1.08	29.26
	5	33	7989	1.50	52.22
	6	146	54,117	2.21	72.64
	7	560	384,231	2.74	83.16
	8	2343	2,820,170	3.64	89.56
	9	18,389	21,157,272	10.29	93.40
	10	112,147	160,957,128	14.88	95.62
		unique()∧sorted()∧size()		*size()∧sorted()∧unique()*	
		Time [*ms*]	Explored [#]	Speedup [×]	Reduction [%]
singly-linked-list	4	4	501	1.33	50.30
	5	15	3628	2.14	70.15
	6	44	30,276	4.40	84.17
	7	233	282,285	6.85	92.53
	8	1634	2,903,095	16.34	96.85
	9	19,995	32,659,266	90.91	98.79
	10	255,654	399,168,078	285.71	99.58

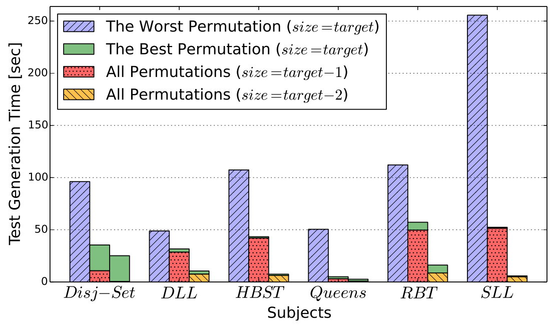

Figure 6: Test generation time for the worst permutation of *repOK* constraints (blue), compared to the time needed to (1) prioritize constraints on a smaller size, e.g., $target{-}1$ (red) or $target{-}2$ (orange); and (2) apply the best permutation found to generate tests for the original size (green)

Recall from Section 3.2 that it can be counter-intuitive for a regular Java user to select the best permutation for Korat search. Interestingly, while implementing these subjects, one co-author originally implemented the `red-black-tree` *repOK* as *parents()∧bst()∧colors()*. The intuition behind this implementation was to follow the natural flow of software evolution, i.e., starting from a `binary-tree` (1) introducing the *parent* field, (2) checking for the *bst* property, and (3) enforcing the *color* property on tree nodes to generate a `red-black-tree`. However, based on Table 3, following this intuition results in the worst permutation for the Korat search; Korat-API determines the exact reverse order as the best permutation.

4.2.2 RQ2: Speedup in test generation when prioritizing constraints on a smaller size. Recall from Section 3.2 that Korat-API uses a heuristic, that constraint prioritization can be done on a smaller size of a problem, and used to generate tests for a larger size of the same problem. To answer **RQ2**, we considered the largest finitization of each subject from Table 3, i.e., *size*=6 for *Disj-Set*, *size*=9 for *Queens*, and *size*=10 for other subjects.

Figure 6 shows test generation times for 3 cases: (1) direct test generation for the *target* size, using the worst permutation; (2) prioritize constraints for *size*=*target*−1, and use the result to generate tests for the *target* size; (3) prioritize constraints for *size*=*target*−2, and use the result to generate tests for the *target* size.

The average test generation speedup for the *target* size was 2.71× and 14.75×, when prioritization was performed on *target*−1 and *target*−2 sizes respectively. The reported times are end-to-end execution times, including the Korat-API translation overhead, the in-memory compilation time, etc. Note the best and the worst permutations remained the same across all studied sizes (Table 3). Hence, one may want to consider prioritizing repOK constraints on a much smaller base size, e.g., for *size*=4.

4.2.3 RQ3: Speedup in model counting time and reduction in the number of explored candidates. Recall from Section 2 that a *heap-PC* is a path condition on fields of heap-allocated objects. To answer **RQ3**, we generated 100 unique heap-PCs for each subject. Each heap-PC was obtained from part of a valid structure created by Korat. We made a partial breadth-first-search (BFS) traversal of the structure using Java reflection. Based on the concrete values of field

Experiment: Construct problems for model counting

```
   Input: subject, size, depth of exploration, random seed.
   Output: 100 unique heap-PCs, implemented as Korat-API programs,
           in 2 approaches: (A1) conjunction of repOK with heap-PC,
           and (A2) finer-grained finitization calls.
1  Korat.init(subject, size)
2  Conditions ← []
3  Calls ← []
4  while Korat.hasNextSolution() do
5      sol ← Korat.nextSolution()
6      conditions ← TraverseUsingReflection(sol, depth)
7      apiCalls ← BuildFixAndExcludeNullAPICalls(sol, depth)
8      PC ← conditions.removeFirst()
9      for condition ∈ conditions do
10         PC ← PC.and(condition)
11     end
12     if PC not ∈ Conditions then
13         Conditions.append(PC)
14         Calls.append(apiCalls)
15     end
16 end
17 Conditions.shuffle(seed)
18 Calls.shuffle(seed)
19 assert size(Conditions) ≥ 100
20 repOK ← subject.getRepOK()
21 fin ← subject.getFinitization()
22 for $i \leftarrow 0$; $i < 100$; $i \leftarrow i + 1$ do
23     newRepOK ← repOK.and(Conditions[i])
24     src ← KoratAPISource(newRepOK, fin, size)
25     writeToDisk(src, Korat.HeapPCPath, subject.name, i)
26     extendedFin ← fin.extend(Calls[i])
27     src ← KoratAPISource(repOK, extendedFin, size)
28     writeToDisk(src, Korat.CallsPath, subject.name, i)
29 end
```

objects visited, we built a path-condition, that consists the three categories of clauses discussed in Section 3.3.

We modeled each heap-PC following the two different approaches discussed in Section 3.3, namely: (A1) the traditional approach of using Korat; and (A2) leveraging the finer-grained finitization calls, i.e., a sequence of *excludeNull* and *fix* calls. Note that the two approaches result in the same final result, though they may have very different performance, as the second one (supported by Korat-API) is capable of more efficiently pruning the state space.

Experiment summarizes our procedure for heap-PC generation. Line 1 initializes the Korat problem for a given subject and size. Line 2 declares a list containing each generated heap-PC using the first approach (A1). Line 3 declares a list to contain generated heap-PCs using the second approach (A2). Lines 4-16 iterate over each valid structure. Line 5 retrieves the next solution. Line 6 traverses a valid test case's object graph up to a given depth and obtains the clauses of heap-PC. Note that we used a depth to maintain conditions for a partial valid substructure since encoding a complete structure in a path condition would restrict model count to just 1. The depth we used, keeps the first 50% of the constraints on a path condition. Line 7 obtains the finer-grained API calls (*fix* and *excludeNull* invocations), corresponding to the clauses stored in *conditions*. Lines 8-15 build the path condition as a conjunction of all clauses, and add (only) unique generated *PCs* and *apiCalls* to their corresponding lists.

Next, Lines 17-18 shuffle the *Conditions* and *Calls* with the same random seed to preserve the relative order between elements of the two lists. Line 19 assures the existence of at least 100 heap-PCs in each list. Lines 20-21 obtain the *repOK* and finitization of the subject. Lines 22-29 build 100 heap-PC problem pairs using Korat-API. Each pair consists of two semantically equivalent Korat-API problems, implemented in the two aforementioned approaches. Specifically, Lines 23-25 build the new *repOK* as conjunction of a subject's *repOK* with clauses of the heap-PC, then store the result as a Korat-API program on disk (A1); Lines 26-28 represent the same problem, this time leaving the *repOK* as just the class invariant, while adding finer-grained finitization calls, i.e., a sequence of *fix* and *excludeNull* invocations (A2).

Table 4 summarizes our results. Column 1 shows the name of subjects we used; we included only subjects that are recursive data structures and allow heap-PCs that have comparisons over multiple reference fields, which are our focus, and excluded *disjoint-set*, which is based on arrays. For each chosen subjects, we considered 10 nodes to generate heap-PC problems. Columns 2-4 show the minimum, maximum, and average number of solutions each generated heap-PC problem found. Columns 5-6 show the average execution times (in milliseconds), and average number of explored candidates, for the heap-PC problems generated by approach 1 (A1). Columns 7-8 show the average speedup and average number of candidates explored for heap-PC problems generated using approach 2 (A2).

We observed that while the two approaches produced the same solutions, A2, supported only by Korat-API, performed 133.96× faster than A1. This outperformance is also evident from the average number of candidates each approach explored (11,464 vs. 8,077,866) resulting in 99.86% reduction in the explored space.

4.3 Threats to Validity

External. The subjects used in our study may not be representative. To mitigate this threat, we used 5 widely-used data structures and a classic puzzle (*n-queens*), previously used in several bounded exhaustive studies [6, 34, 38]. Further, the random *seed* and the chosen *depth* (Section 4.2.2) could affect our results. To mitigate this threat, we repeated our experiments with several different values for seed and depth.

Internal. The default Korat, the Korat-API framework, or our automation scripts may contain bugs. We were somehow confident about the correctness of Korat, as it has been tested, and harvested by several prior studies. To improve our code quality, we did peer code review and wrote unit tests. Further, we injected code assertions at various places in our Korat-API prototype to perform simple sanity checks, e.g., for constraint prioritization (Section 4.2), we checked that each permutation of *repOK* constraints produced the same number of solutions.

Construct. We generated structures of up to size 10. Prior work showed that bounded exhaustive testing can detect many bugs for even smaller sizes [32]. Further, we report both speedup in the execution time, and reduction in the explored space, as both metrics can be useful.

Table 4: Solving 100 Unique Heap-PCs Per Subject in 2 Approaches for size=10, Using Korat-API (800 Total Java Files)

Subject	Count			A1. `repOK() && heap-PC`		A2. `repOK()` + Fine-Grained Finitization	
	Min	*Max*	*Avg*	*Avg. Time* [*ms*]	*Avg. Explored* [#]	*Avg. Speedup* [×]	*Avg. Explored* [#]
doubly-linked-list	1	3003	213.97	3143	3,233,440	32.07	19,945
height-balanced-bst	1	14	3.82	4164	3,716,300	90.52	2857
red-black-tree	1	4	2.05	10341	7,056,576	279.49	2181
singly-linked-list	1	462	26.34	9630	18,305,148	133.75	20,875
Average				6819	8,077,866	133.96	11,464

5 DISCUSSION

Similar to using other constraint solvers [6, 9, 41], there are some potential issues that may arise when using the Korat-API. This section discusses some of these issues, which the user may want to pay particular attention to.

While Korat-API introduces a new way to use Korat as a backend solver and as a test generator, the use of Korat-API requires using a different way to write imperative predicates than is common for writing Java methods. Specifically, if the user manually writes the *repOK*() predicate using Korat-API, the user has to take specific care to conform to Java language syntax and semantics. Using standard Korat, the user can simply leverage a standard IDE, such as Eclipse, which provides incremental compilation and can highlight any compilation errors as the user writes *repOK*(). In contrast, when using Korat-API, Java compilation errors in the *repOK*() body may only be detected much later, and may be harder to debug. However, this issue does not arise when Korat-API is used programmatically by a tool to create constraint solving problems.

Korat-API users also need to take special care when defining logical conjunctions. If the user erroneously defines a conjunction to be logical and the backend solver tries to exploit it by constraint re-ordering, an exception may occur.

Further, for the finer-grained finitization, an important limitation of our implementation is that it requires an in-depth understanding of repOK execution as well as Korat search since by setting fields directly to a specific value from the value domain or to exclude a specific value requires knowledge at the candidate vector level in the Korat search. Given an arbitrary heap-PC, determining the finer-grained field assignments automatically is a challenging problem, which we are considering for future work.

Future work can extend Korat-API to support other optimizations, for example, where Korat problems are solved *incrementally* and solutions to previously solved problems are re-used [11], or *separately* and solutions to sub-problems are combined [43].

6 RELATED WORK

We take inspiration from the success of Kodkod, which has been used in a number of applications [4, 28]. Kodkod provides an API to build Alloy models programmatically and to solve them using off-the-shelf propositional satisfiability (SAT) solvers. The Kodkod API allows declaring the basic relations in the model, writing Alloy formulas over the relations, invoking Alloy commands that direct SAT solving, and reporting the solutions. Kodkod also implements a number of optimizations to provide efficient analysis for Alloy. We believe our Korat-API can leverage further insights into the design and implementation of Kodkod and make Korat even more useful. A key difference between Korat and Alloy's SAT backend is that Korat solves imperative predicates which have a more complex structure and semantics than propositional formulas that SAT solves. However, we believe the basic ideas at the heart of Kodkod can allow future work to create optimized problems that Korat can handle more efficiently. Uzuncaova et al. [42] explored constraint prioritization for Alloy.

Our work is also motivated by recent work on software reliability that uses Korat as a backend model counter [17, 18]. While the goal of these projects, i.e., to use Korat – as is – in a specific problem context, was quite different from our goal of designing Korat-API to facilitate such use of Korat, they guided in part our design.

Parallel Korat [34] introduced a parallel technique for test generation and execution using Korat. This project introduced the idea of ranging where Korat is run to explore a contiguous part of the input space. Follow-up work on PKorat [38] introduced a work-list based algorithm for distributing the search among different workers and used work stealing for load balancing. More recent work [11] introduced infeasible ranges that characterize parts of space that Korat explores without finding any valid input to optimize parallel Korat. Most recently, Korat was optimized using GPUs [7].

Recent incremental techniques for Korat [11, 12] memoized key steps of Korat search in solving one constraint solving problem and re-used them when possible to solve the next problem. Our Korat-API introduces a foundation that can support (in future work) incremental analysis directly by providing appropriate constructs, e.g., to incrementally define constraints and finitization, thereby bringing the spirit of incremental SAT to Alloy.

Static analysis [40] and dynamic analysis [15, 30, 39], have also been used to enhance Korat. The goal of these analyses is to guide the Korat search to reduce the amount of exploration. Our Korat-API provides a way to directly guide some of these analyses and more directly improve the Korat search's effectiveness.

UDITA [22] builds on Korat and integrates test generation using a combination of imperative constraints and dedicated generators, where a structure is generated in part by solving the constraints and in part by employing the generator. While a generator in UDITA could support the lower-level requirements, it does not support optimized solving based on constraint re-ordering.

While Korat and its successor techniques use imperative constraints, a number of techniques use declarative constraints, e.g., written in first-order logic, for systematic testing and bug finding [20, 25, 33, 36]. Jackson and Vaziri introduced an encoding of a subset of Java into Alloy and used SAT for static checking.

TestEra [33] introduced bounded exhaustive testing using constraints based on the Alloy tool-set. TACO [20] translates JML-annotated Java code to Alloy and use its Kodkod backend for bug finding and make the overall analysis efficient by using symmetry breaking and introducing tight-bounds. BLISS [36] uses tighter bounds in the context of lazy initialization [26] for symbolic execution and combines it with SAT for more efficient analysis. Our work on Korat-API has insights in common with these techniques; the key difference is our focus on providing an API for more effective use of Korat as a constraint solver.

Korat is one of many frameworks for automated testing [2, 19, 35, 37]. The specific reliability technique [17, 18] that motivated part of our Korat-API framework design is also just one of many techniques that researchers have developed in this rich research area of software reliability [5, 8, 13, 16, 21]. We believe Korat-API can help with design of APIs for other testing and reliability tools that serve as backend engines.

7 CONCLUSION

We presented Korat-API, a novel framework that introduces an API for building constraint solving problems, and algorithms for solving them to enhance Korat. Our goal was two-fold: (1) to facilitate the use of Korat as an optimized standalone constraint solver, or as a backend engine; (2) to optimize Korat for more efficient constraint solving by exploiting problem-specific information. We described our API and algorithms that embody our framework. We further presented an experimental evaluation using a suite of complex constraint solving problems. The experimental results demonstrated that using Korat-API can provide up to two orders of magnitude speedup for test generation and model counting. We believe our work introduces a promising direction into making the use of imperative constraints, in general, and Korat, in particular, more widely applicable for improved software testing and reliability.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. Additionally, we thank Ahmet Celik, Milos Gligoric, Rui Qiu, and Marko Vasic for their constructive comments on this work. This research was partially supported by the US National Science Foundation under Grants Nos. CCF-1319688 and CNS-1239498.

REFERENCES

[1] Zakaria Alrmaih. 2017. *Korat-API: An API for building constraint solving problems for Korat.* Master's thesis. University of Texas at Austin.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* (2013).

[3] Krzysztof R. Apt and Mark Wallace. 2007. *Constraint Logic Programming Using Eclipse.* Cambridge University Press.

[4] Hamid Bagheri and Sam Malek. 2016. Titanium: efficient analysis of evolving Alloy specifications. In *FSE*.

[5] Christian Bird, Venkatesh-Prasad Ranganath, Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. 2014. Extrinsic influence factors in software reliability: a study of 200, 000 windows machines. In *ICSE*.

[6] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. In *ISSTA*.

[7] Ahmet Celik, Sreepathi Pai, Sarfraz Khurshid, and Milos Gligoric. 2017. Bounded Exhaustive Test-Input Generation on GPUs. In *OOPLSA*.

[8] Marcello Cinque, Claudio Gaiani, Daniele De Stradis, Antonio Pecchia, Roberto Pietrantuono, and Stefano Russo. 2014. On the Impact of Debugging on Software Reliability Growth Analysis: A Case Study. In *ICCSA*.

[9] Leonardo de Moura and Nikolaj Bjorner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

[10] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover. In *CADE-25*.

[11] Nima Dini. 2016. *MKorat: A Novel Approach for Memoizing the Korat Search and Some Potential Applications.* Master's thesis. University of Texas at Austin.

[12] Nima Dini, Cagdas Yelen, and Sarfraz Khurshid. 2017. Optimizing Parallel Korat Using Invalid Ranges. In *SPIN*.

[13] Romney B. Duffey and Lance Fiondella. 2014. Software, Hardware, and Procedure Reliability by Testing and Verification: Evidence of Learning Trends. *IEEE Trans. Human-Machine Systems* (2014).

[14] Niklas Een and Niklas Sorensson. 2003. An Extensible SAT-solver. In *SAT*.

[15] Bassem Elkarablieh, Darko Marinov, and Sarfraz Khurshid. 2008. Efficient solving of structural constraints. In *ISSTA*.

[16] Felipe Febrero, Coral Calero, and Maria Ángeles Moraga. 2014. A Systematic Mapping Study of Software Reliability Modeling. *Information & Software Technology* (2014).

[17] Antonio Filieri, Marcelo F. Frias, Corina S. Pasareanu, and Willem Visser. 2015. Model Counting for Complex Data Structures. In *SPIN*.

[18] Antonio Filieri, Corina S. Pasareanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *ICSE*.

[19] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *ISSTA*.

[20] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. 2013. TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds. *Transactions on Software Engineering* (2013).

[21] Carlo Ghezzi, Mauro Pezzè, and Giordano Tamburrelli. 2013. Adaptive REST applications via model inference and probabilistic model checking. In *IM*.

[22] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *ICSE*.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*.

[24] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press.

[25] Daniel Jackson and Mandana Vaziri. 2000. Finding Bugs with a Constraint Solver. In *ISSTA*.

[26] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*.

[27] James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19, 7 (1976).

[28] Kodkod applications website. 2017. http://emina.github.io/kodkod/apps.html. (2017).

[29] KoratWebPage 2017. Korat Home Page. (2017). http://korat.sourceforge.net/index.html.

[30] Amresh Kulkarni. 2007. *Constraint Prioritization for Efficient Test Generation Using Korat.* Master's thesis. University of Texas at Austin.

[31] Barbara Liskov and John Guttag. 2000. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.*

[32] Darko Marinov. 2004. *Automatic Testing of Software with Structurally Complex Inputs.* Ph.D. Dissertation. Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

[33] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE*.

[34] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. 2007. Parallel test generation and execution with Korat. In *FSE*.

[35] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*.

[36] N. Rosner, J. Geldenhuys, N. M. Aguirre, W. Visser, and M. F. Frias. 2015. BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support. *Transactions on Software Engineering* (2015).

[37] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *ASE*.

[38] Junaid Haroon Siddiqui and Sarfraz Khurshid. 2009. PKorat: Parallel Generation of Structurally Complex Test Inputs. In *ICST*.

[39] Junaid Haroon Siddiqui, Darko Marinov, and Sarfraz Khurshid. 2009. Optimizing a Structural Constraint Solver for Efficient Software Checking. In *ASE*.

[40] Raghavendra Srinivasan. 2015. *Improving constraint-based test input generation using Korat.* Master's thesis. University of Texas at Austin.

[41] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS*.

[42] Engin Uzuncaova and Sarfraz Khurshid. 2008. Constraint Prioritization for Efficient Analysis of Declarative Models. In *FM*.

[43] Hua Zhong, Lingming Zhang, and Sarfraz Khurshid. 2016. Combinatorial generation of structurally complex test inputs for commercial software applications. In *FSE*.