Optimizing Parallel Korat Using Invalid Ranges

Nima Dini
The University of Texas at Austin
Austin, Texas 78712
nima.dini@utexas.edu

Cagdas Yelen
The University of Texas at Austin
Austin, Texas 78712
cagdas@utexas.edu

Sarfraz Khurshid
The University of Texas at Austin
Austin, Texas 78712
khurshid@ece.utexas.edu

ABSTRACT

An effective form of systematic testing is *constraint-based testing* where the user writes logical constraints to describe properties of desired inputs, and constraint solvers enumerate all tests within a bound on the input size. The key challenge in systematic constraint-based testing is efficiently exploring very large spaces of all possible inputs to enumerate the desired valid inputs. Previous work introduced the Korat technique to address this challenge. Korat uses desired input properties written as imperative predicates and performs a backtracking search that prunes large parts of the input space to enumerate all non-equivalent (i.e., non-isomorphic) inputs within a given bound on input size. While Korat's pruning improves its performance, systematically generating large numbers of inputs and testing against them can be costly in practice.

This paper introduces a novel approach to reduce the cost of the Korat search in certain application scenarios. Our key insight is that sometimes the Korat search over the *same* state space is repeated across *separate* runs of Korat, and an earlier run of the search can be summarized to more efficiently perform a later run. Specifically, we introduce the idea of *invalid ranges* which succinctly encode parts of the exploration space that do not contain any valid inputs but have to be explicitly explored by the Korat search since it is unable to prune them. Our approach directly prunes these parts in a future run of Korat over the same input space. We develop our approach for two settings: a sequential setting where the Korat search is run using one worker (i.e., processing unit), and a parallel setting where the Korat search is distributed to several workers. In the parallel setting, we build on a previous technique for parallel Korat, namely *SEQ-ON*, and integrate invalid ranges with it. Our prototype tool MKorat embodies our approach. Experimental evaluation using 6 subjects from the standard Korat distribution show that MKorat achieves: in the sequential setting, a speedup of up to 2.82X over sequential Korat (in comparison, SEQ-ON does not provide any speedup in the sequential setting); and in the distributed setting, using up to 32 workers, a speedup of up to 38.84X over sequential Korat (using 1 worker), and up to 3.04X over SEQ-ON in terms of total execution time across the workers.

KEYWORDS

Constraint-based testing, Test input generation, Korat, Parallel search

Submitted to SPIN'17, Santa Barbara, California USA

DOI: 10.475/123_4

1 INTRODUCTION

Systematic software testing [1, 3, 5, 13, 14, 16, 21, 25], which has its roots in the core idea of systematic exploration of bounded state spaces in model checking [6, 14, 37], has been used in a number of applications for finding subtle bugs in software systems. A particularly effective approach for systematic testing is *constraint-based testing* where logical constraints characterize desired inputs and expected program behaviors as preconditions and postconditions respectively [3, 25]. A number of different techniques embody this approach and support constraints written in different languages, including declarative languages [25] and imperative languages [3].

Our work focuses on the constraints written as *imperative predicates*, termed `repOK` [24], which are executable checks that characterize desired properties using an imperative language, e.g. Java, and likely pose minimal learning burden on users because of the wide use of such languages. The foundation of our work is the *Korat* technique for test input generation using imperative constraints [3, 26]. Given a `repOK` predicate, which characterizes desired inputs, and a *finitization*, i.e., a bound on the input size, Korat enumerates each *non-isomorphic* input within the bound such that executing `repOK` on the input returns true. Thus, the inputs generated by Korat form *bounded exhaustive tests* and include every *valid* input with respect to the given `repOK` and finitization bound. The space of all candidates to consider as inputs to `repOK` is usually very large, e.g., $> 2^{72}$, even for small bounds on input size, e.g., 10 nodes in a binary search tree [3]. The Korat algorithm performs pruning and isomorph-breaking to exhaustively explore such large input spaces. However, the application of Korat in practice is limited by two key factors: the size of the underlying state spaces and the number of valid inputs created.

We introduce a novel approach to reduce the cost of the Korat search in certain application scenarios. Our key insight is that the Korat search is sometimes repeated over the *same* state space across *separate* runs of Korat, and an earlier run of the search can be summarized to more efficiently perform a later run. Such a scenario arises, for example, when Korat search is used for testing multiple methods where some of the methods share a common input constraint but storing (all) the inputs is not feasible [26]: the testing approach chooses one method to test, and iteratively creates an input, runs the method, and checks its behavior until testing this chosen method is complete, and then selects the next method, and continues until all methods have been tested. Another example scenario arises when Korat is used as an external constraint solver [11]. To illustrate, consider systematic *checking* [20, 21] of two methods $m()$ and $n()$ of a class with class invariant $inv()$, which represents a precondition for both $m()$ and $n()$. Checking $m()$ requires checking the program p: "`if (inv()) m();`" and checking $n()$ requires checking the program q: "`if (inv()) n();`". Thus, checking $m()$ and $n()$ requires *solving* the same constraint $inv()$ –

all execution paths that reach $m()$ in p (likewise $n()$ in q) require $inv()$ to evaluate to true. Indeed, the same optimization opportunity arises when $n()$ is actually just an updated version of $m()$, which needs to be re-checked say after on a bug fix or feature addition in the context of evolution.

We introduce the idea of *invalid ranges* which succinctly encode parts of the exploration space that do not contain any valid input but must be explicitly explored by the Korat search since it is unable to prune them [8]. Our approach prunes these parts in a future run of Korat over the same input space. We develop our approach for two settings: a sequential setting where the Korat search is run using one worker (i.e., processing unit), and a parallel setting where the Korat search is distributed to several workers. In the sequential setting, we directly use invalid ranges to prune the search. In the parallel setting, We build on a previous technique, namely *SEQ-ON* [26], and integrate invalid ranges with it.

SEQ-ON was defined by the parallel Korat approach [26], which originally introduced the idea of parallel test generation and execution using Korat to mitigate the two key limiting factors. Conceptually, parallel execution of tests generated using Korat is relatively straightforward: distribute the tests evenly among the parallel workers. However, parallel generation of tests using Korat is a non-trivial problem because Korat's pruning is inherently *sequential*: what to prune depends on what was explored and cannot simply be determined a priori. Specifically, Korat considers one candidate input at a time, checks the validity of the current candidate by executing `repOK` against it, and uses the execution as a basis of creating the next candidate, and by doing so prunes many candidates from the search. Thus, evenly distributing the test generation workload among parallel Korat workers is challenging.

SEQ-ON was specifically designed to address the scenario where Korat is used to create inputs for testing a number of different methods under test but the inputs are *not* stored: for each method under test, inputs are created and the method executed against each input as it is created. A key contribution of the SEQ-ON algorithm is that it uses the first execution of Korat for input generation to create a fixed number of *equidistant* candidates based on the number of workers, which allows all subsequent executions of Korat on the same constraint solving problem to be performed in parallel such that each parallel worker only explores the *range* defined by two consecutive equidistant candidates, and the workload is evenly distributed among the parallel workers.

We define invalid ranges as sequences of consecutive invalid candidates that are explored by the standard Korat search. We summarize such a sequence succinctly using just two candidates as end-points during the first execution of Korat for input generation, and re-use it for more efficient exploration in the subsequent executions of Korat for input generation – the subsequent executions are able to *prune* invalid candidates that the initial Korat search was unable to prune and had to explicitly check using `repOK`. We apply invalid ranges in tandem with equidistant candidates to define a more effective technique for parallel test generation using Korat.

Our prototype tool MKorat embodies our approach. We show the effectiveness of our approach using a suite of controlled experiments. Specifically, we evaluate how MKorat compares with traditional Korat in a sequential setting and how the use of invalid ranges improves over just equi-distancing. Moreover, we evaluate how the performance of MKorat varies as the number of invalid ranges is increased.

This paper makes the following contributions:

- **Idea.** We introduce the idea to summarize and re-use parts of the state-space that do *not* contain any valid input to enhance solving of imperative constraints for systematic input generation.
- **Invalid ranges.** We define invalid ranges, which succinctly represent consecutive invalid candidates that the standard Korat search is unable to prune and must explicitly check by invoking `repOK`.
- **Test generation technique.** We introduce a new technique to optimize input generation using imperative constraints when the Korat search is re-run for the same exploration space. We develop our technique for a sequential setting and a parallel setting by building on the SEQ-ON algorithm from previous work [26] and integrating invalid ranges with it.
- **Evaluation.** We use a suite of 6 subjects from the standard Korat distribution to evaluate our approach. Experimental results show that MKorat achieves: in the sequential setting, a speedup of up to 2.82X over sequential Korat (in comparison, SEQ-ON does not provide any speedup in the sequential setting); and in the distributed setting, using up to 32 workers, a speedup of up to 38.84X over sequential Korat (using 1 worker), and up to 3.04X over SEQ-ON in terms of total execution time across the workers.

We believe invalid ranges provide the foundation for an exciting method for increasing the efficacy of systematic testing and analysis using imperative constraints. While our focus in this paper is on re-execution of the Korat search over the same state space as the previous execution, we believe invalid ranges can be generalized to enable re-use in more general settings where state spaces among different executions differ, e.g., due to a change in the constraint being solved or the bound being used. We plan to address such settings in future work.

2 ILLUSTRATIVE EXAMPLE

We illustrate the concept of invalid ranges and the basis of our approach using an example from the Korat project's source code[1]. Figure 1 shows the Java declaration of the red-black tree data structure [7], which implements a balanced binary search tree, the `repOK` predicate that implements a check for the structural integrity constraints of a red-black tree (acyclicity, correct coloring of nodes etc.), and the finitization description (the *finRedBlackTree* method) which sets a bound on Korat search. Each tree has a `root` node and caches the number of nodes in the size field. Each node contains an integer `key` and `value`, and has a `left` child, a `right` child, and a `parent` pointer, as well as a `color`, which is `RED` or `BLACK`. To test a method that operates on an input red-black tree, such as instance method "`add(int x)`", we must generate a valid tree `t`, i.e., `t.repOK()` returns true, as an input (the receiver object) as well as provide an integer input `x`. To create valid red-black trees, Korat uses the given `repOK` method and finitization to create the space

[1] https://korat.svn.sourceforge.net/svnroot/korat/trunk, revision 12

```
1  public class RedBlackTree {
2    private Node root = null;
3    private int size = 0;
4    private static final int RED = 0;
5    private static final int BLACK = 1;
6    public static class Node  {
7      int key;
8      int value;
9      Node left = null;
10     Node right = null;
11     Node parent;
12     int color = BLACK;
13   }
14   public boolean repOK() { ... }
15
16   // Bound Korat exploration to trees with 'num' Nodes.
17   public static IFinitization finRedBlackTree(int num) {
18     IFinitization f = FinitizationFactory.create(
19       RedBlackTree.class);
20     IClassDomain entryDomain = f.createClassDomain(
21       Node.class, num);
22     IObjSet entries = f.createObjSet(Node.class, true);
23     entries.addClassDomain(entryDomain);
24
25     IIntSet sizes = f.createIntSet(num, num);
26     IIntSet keys = f.createIntSet(-1, num - 1);
27     IIntSet values = f.createIntSet(0);
28     IIntSet colors = f.createIntSet(0, 1);
29
30     f.set("root", entries);
31     f.set("size", sizes);
32     f.set("Node.left", entries);
33     f.set("Node.right", entries);
34     f.set("Node.parent", entries);
35     f.set("Node.color", colors);
36     f.set("Node.key", keys);
37     f.set("Node.value", values);
38
39     return f;
40   }
41 }
```

Figure 1: RedBlackTree subject[1].

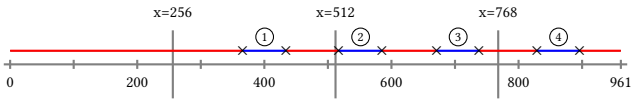

Figure 2: Korat generates 961 candidates for *finRedBlackTree(4)*. A valid candidate index is marked by a cross (×).

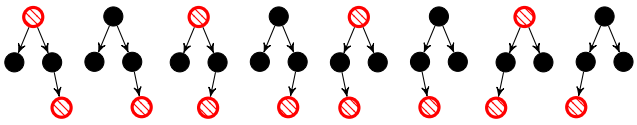

Figure 3: Valid red black trees with 4 nodes generated by Korat. Node keys are uniquely assigned from set $S = \{1, 2, 3, 4\}$.

of candidate inputs to explore and generates all inputs for which `repOK` returns true.

The Korat search operates on *candidate vectors*, which are integer arrays that encode object graphs and allow efficient backtracking. Figure 2 shows the consecutive range of all candidate vectors generated by Korat search for *finRedBlackTree(4)*, which are red-black trees containing exactly 4 nodes. Korat explores the total number of 961 non-isomorphic candidates, out of which only 8 satisfy the `repOK` predicate. Figure 3 shows all 8 valid instances found[2].

Korat supports *ranging* the search, i.e., bounding it to explore a subset of the space of all candidates [26]. Specifically, given a pair of *start* and *end* candidate vectors, such that the standard Korat search would explore start before end, Korat can be ranged to only search for valid structures between start and end. Ranging allows the distribution of Korat execution across several individual workers. The *SEQ-ON* equi-distancing algorithm in prior work [26] introduces a technique to distribute Korat execution for future runs of the same search.

For the *finRedBlackTree(4)* example, given 4 workers, the equi-distancing algorithm splits the explored candidates into 4 partitions, each with the same approximate size of 256 candidate vectors[3]. As shown in Figure 2, there are 8 valid candidates among 961 explored, i.e., 99.16% of the candidates explored are invalid and represent *redundant* search. The explored indexes of the valid candidate vectors in this example are: 366, 434, 517, 585, 671, 738, 829, and 896. The main pitfall of SEQ-ON algorithm is that while it parallelizes the executions, the total number of explored candidates across all workers remains the same as the sequential run. Since exploring invalid candidates is redundant, our approach tries to prune them and not to re-explore them.

We define an *invalid range* as a sequence of consecutive invalid candidates explored by Korat search, such that `repOK` predicate returns *false* on all of them. MKorat removes a bounded number ($m \geq 1$) of invalid ranges as desired by the user. Further, MKorat distributes the remaining ranges among $k \geq 1$ workers with respect to the k equidistant ranges maintained by SEQ-ON. Given $m = 3$, MKorat removes the 3 largest invalid ranges [435, 517), [586, 671), and [739, 829), in addition to the head and tail invalid ranges, i.e., [0, 366) and [897, 961]. In total 5 invalid ranges will be removed that are highlighted in red.

The only 4 remaining ranges needed to be re-explored (highlighted blue) are: [366, 435), [517, 586), [671, 739), and [829, 897). Worker one remains idle, as the range [0, 256) belongs in a known invalid range. Worker two takes subrange ①. Worker three explores subranges ② and ③. The final worker takes over range ④. The result is 71.48% reduction in the re-explored state space, while using fewer computing resources. Note that for this example, MKorat can achieve a higher reduction for larger values of invalid ranges if so desired by the user.

3 TECHNIQUE

In this section, we first describe our technique to build invalid ranges using an initial run of Korat. Next, we recall the original *SEQ-ON* algorithm [26] (Section 3.1) and present our parallel technique MKorat, which builds on SEQ-ON (Section 3.2). Next, we discuss some implementation details of MKorat (Section 3.3) and several key properties of our prototype (Section 3.4).

Our core approach performs an initial run of Korat to build invalid ranges (Figure 4), which are used for additional pruning in subsequent runs of Korat when exploring the same input space. In

[2]The root of a red-black tree should be colored black. However, this rule can be relaxed, as in the provided `repOK`, because the root can always be changed from red to black.
[3]In this example, the last range has 961-768=193 elements.

```
Input: subject w/ repOK and fin; user provided upper bound m
 1: function MKorat(subject, m)
 2:     Korat.init(subject)
 3:     prevCV ← Korat.getStartCandidateVector()
 4:     ranges ← boundedDescendingSortedTree(m)
 5:     while Korat.hasNextValidCandidate() do:
 6:         validCV, numExplored ← Korat.nextValidCandidate()
 7:         ranges.insert(tuple(prevCV, validCV), numExplored)
 8:         prevCV ← validCV
 9:     done
10:     if Korat.hasNextCandidate() then:
11:         endCV, numExplored ← Korat.lastCandidate()
12:         ranges.insert(tuple(prevCV, endCV), numExplored)
13:     fi
14:     return ranges.buffer()
15: end function
```

Figure 4: Collecting an upper bound of m invalid ranges in Korat search.

addition to the inputs that the standard Korat algorithm takes, our approach takes as input an upper bound ($m > 1$) on the number of invalid ranges to maintain and reuse. Increasing this number of invalid ranges likely allows more efficient subsequent search but increases the storage (space) requirements. Line 2 (in Figure 4) initializes the Korat search problem for a given subject. Line 3 sets `prevCV` to the initial candidate vector Korat has to explore. Next, Line 4 builds a max heap data structure to contain the m largest candidate vector `tuples`, plus *head* and *tail* (Section 3.2) invalid ranges upon first and last insertion respectively (if they exist). Lines 6 and 7 finds the next valid candidate by running Korat search and inserts a new invalid range into `ranges` encoded by a tuple of candidates plus the size of the invalid range. Lines 11 and 12 find the *tail* invalid range (if any) and insert it into the `ranges`. Finally Line 14 returns the invalid ranges maintained.

3.1 Background: Equi-distancing for SEQ-ON

The key novelty of the original parallel test generation and execution algorithm using Korat [26] (SEQ-ON) is to not store all inputs for creating equidistant candidates. The design goal behind this algorithm is to store sufficient information during the first sequential run, so that all future runs can be parallelized and load-balanced. Specifically, this algorithm obtains a sequence of equidistant candidate vectors $\langle C_1, C_2, ..., C_n \rangle$, i.e., Korat explores (almost) the same number of candidates in any range $[C_i, C_{i+1})$ for $0 \leq i < n-1$ and $[C_{n-1}, C_n]$, and the union of all such ranges, becomes the entire explored space $\bigcup_{i=0}^{n-2}[C_i, C_{i+1}) \bigcup [C_{n-1}, C_n] = [C_0, C_n]$, where C_0 is the initial and C_n is last candidate vectors explored.

Figure 5 shows the pseudo-code of the SEQ-ON algorithm. The *equiDistantCandidates* function keeps an array of candidates, with size twice as large as the number of maximum workers. As the number of explored candidates in Korat search is not known beforehand, this technique records each candidate being explored in the first round. When the array capacity is full, it moves the candidates at even indexes in the array to left half, and continues recording every second candidate in the right half. In the next round, it records every fourth candidate being explored. This process continues, and at the end, the function returns the candidates to keep for the future parallel executions.

```
1  // input: 'm' is the maximum number of workers
2  // output: an array of equidistant candidates,
3  // with the array length between m and 2 * m
4  Candidate[] equiDistantCandidates(int m) {
5    Candidate[] candidates = new Candidate[2 * m];
6    int distance = 1;
7    int index = 0;
8    while (Korat.hasNext()) {
9      for (int i = 0; i < distance; i++) {
10       candidates[index] = Korat.next();
11       if (!Korat.hasNext()) break;
12     }
13     if (index < candidates.length) index++;
14     else {
15       // half the array and double the distance
16       for (int j = 0; j < candidates.length / 2; j++)
17         candidates[j] = candidates[2 * j + 1];
18       distance = distance * 2;
19       index = m;
20     }
21   }
22   // resize the output length to valid indexes
23   Candidate[] result = new Candidate[index];
24   for (int i = 0; i < index; i++)
25     result[i] = candidates[i];
26   return result;
27 }
```

Figure 5: Equi-distancing for SEQ-ON [26].

3.2 MKorat

Our parallel technique, MKorat, builds on SEQ-ON and stores $m \geq 1$ largest invalid ranges during the first sequential run, and excludes those ranges prior to distribution for future parallel runs. The remaining ranges $\langle r_1, r_2, ..., r_s \rangle$, are distributed with respect to the $k \geq 1$ equidistant candidate vectors maintained in SEQ-ON. Moreover, if any equidistant candidate vector C_y falls into a range $r_i = [C_x, C_z)$, the algorithm breaks the range to $[C_x, C_y)$, $[C_y, C_z)$. The splitting phase continues until no range in the final collection of ranges $Q = \langle q_1, q_2, ..., q_t \rangle$ contains an equidistant candidate unless it is the starting endpoint of a range. Finally, each worker takes a subset of ranges from Q which belong to its equi-distant range.

Note that if an equi-distant range does not contain any valid candidate, it will be discarded and no worker will be assigned to that range, saving computational resources. Further, MKorat borrows the single-pass spirit of SEQ-ON and does not impose extra time and space complexity on top of this algorithm.

MKorat$_{exc}$: Korat provides a command-line option *--cvWrite*, which writes all explored candidates to a serialized file *f*. Further, two additional options *--cvStart <num1>* and *--cvEnd <num2>* are supported to limit the search exploration range to the *num1-st* and *num2-nd* candidates from file *f*. For two main reasons this existing option was not sufficient:

(1) Writing all generated tests of a sequential execution to a file can be prohibitively expensive, e.g., for *finRedBlackTree(12)* the size of the *candidates.dat* file on disk is about 7.63GB. However, to re-explore a range (for both SEQ-ON and MKorat), only *start* and *end* candidates are required. Hence, we overrode the *--cvWrite* command-line option to only writes the endpoints of desired ranges into a file.
(2) Due to existence of invalid ranges, the distribution phase of MKorat may assign a worker several subranges to run.

Option	Values
--version	0: Korat, 1: SEQ-ON, 2: MKorat, 3: MKorat$_{exc}$
--equi	# of equidistant candidates
--invalid	# of invalid ranges
--subranges	set of <start, end> candidate vector pairs to run

Table 1: Korat extended command-line options

For example, worker three in Figure 2 takes two subranges. We implemented an extension of Korat, namely MKorat$_{exc}$ which takes arbitrary number of subranges (<start, end> candidate vector pairs), and explores only candidates within those given subranges.

For each Korat exploration, there are two special ranges which only contain invalid candidates, namely *head* and *tail* invalid ranges, described below. MKorat safely removes these two ranges in addition to the $m \geq 1$ parameter provided by the user:

- *head invalid range* is the range $[C_0, C_v)$ where C_0 is the initial candidate vector and C_v is the first valid candidate generated by the Korat search. For example, the range [0, 366) is the head invalid range in Figure 2. Note in case that C_0 is equal to C_v, the range $[C_0, C_v)$ contains no element, and *head invalid range* does not exist.
- *tail invalid range* is the range $(C_w, C_n]$ where C_n is the last candidate vector and C_w is the last valid candidate generated by the Korat search. For instance, (896, 961] is the tail invalid range in Figure 2. In case C_w is equal to C_n, the range $(C_w, C_n]$ is empty and *tail invalid range* does not exist.

3.3 Implementation

Table 1 shows the new run-time options we introduced in our MKorat framework. Option *--version* chooses between the 4 techniques implemented within Korat framework, namely: the original Korat, SEQ-ON, MKorat, and MKorat$_{exc}$. Next, option *--equi* determines the number of equidistant candidates for SEQ-ON and MKorat. Note that this number cannot be greater than the total number of explored candidates. Therefore, our implementations will use the minimum of the two numbers. Option *--invalid* is the number of invalid ranges MKorat considers. Similar to the *--equi* option, if this option exceeds the total number of invalid ranges, the minimum of the two values will be selected. Finally, option *--subranges* specifies subranges (<start, end> candidate pairs) to run. Table to the right shows which options are required for each Korat extension we used in our study.

	Option		
Version	equi	invalid	subranges
0	✗	✗	✗
1	✓	✗	✗
2	✓	✓	✗
3	✗	✗	✓

3.4 Properties

Given a Korat search problem (repOK and finitization), we define the *Reduction* achieved by MKorat as follows:

$$Reduction = \frac{\#\ of\ invalid\ candidates\ MKorat\ prunes}{\#\ of\ candidates\ Korat\ explores}$$

The denominator of the equation above is the total number of candidates Korat explores in a sequential run, which is the same number workers re-explore in SEQ-ON algorithm. The numerator is the number of invalid candidates MKorat prunes for future runs. Given a large enough $m \geq 1$ parameter, MKorat can achieve the $Reduction_{max}$ by pruning all existing invalid ranges from future executions. MKorat has the following properties:

(1) Maximum candidates re-explored by a single worker, is at most equal to the number explored by a worker in SEQ-ON algorithm, because MKorat assigns each worker a subset of its original equi-distant range.
(2) By definition of *Reduction*, the larger the number of explored invalid candidates are (compared to the valid explored candidates), the better MKorat is expected to perform. As an extreme case, for the constant returning repOK, i.e, return *true* or *false*, $Reduction_{max}$ will be 0% and 100% respectively.
(3) The number of valid instances Korat finds for a given subject is an upper bound on the number of invalid ranges that subject can have.

4 EVALUATION

We evaluate the effectiveness of MKorat, on a suite of standard subjects shipped with Korat and used in prior studies. This section describes the experiment procedure we designed to answer the following questions:

Q1. *Can MKorat achieve $Reduction_{max}$?*

Q2. *How does the number and distribution of valid candidate vectors affect MKorat reduction?*

Q3. *What are the practical benefits of MKorat in terms of execution time and required computational resources for sequential and parallel settings?*

4.1 Study

Table to the right shows the 6 subjects used in our study, which are taken from Korat's open-source repository[1]. Prior studies used similar subjects in their evaluation [3, 26, 29]. Due to the bounded exhaustive nature of Korat search, running these subjects does not scale for large finitization values. For instance, given $finRedBlackTree(12)$ for the red-black tree example discussed in Section 2, Korat explores 205,512,574 candidates in 4 minutes and finds 1,296 valid structures. We evaluated the effectiveness of MKorat for each subject, and compared it with the original SEQ-ON algorithm discussed in Section 3.1, with respect to the *reduction* definition in Section 3.4.

Subjects
BinaryTree (*BT*)
BinomialHeap (*BH*)
DoublyLinkedList (*DLL*)
RedBlackTree (*RBT*)
SearchTree (*ST*)
SinglyLinkedList (*SLL*)

4.2 Results

Table 2 shows basic information obtained by Korat execution for our 6 subjects. This table includes the number of candidates explored, valid instances found, and number of invalid ranges for 5 different finitizations. Recall from section 3.2 that MKorat safely removes the head and tail invalid ranges from the explored range; hence, the number of invalid ranges in table 2 excludes these two ranges (All 6 subjects across different finitizations had head and tail invalid ranges). As shown in Table 2, the number of candidates

	# of	Finitization 2	4	6	8	10
BT	Explored candidates	16	245	3653	54418	815100
	Valid candidates	2	14	132	1430	16796
	Invalid ranges	1	13	131	1429	16795
BH	Explored candidates	58	1666	42815	1323194	150727471
	Valid candidates	6	120	7602	603744	117157172
	Invalid ranges	3	23	941	33555	6628009
DLL	Explored candidates	27	94	776	17166	562823
	Valid candidates	3	37	674	17007	562595
	Invalid ranges	0	0	0	0	0
RBT	Explored candidates	40	961	16487	322806	7530712
	Valid candidates	2	8	20	64	260
	Invalid ranges	1	7	19	63	259
ST	Explored candidates	22	875	45233	2606968	155455872
	Valid candidates	2	14	132	1430	16796
	Invalid ranges	1	13	131	1429	16795
SLL	Explored candidates	17	139	2194	52567	1702171
	Valid candidates	2	15	203	4140	115975
	Invalid ranges	1	14	202	4139	115974

Table 2: Number of candidates explored, valid instances found, and invlid ranges explored by Korat.

Subject	Invalid ranges 1	4	16	64	256	1024
BT	0.37	0.88	2.71	8.91	28.48	80.25
BH	12.77	34.21	45.76	45.84	46.10	46.79
DLL	0.92	0.92	0.92	0.92	0.92	0.92
RBT	48.34	58.45	69.22	99.98	99.98	99.98
ST	0.17	0.43	1.40	5.01	18.81	72.54
SLL	0.22	0.50	1.45	4.54	13.84	38.79

Table 3: MKorat *Reduction* [%] for *Fin* = 8.

explored grows considerably as the finitization increases, which shows Korat's bounded exhaustive testing technique does not scale with the finitization growth.

Based on Table 2, given a finitization, the number of invalid ranges for all subjects, except *DLL* and *BH*, is always 1 unit greater than valid instances found (if head and tail invalid ranges are counted). Our investigation showed that no two valid candidates are consecutive in the explored candidates of these 4 subjects. For *DLL*, the number of invalid ranges is 0, because all valid candidates are consecutive and the only 2 invalid ranges are the head and tail. Finally, *BH* has both consecutive and non-consecutive valid candidates, resulting in a number of invalid ranges between 0 and number of valid candidates.

For a fixed finitization, the number of invalid ranges obtained for each subject in Table 2, is an upper bound on the number of ranges MKorat can remove from the explored space. For instance, for the *RBT* of size 8, at most 63 invalid ranges can be removed and running MKorat for any $m > 63$ number of invalid ranges does not increase the reduction. Table 3 shows the reduction achieved by MKorat for finitization = 8, provided by 6 different values for

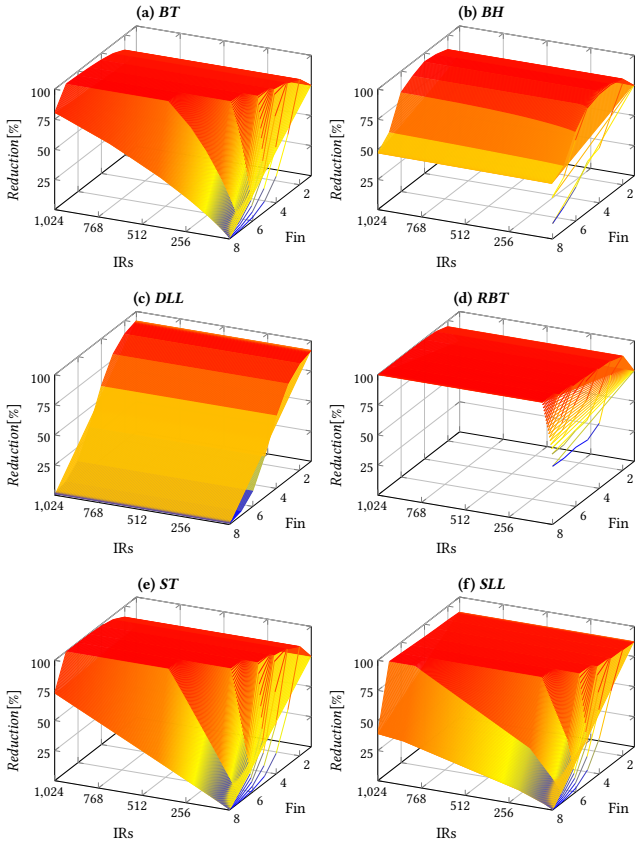

Figure 6: *Reduction* achieved by MKorat for different number of invalid ranges (*IRs*) and *finitization* sizes (*Fin*).

number of invalid ranges. For *RBT*, the last 3 columns ($m \geq 64$) achieve the $Reduction_{max}$ which is 99.98%.

MKorat removes the m-largest invalid ranges independently of the number of equidistant candidates. Hence, in terms of the achieved *Reduction*, MKorat is agnostic to the number of equidistant candidates, which we empirically validated on all our 6 subjects, using various number of invalid ranges and equi-distant candidates.

Q1. Can MKorat achieve $Reduction_{max}$?

> MKorat implementation can provide the $Reduction_{max}$ (introduced in Section 3.4), by removing the m-largest invalid ranges, given m >= total number of invalid ranges explored.

The 3D plots in Figure 6 show how the number of invalid ranges and *finitization* can affect reduction achieved by MKorat for all 6 subjects. Table 3 (discussed earlier) is basically an snapshot of Figure 6 for *Fin* = 8. By definition (Section 3.4), higher *Reduction* can be achieved for subjects with smaller ratio of valid to explored candidates. As shown in Table 2, *RBT* has the smallest ratio of valid to explored candidates for each *finitization*. Figure 6 shows that higher *Reduction* is achieved for *RBT* compared to the other subjects and this reduction is reached for smaller values of invalid ranges maintained by MKorat. Unlike, *RBT*, *DLL* has the highest ratio of valid to explored candidates and the reduction reached for this subject is smaller than others. This observation is noticable in Table 3 as well for these two subjects.

Workers	IRs [%]	Subject(*Fin*)																			Average		
		BT(12)			*BH(9)*			*DLL(11)*			*RBT(10)*			*ST(9)*			*SLL(11)*					
		min	max	sum	min	max	sum	min	max	sum	min	max	sum	min	max	sum	min	max	sum	min	max	sum
1	0		12.43			14.65			6.04			9.96			15.81			7.73			11.10	
	4		11.42			11.81			6.04			4.75			15.19			7.25			9.41	
	8		10.45			11.67			6.09			4.51			14.50			6.75			8.99	
	16		9.08			11.80			6.05			4.23			13.66			5.81			8.44	
	32		6.81			11.70			6.00			3.53			11.09			4.90			7.34	
2	0	6.10	6.42	12.52	6.64	7.79	14.44	3.14	3.15	6.28	4.46	6.30	10.76	7.90	8.36	16.26	3.93	3.94	7.88	5.36	5.99	11.36
	4	5.67	5.95	11.62	4.45	7.46	11.92	3.15	3.15	6.30	0.01	4.79	4.80	7.49	7.97	15.46	3.64	3.77	7.42	4.07	5.51	9.59
	8	5.40	5.61	11.02	4.48	7.53	12.02	3.13	3.15	6.28	0.01	4.52	4.52	7.13	7.57	14.70	3.20	3.80	7.00	3.89	5.36	9.26
	16	4.58	4.84	9.42	4.44	7.48	11.92	3.15	3.16	6.32	0.01	4.19	4.20	6.69	7.01	13.70	2.90	3.14	6.04	3.63	4.97	8.60
	32	3.42	3.51	6.92	4.46	7.46	11.92	3.13	3.13	6.26	0.01	3.53	3.54	5.46	5.81	11.26	2.07	3.02	5.08	3.09	4.41	7.50
8	0	1.70	1.80	13.84	1.37	2.17	15.60	0.87	0.88	7.04	1.28	2.10	12.80	2.16	2.26	17.68	1.11	1.13	8.96	1.42	1.72	12.65
	4	1.56	1.72	12.96	0.06	2.16	13.04	0.87	0.89	7.04	0.01	2.07	5.75	2.02	2.19	17.04	0.97	1.08	8.40	0.91	1.68	10.71
	8	1.44	1.58	12.00	0.06	2.16	13.04	0.87	0.88	7.04	0.01	1.98	5.55	1.90	2.13	16.40	0.79	1.04	7.52	0.84	1.63	10.26
	16	1.21	1.44	10.40	0.06	2.16	12.96	0.87	0.89	7.04	0.01	1.82	5.20	1.77	1.97	15.28	0.73	0.89	6.16	0.78	1.53	9.51
	32	0.86	0.97	7.44	0.06	2.17	13.04	0.87	0.89	7.04	0.01	1.59	4.35	1.53	1.65	12.88	0.45	0.94	6.16	0.63	1.37	8.48
32	0	0.55	0.58	17.92	0.39	0.61	17.28	0.26	0.26	8.32	0.46	0.87	20.16	0.71	0.75	23.68	0.32	0.33	10.56	0.45	0.57	16.32
	4	0.50	0.54	16.64	0.01	0.60	14.40	0.26	0.26	8.32	0.01	0.87	9.52	0.64	0.75	22.72	0.26	0.34	10.56	0.28	0.56	13.69
	8	0.43	0.51	15.36	0.01	0.61	14.08	0.26	0.27	8.32	0.01	0.87	9.38	0.60	0.72	21.76	0.27	0.36	10.24	0.26	0.56	13.19
	16	0.38	0.47	13.76	0.01	0.60	14.40	0.26	0.27	8.32	0.01	0.83	8.82	0.57	0.68	20.16	0.20	0.27	7.04	0.24	0.52	12.08
	32	0.26	0.32	9.60	0.01	0.60	14.40	0.26	0.27	8.32	0.01	0.78	7.28	0.43	0.57	16.64	0.07	0.27	6.72	0.17	0.47	10.49

Table 4: The *min*, *max*, and *total* execution time (in *sec*) for (1) MKorat used in sequential settings (1 worker) compared to Korat and (2) MKorat used in distributed settings (2, 8, and 32 workers) compared to SEQ-ON. For each subject the user provided upper bound on number of invalid ranges (IRs), m in Figure 4, is a percent of number of valid candidates for that subject(*Fin*).

Subject(*Fin*)	SEQ-ON Explored	Invalid Ranges [%]			
		4	8	16	32
BT(12)	12284830	8.26	14.94	26.56	46.10
BH(9)	11778107	25.74	25.74	25.74	25.74
DLL(11)	3535294	0.01	0.01	0.01	0.01
RBT(10)	7530712	65.88	67.40	70.33	76.12
ST(9)	20086300	4.31	8.41	16.52	32.58
SLL(11)	10639556	9.30	15.67	28.43	41.58

Table 5: Total number of candidates explored by SEQ-ON and the percent of those candidates MKorat prunes.

Q2. How does the number and distribution of valid candidate vectors affect MKorat reduction?

> MKorat achieves higher *Reduction* on subjects where the majority of explored candidates are invalid. As an extreme case, a `repOK` method returning a constant, can achieve 0% or 100% reduction for the constant values *true* and *false* respectively (Validated on two constant returning `repOK`s).

Table 4 shows the *minimum*, *maximum*, and *total* execution time (in seconds) for 4 different number of workers. Specifically, 1 worker is used for the sequential setting comparing Korat with MKorat, while 2, 8, and 32 workers indicate a parallel setting using SEQ-ON and MKorat techniques). For each subject, we chose the largest *finitization* for which a sequential execution of Korat terminated within 30 seconds. Further, the m number of invalid ranges MKorat maintains is provided based on a percent of number of valid candidates explored by Korat for a given subject and *Fin*. Node that the number of valid candidates found for a given subject is an upper bound on the number of invalid ranges that subject may have (Section 3.4).

Our results show that for both sequential and parallel settings, MKorat can speedup the test generation. For example, the first two rows in Table 4 show that given 1 worker, when only 4% of test input pairs are maintained, the re-generation of test inputs becomes 18% faster on average (9.41 *sec* compared to 11.10 *sec*). Further, as the percent of user-provided invalid ranges increase to 32%, the re-execution becomes 51% faster on average (7.34 *sec* as opposed to 11.10 *sec*).

Recall from Figure 6 and Table 3 that the increase in number of invalid ranges m maintained by MKorat, affect the *Reduction* differently for different subjects. Table 4 validates the same trend of growth we expected for each subject. For instance, *DLL* had the lowest growth of *Reduction* among other subjects (due to having only two small *head* and *tail* invalid ranges); the 3 columns for *DLL* in Table 4 show that the *Reduction* achieved for this subject is almost agnostic to the increase of m invalid ranges. *RBT*, on the contrary, had the highest rise in *Reduction* as m increased (Figure 6); this outperformance is also evident from Table 4. The effect of increasing m on speedup for the other 4 subjects, is smaller than *RBT* and larger than *DLL*, which is in alignment with the trend observed earlier.

Besides saving in execution time, MKorat can save on the number of workers required. Specifically, if each candidate of an equidistant range e belongs to an invalid range that MKorat maintains, then no worker would be assigned to re-explore range e. We observed this case for *RBT* with 8 and 32 workers, when 4% (or above) test input pairs are maintained, MKorat requires 5 (37.5% fewer

machines) and 14 workers (56.25% savings) respectively. The effect of using fewer machines is reflected on the total execution time for these cases (highlighted in blue in Table 4).

Table 5 shows the number of candidates explored by SEQ-ON and the percent of those candidates MKorat explored for the same subjects, *finitizations*, and invalid ranges reported in Table 4. As shown, *RBT* achieved the highest *Reduction* (76.12%) across all subjects for any percent of invalid ranges used, which explains why MKorat performed better on *RBT* (execution times shown in Table 4). This observation was also expected based on the *Reduction* achieved for *RBT* in Figure 6. Further, *BH* and *DLL* subjects achieve their $Reduction_{max}$ (25.74% and 0.01% respectively) for all 4 values of invalid ranges used in our study. For these two subjects, we observed that 4% of the number of valid candidates is a larger value than their total number of invalid ranges maintained by MKorat. Therefore, providing any percent of invlaid ranges larger than 4% (of valid candidates) is not expected to affect the *Reduction* or execution time of *BH* and *DLL*. This observation justifies why given the number of workers, the *min*, *max*, and *total* execution time of MKorat for *BH* and *DLL* (in Table 4) stays (almost) the same.

Q3. What are the practical benefits of MKorat in terms of execution time and required computational resources for sequential and parallel settings?

> MKorat speeds up the *minimum*, *maximum*, and *total* worker execution time by up to 2.82X in the sequential setting (1 worker) and up to 446X, 1.86X, and 3.04X for the distributed setting with up to 32 workers. In the distributed setting, for subjects with small ratio of valid to explored candidates, like *RBT*, MKorat can provide up to 56.25% savings in number of physical workers required for re-exploration.

Execution platform: We obtained all data on a dedicated cluster in which each node has 16-core 2.7 GHz Intel Xeon CPU E5-2680 with 32GB of RAM, running CentOS release 6.8 (Final). We used Oracle Java: 1.8.0_25. Each MKorat_{exc} was run on a single physical node (on a separate JVM) provided by -Xms2g -Xmx3g command line options.

4.3 Threats to Validity

External: The subjects used in our study may not be representative. To mitigate this threat, we considered 6 subjects shipped with Korat source code that vary in code size, complexity of `repOK`, number of explored and valid candidates. Some of these projects have also been used in prior studies on Korat. Our results may vary for different *finitizations* and number of invalid ranges maintained. Exploring all the combinations was not feasible. To mitigate this threat, we considered several combinations to show the existing relation between different values of parameters. Further, the finitizations and number of equidistant candidates considered in our study is on a par with prior work [3, 26, 29].

Internal: Korat, implementation of MKorat, and our automation scripts may contain bugs that can impact our conclusions. We are mostly confident in the correctness of Korat, as it is a robust tool used in several prior studies. To increase the confidence in our scripts, we developed core parts of our technique twice following (1) an efficient approach, and (2) a less efficient technique (in terms of execution time) and observed that their functional behavior remained the same across two versions. Further, we placed assertions at certain points in our scripts to perform some sanity checks. For instance, the script which distributed subranges (obtained by MKorat) on workers for re-exploration (in Table 4), ensured that per each MKorat execution, the total number of valid instances found across all workers, is equal to the ones found using a sequential execution of Korat. Further, to increase the confidence in our scripts, we reviewed our code, tested it on a number of subjects manually, and inspected several results. To reduce noise and get more consistent numbers for Table 4 (which contain execution times), we measured the values several times and reported the average.

Construct: To find equi-distant candidates (Table 4) we used $m = 2048$ as *maximum* number of workers maintained by SEQ-ON algorithm [26]. We chose this large enough value to form evenly distributed subranges for distribution among up to 32 workers used in our study. For each subject in Table 4, we considered different number of invalid ranges (equal to a percent of number of valid test inputs found for that specific subject), to have a more meaningful and fair comparison between the execution times among different subjects. For Figure 6, we considered a wide range of finitizations and invalid ranges to observe the unique trend in *Reduction* increase for each subject.

5 RELATED WORK

This chapter presents related work on parallel analysis for systematic testing. Specifically, we consider two approaches for testing sequential programs, including one black-box testing technique, namely Korat [3], and one white-box testing technique, namely *symbolic execution* [22], and one approach, namely *model checking* [6], for testing multi-threaded programs. Moreover, we consider incremental analysis techniques that re-use results from previous runs.

5.1 Parallel Korat

The idea of parallel test generation and execution in the context of Korat was introduced by Misailovic et al. [26]. The idea of invalid ranges is rooted in their discussion on potential optimizations [26] where they observe the potential usefulness of creating sub-ranges that start and end at valid candidates. Our technique, MKorat, builds on this observation.

PKorat [29] introduced a different approach for parallel test generation using Korat. The key idea in PKorat is to explore Korat's non-deterministic field assignments in parallel. Thus, PKorat does not require a previous execution of the Korat search but can still explore the space of candidate structures in parallel. However, re-running PKorat in the *online* test generation setting does not utilize any information about any previous execution of Korat; specifically, re-running PKorat does not utilize invalid ranges and re-explores all candidates that sequential Korat explores by default. While the original PKorat technique was defined for execution for a cluster of traditional computing platforms, recent work specialized PKorat for modern GPU's [27]. Our approach is orthogonal to PKorat and can be integrated with PKorat. For example, PKorat can be used to explore each range that our approach creates based on the first execution of Korat search.

5.2 Parallel symbolic execution

ParSym [30] applies the PKorat approach to symbolic execution – a classic program analysis based on systematic exploration of the program's bounded execution paths. *Simple Static Partitioning* [33] for parallel symbolic execution first performs a *shallow* depth execution to build a set of preconditions based on the number of available workers who perform *deeper* exploration with respect to their individual preconditions. The parallel symbolic execution tool *Cloud9* [4] embodies a production quality infrastructure based on load balancing.

Ranged symbolic execution [31] defines *ranges* for symbolic execution and uses them for distributing the symbolic exploration of bounded execution paths; each range is defined by a pair of *in-order* concrete inputs where the first input represents the path where symbolic execution starts and the second input represents the path where symbolic execution ends; moreover, work stealing is used for dynamic load balancing.

Most recent work by Qiu [28] introduces the idea of *feasible* ranges for succinctly memoizing symbolic execution results where the path conditions for all paths in a feasible range are satisfiable. Our idea of invalid ranges for Korat is inspired by Qiu's idea of feasible ranges for symbolic execution and complements it. We could extend our work and support feasible ranges for Korat, so the cost of running it to generate valid inputs in a feasible range is reduced; for example, any candidate within a feasible range is known to be valid and therefore its validity does not need to be checked again; however, `repOK` may still need to be partially (and in some cases fully) executed on it to determine what the next candidate (which is also known to be feasible) is. Likewise, we could introduce the use of invalid ranges in symbolic execution.

5.3 Parallel model checking

Funes et al. [12] introduced the idea of ranging for software model checking using *Java PathFinder (JPF)* [37], an explicit state model checker; specifically, the exploration by the model checker is ranged by a pair of *in-order* paths that define the start and end of the model checking run. Previous work on parallel randomized state space search used multiple randomly generated start configurations for JPF and ran them in parallel with the expectation that one of them would find an erroneous state faster than the sequential run of the model checker [9]. One of the earliest techniques for parallel search for explicit state checking was parallel *Murϕ*, introduced by Stern and Dill [34], and shown to provide approximately linear speedups. Swarm verification [19] shows how to leverage multi-core computation platforms in the context of the SPIN model checker [18].

5.4 Incremental analysis

A number of incremental analyses re-use results from previous runs to optimize subsequent runs, e.g., for test generation [17, 35], symbolic execution [15, 36, 39], and model checking [2, 23, 32, 38]. The key difference between our approach and previous work is to re-use state-space exploration results, specifically about consecutive invalid candidates, to optimize constraint solving. Our approach shares the spirit of incremental SAT and conflict-driven clause learning [10] but works at a very different level (Java predicates versus CNF formulas).

6 CONCLUSION

This paper introduced a novel approach to reduce the cost of systematic testing using the Korat approach in certain application scenarios. Our key insight is that sometimes Korat's backtracking search is repeated over the *same* state space across *separate* runs of Korat, and an earlier run of the search can be summarized to more efficiently perform a later run. We introduced the idea of *invalid ranges* which succinctly encode parts of the exploration space, which do not contain any valid inputs but have to be explicitly explored by the Korat search since it is unable to prune them. Our approach directly prunes these parts in a future run of Korat over the same input space. We developed our approach for two settings: a sequential setting where the Korat search is run using one worker (i.e., processing unit), and a parallel setting where the Korat search is distributed to several workers. In the parallel setting, we build on a previous technique for parallel Korat, namely *SEQ-ON*, and integrate invalid ranges with it. An experimental evaluation using a suite of standard subjects shows the efficacy of our approach.

Acknowledgments. 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-0845628 and CNS-1239498.

REFERENCES

[1] Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee White. 2000. State generation and automated class testing. *Software Testing, Verification and Reliability* 10, 3 (2000), 149–170.

[2] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. 2012. Conditional Model Checking: A Technique to Pass Information Between Verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, 57:1–57:11.

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. In *ISSTA*. ACM, New York, NY, USA, 123–133.

[4] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys)*. 183–198.

[5] Cristian Cadar and Dawson R. Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *12th International Conference on Model Checking Software (SPIN'05)*. Springer-Verlag, Berlin, Heidelberg, 2–23.

[6] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Cambridge, MA, USA.

[8] Nima Dini. 2016. MKorat: A Novel Approach for Memoizing the Korat Search and Some Potential Applications. (2016).

[9] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. 2007. Parallel Randomized State-Space Search. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, Vol. 37. IEEE Computer Society, Washington, DC, USA, 3–12.

[10] Niklas Een and Niklas Sorensson. 2003. An Extensible SAT-solver. In *6th Conference on Theory and Applications of Satisfiability Testing (SAT)*. Santa Margherita Ligure, Italy.

[11] Antonio Filieri, Marcelo F. Frias, Corina S. PăSăReanu, and Willem Visser. 2015. Model Counting for Complex Data Structures. In *Proceedings of the 22nd International Symposium on Model Checking Software (SPIN 2015)*, Vol. 9232. Springer-Verlag New York, Inc., New York, NY, USA, 222–241.

[12] Diego Funes, Junaid Haroon Siddiqui, and Sarfraz Khurshid. 2012. Ranged Model Checking. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

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

In *32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 225–234.

[14] Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium*. ACM, New York, NY, USA, 174–186.

[15] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 47–54.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223.

[17] M.J. Harrold and M.L. Souffa. 1988. An incremental approach to unit testing during maintenance. In *Conference on Software Maintenance*.

[18] Gerald Holzmann. 1997. The Model Checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (May 1997).

[19] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. 2011. Swarm Verification Techniques. *IEEE Trans. Softw. Eng.* 37, 6 (Nov 2011), 845–857.

[20] Daniel Jackson and Mandana Vaziri. 2000. Finding Bugs with a Constraint Solver. In *International Symposium oAn Software Testing and Analysis (ISSTA)*.

[21] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568.

[22] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[23] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. 2008. Incremental state-space exploration for programs with dynamically allocated data. In *30th international conference on Software engineering*.

[24] Barbara Liskov and John Guttag. 2000. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[25] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *16th IEEE International Conference on Automated Software Engineering (ASE '01)*. IEEE Computer Society, Washington, DC, USA, 22–31.

[26] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. 2007. Parallel test generation and execution with Korat. In *6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, New York, NY, USA, 135–144.

[27] Omitted. 2017. Due to double-blind reviewing. (2017).

[28] Rui Qiu. 2016. *Scaling and Certifying Symbolic Execution*. Ph.D. Dissertation. University of Texas at Austin.

[29] Junaid Haroon Siddiqui and Sarfraz Khurshid. 2009. PKorat: Parallel Generation of Structurally Complex Test Inputs. In *Second International Conference on Software Testing Verification and Validation, ICST*. IEEE, Denver, CO, USA, 250–259.

[30] J. H. Siddiqui and S. Khurshid. 2010. ParSym: Parallel symbolic execution. In *2nd International Conference on Software Technology and Engineering*. V1–405–V1–409.

[31] Junaid Haroon Siddiqui and Sarfraz Khurshid. 2012. Scaling Symbolic Execution Using Ranged Analysis. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, part of SPLASH*. ACM, New York, NY, USA, 523–536.

[32] O. Sokolsky and S. A. Smolka. 1994. Incremental model checking in the modal mu-calculus. In *International Conference on Computer Aided Verification*.

[33] Matt Staats and Corina Păsăreanu. 2010. Parallel Symbolic Execution for Structural Test Generation. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 183–194.

[34] Ulrich Stern and David L. Dill. 1997. Parallelizing the Murphi Verifier. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*. Springer-Verlag, London, UK, UK, 256–278.

[35] Engin Uzuncaova. 2008. *Efficient Specification-based Testing Using Incremental Techniques*. Ph.D. Dissertation. University of Texas at Austin.

[36] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Rusing and Recycling Constraints in Program Analysis. In *ESEC/FSE*.

[37] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. 2000. Model Checking Programs. In *The Fifteenth IEEE International Conference on Automated Software Engineering, ASE,Grenoble, France (ASE '00)*. IEEE Computer Society, Washington, DC, USA, 3–12.

[38] G. Yang, M. B. Dwyer, and G. Rothermel. 2009. Regression model checking. In *ICSM*. 115–124.

[39] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized Symbolic Execution. In *ISSTA*. 144–154.