The Thesis Committee for Nima Dini
certifies that this is the approved version of the following thesis:

MKorat: A Novel Approach for Memoizing the Korat Search and Some Potential Applications

APPROVED BY

SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Milos Gligoric

MKorat: A Novel Approach for Memoizing the Korat Search and Some Potential Applications

by

Nima Dini, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2016

Dedicated to my beloved parents, Gita Mostafizi and Mohammand Dini, and my supportive brother Navid.

Acknowledgments

I wish to express my sincere gratitude to my supervisor Dr. Sarfraz Khurshid, for his continuous support and encouragement in drafting this thesis, for his patience, motivation, and immense knowledge. His guidance helped me all the time in graduate school. Further, I would like to thank Dr. Milos Gligoric for taking the time to review this work, and providing helpful feedback.

I would like to thank The University of Texas at Austin Cockrell School of Engineering and the Cockrell Foundation for their encouragement and financial support. In addition, this work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-0845628 and CNS-1239498).

MKorat: A Novel Approach for Memoizing the Korat Search and Some Potential Applications

Nima Dini, M.S.E
The University of Texas at Austin, 2016

Supervisor: Sarfraz Khurshid

Writing logical constraints that describe properties of desired inputs enables an effective approach for systematic software testing, which can find many bugs. The key problem in systematic constraint-based testing is efficiently exploring very large spaces of all possible inputs to enumerate desired valid inputs. The *Korat* technique provides an effective solution to this problem. Korat uses desired input properties written as imperative predicates and implements a backtracking search that prunes large parts of the input space and enumerates all non-isomorphic inputs within a given bound on input size. Despite the effectiveness of Korat's pruning, systematically creating and running large numbers of tests can be costly in practice. Previous work introduced parallel test generation and execution using Korat to make it more practical. We build on a specific algorithm, *SEQ-ON*, introduced in previous work for *equi-distancing* candidate inputs, which allows re-execution of Korat for input

generation using parallel workers with evenly distributed workload. Our key insight is that the Korat search typically encounters many *consecutive* candidates that are all invalid inputs and such *invalid ranges* of candidates can be *memoized* succinctly to optimize re-execution of Korat. We introduce a novel approach for memoizing Korat's checking of consecutive invalid candidates, embody the approach into three new techniques based on SEQ-ON, evaluate the techniques using a standard suite of data structure subjects to show the efficacy of our approach, and show some potential applications of it in two new application domains for Korat. We believe our work opens a promising new direction to optimize solving of imperative constraints and using them in novel application domains.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Systematic software testing (aka bounded exhaustive testing), i.e., testing against all inputs within a given bound on input size, is an effective methodology for finding many bugs in programs [1, 2, 10–12, 15, 19]. 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 [1, 19]. A number of different techniques embody this approach and support constraints written in different languages, including a variety of declarative languages [19] and imperative languages [1].

Our work focuses on the constraints written as *imperative predicates*, 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 [1, 20]. Given a predicate, termed *repOK* [18], 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.

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 [1]. 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. To mitigate these factors, previous work by Misailovic et al. introduced the idea of parallel test generation and execution using Korat [20]. 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 pruning many candidates from the search. Thus, evenly distributing the test generation workload among parallel Korat workers is challenging.

One technique that Misailovic et al. introduce, named *SEQ-ON*, to mitigate the two limiting factors, addresses 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 property of the SEQ-ON algorithm is that it uses the first execution of Korat for input generation to create *equidistant* candidates, 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.

Our thesis is that Korat's exploration often encounters many *consecutive* invalid candidates, which form *invalid ranges* that can be *memoized* [4] succinctly (using just two candidates as end-points) during the first execution of Korat for input generation, and re-used 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 introduce a novel approach for memoizing Korat's checking of consecutive invalid candidates, embody the approach into three new techniques that build on SEQ-ON, evaluate the techniques using a standard suite of subjects to show the efficacy of our approach.

We make the following contributions:

- **Infeasible ranges.** We introduce the idea of infeasible ranges, which succinctly represent consecutive invalid candidates that the standard Korat search is unable to prune and must explicitly check using repOK.

- **Memoization approach.** We introduce a novel approach for re-using results from one execution of Korat in subsequent executions of Korat for input generation using fixed repOK and finitization when all inputs generated by Korat cannot be stored for re-use later.

- **Techniques.** We introduce three techniques that embody our approach and build on the SEQ-ON algorithm from previous work [20] for more efficient input generation.

- **Evaluation.** We use a suite of standard subjects to evaluate our approach. Evaluation results show that our approach provides substantial reduction in the number of candidates to consider.

- **Abstract search problem.** We introduce an abstract search problem, named *online next-valid-neighbor*, which maps any given invalid input structure to the first valid structure that Korat finds after inspecting the input. We provide a solution to this problem and apply it in two novel application domains.

- **Constraint-based game play.** We introduce the idea of using logical constraints written as imperative predicates to define *rules* for simple 2-player games, apply Korat for automated game play, and use our memoization approach to optimize our solution for *repeated* game play.

- **Constraint-based data structure repair.** We apply Korat for repair of erroneous program states (specifically, data structures) with respect

to given expected structural constraints that are written as imperative predicates, and use our memoization approach to provide an efficient solution for *repeated* data structure repair.

Chapter 2

Example

In this chapter, we discuss the motivation of our technique by a simple example taken from Korat project's source code[1]. The Java code in Figure 2.1, defines a red-black tree and specifies its repOK method.

```
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
7    public static class Node  {
8      int key;
9      int value;
10     Node left = null;
11     Node right = null;
12     Node parent;
13     int color = BLACK;
14   }
15
16   public boolean repOK() { ... }
17 }
```

Figure 2.1: *RedBlackTree* example.

Figure 2.2 shows the finitization description that Korat search algorithm requires to generate red-black trees of a given size. For instance, given

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

```
1  public static IFinitization finRedBlackTree(int numEntries,
2      int minSize, int maxSize, int numKeys) {
3    IFinitization f = FinitizationFactory.create(
4                      RedBlackTree.class);
5    IClassDomain entryDomain =
6              f.createClassDomain(Node.class, numEntries);
7    IObjSet entries = f.createObjSet(Node.class, true);
8    entries.addClassDomain(entryDomain);
9
10   IIntSet sizes = f.createIntSet(minSize, maxSize);
11   IIntSet keys = f.createIntSet(-1, numKeys - 1);
12   IIntSet values = f.createIntSet(0);
13   IIntSet colors = f.createIntSet(0, 1);
14
15   f.set("root", entries);
16   f.set("size", sizes);
17   f.set("Node.left", entries);
18   f.set("Node.right", entries);
19   f.set("Node.parent", entries);
20   f.set("Node.color", colors);
21   f.set("Node.key", keys);
22   f.set("Node.value", values);
23
24   return f;
25 }
```

Figure 2.2: Finitization description for the *RedBlackTree* example

finRedBlackTree(4, 4, 4, 4), Korat automatically generates all non-isomorphic red-black trees with exactly 4 nodes. The total number of candidate vectors explored is 961, out of which only 8 satisfy the repOK method, which means there are 8 valid red-black tree structures containing 4 nodes. Figure 2.3 shows all such structures[2].

[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.

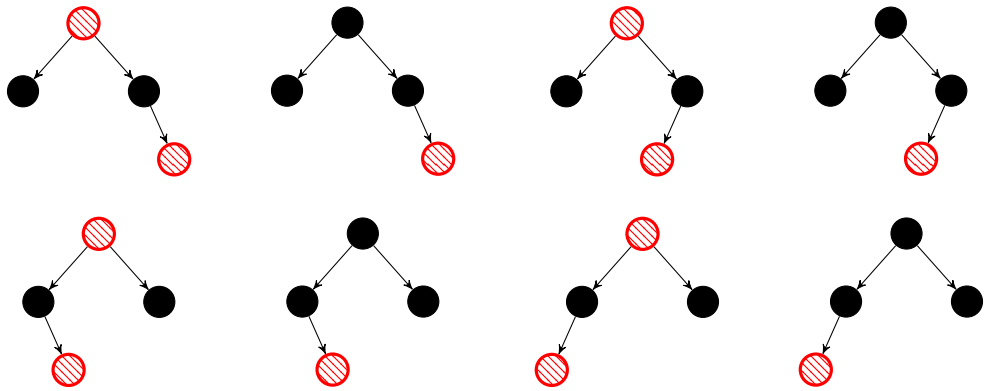

Figure 2.3: Valid red-black trees with 4 nodes

Given *--printCandVects* as the command-line option[3], Korat is able to print all candidate vectors explored. For instance, Figure 2.4 shows all explored candidates for *finRedBlackTree(2, 2, 2, 2)*.

In this example, based on the attributes in the *RedBlackTree* Java class provided, the layout of a candidate vector is an array of 14 integers that represent values of the following fields in order: *T0.root*, *T0.size*, *N0.left*, *N0.right*, *N0.parent*, *N0.color*, *N0.key*, *N0.value*, *N1.left*, *N1.right*, *N1.parent*, *N1.color*, *N1.key*, and *N1.value*. Korat starts the search from a candidate vector containing all zeros (0,0,0,0,0,0,0,0,0,0,0,0,0,0) and runs the user-provided repOK method on it. During each repOK execution, Korat monitors the fields being accessed and uses it to form the next candidate that needs to be explored. For the initial candidate vector in this example, only the first and the second fields are being accessed during repOK execution. Korat backtracks on the last accessed field, which is the second field and selects the next possible value

[3]list of other options can be found here: http://korat.sourceforge.net/manual.html

	Candidate vector	:: Index of fields accessed in repOK	
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	:: 0 1	
2	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0	:: 0 4 2 3 1	
3	1 0 0 1 0 0 0 0 0 0 0 0 0 0 0	:: 0 4 2 3	
4	1 0 0 2 0 0 0 0 0 0 0 0 0 0 0	:: 0 4 2 3 10	
5	1 0 0 2 0 0 0 0 0 0 1 0 0 0 0	:: 0 4 2 3 10 8 9 1 5 11	(1)
6	1 0 0 2 0 0 0 0 0 0 1 1 0 0 0	:: 0 4 2 3 10 8 9 1 5 11	
7	1 0 0 2 0 1 0 0 0 0 1 0 0 0 0	:: 0 4 2 3 10 8 9 1 5 11 6	
8	1 0 0 2 0 1 1 0 0 0 1 0 0 0 0	:: 0 4 2 3 10 8 9 1 5 11 6 12	
9	1 0 0 2 0 1 1 0 0 0 1 0 1 0 0	:: 0 4 2 3 10 8 9 1 5 11 6 12	
10	1 0 0 2 0 1 1 0 0 0 1 0 2 0 0	:: 0 4 2 3 10 8 9 1 5 11 6 12 ***	
11	1 0 0 2 0 1 2 0 0 0 1 0 0 0 0	:: 0 4 2 3 10 8 9 1 5 11 6 12	
12	1 0 0 2 0 1 2 0 0 0 1 0 1 0 0	:: 0 4 2 3 10 8 9 1 5 11 6 12	
13	1 0 0 2 0 1 2 0 0 0 1 0 2 0 0	:: 0 4 2 3 10 8 9 1 5 11 6 12	
14	1 0 0 2 0 1 0 0 0 0 1 1 0 0 0	:: 0 4 2 3 10 8 9 1 5 11	
15	1 0 0 2 0 0 0 0 0 1 1 0 0 0 0	:: 0 4 2 3 10 8 9	
16	1 0 0 2 0 0 0 0 0 2 1 0 0 0 0	:: 0 4 2 3 10 8 9	
17	1 0 0 2 0 0 0 0 1 0 1 0 0 0 0	:: 0 4 2 3 10 8	
18	1 0 0 2 0 0 0 0 2 0 1 0 0 0 0	:: 0 4 2 3 10 8	
19	1 0 0 2 0 0 0 0 0 0 2 0 0 0 0	:: 0 4 2 3 10	(2)
20	1 0 1 0 0 0 0 0 0 0 0 0 0 0 0	:: 0 4 2	
21	1 0 2 0 0 0 0 0 0 0 0 0 0 0 0	:: 0 4 2 10	
22	1 0 2 0 0 0 0 0 0 0 1 0 0 0 0	:: 0 4 2 10 3 8 9 1 5 11	
23	1 0 2 0 0 0 0 0 0 0 1 1 0 0 0	:: 0 4 2 10 3 8 9 1 5 11	
24	1 0 2 0 0 1 0 0 0 0 1 0 0 0 0	:: 0 4 2 10 3 8 9 1 5 11 6	
25	1 0 2 0 0 1 1 0 0 0 1 0 0 0 0	:: 0 4 2 10 3 8 9 1 5 11 6 12	
26	1 0 2 0 0 1 1 0 0 0 1 0 1 0 0	:: 0 4 2 10 3 8 9 1 5 11 6 12	
27	1 0 2 0 0 1 1 0 0 0 1 0 2 0 0	:: 0 4 2 10 3 8 9 1 5 11 6 12	
28	1 0 2 0 0 1 2 0 0 0 1 0 0 0 0	:: 0 4 2 10 3 8 9 1 5 11 6 12	
29	1 0 2 0 0 1 2 0 0 0 1 0 1 0 0	:: 0 4 2 10 3 8 9 1 5 11 6 12 ***	
30	1 0 2 0 0 1 2 0 0 0 1 0 2 0 0	:: 0 4 2 10 3 8 9 1 5 11 6 12	
31	1 0 2 0 0 1 0 0 0 0 1 1 0 0 0	:: 0 4 2 10 3 8 9 1 5 11	
32	1 0 2 0 0 0 0 0 0 1 1 0 0 0 0	:: 0 4 2 10 3 8 9	
33	1 0 2 0 0 0 0 0 0 2 1 0 0 0 0	:: 0 4 2 10 3 8 9	
34	1 0 2 0 0 0 0 0 1 0 1 0 0 0 0	:: 0 4 2 10 3 8	
35	1 0 2 0 0 0 0 0 2 0 1 0 0 0 0	:: 0 4 2 10 3 8	(3)
36	1 0 2 1 0 0 0 0 0 0 1 0 0 0 0	:: 0 4 2 10 3	
37	1 0 2 2 0 0 0 0 0 0 1 0 0 0 0	:: 0 4 2 10 3	
38	1 0 2 0 0 0 0 0 0 0 2 0 0 0 0	:: 0 4 2 10	
39	1 0 0 0 1 0 0 0 0 0 0 0 0 0 0	:: 0 4	
40	1 0 0 0 2 0 0 0 0 0 0 0 0 0 0	:: 0 4	

Figure 2.4: Candidates explored for *finRedBlackTree(2, 2, 2, 2)*

(which is 1) for the first field and the search continues. Overall, the explored space contains 40 candidates; 2 of them are valid (marked with *** in Figure 2.4), and they separate the space into 3 infeasible ranges: [1, 10), (10, 29), and (29, 40], labeled as ①, ②, and ③ respectively in Figure 2.4.

Korat supports *ranging* the search, i.e., bounding it to explore a subset of candidates [20]. 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. Prior work [20] presented *SEQ-ON* equi-distancing algorithm, which aims to distribute Korat execution for future runs of the same search. The technique only keeps a bounded number of equidistant candidate vectors during the first sequential run, which enables the distribution of the explored space among workers in a load-balanced fashion.

For instance, in the red-black tree example with 4 workers, the equi-distancing algorithm splits the explored domain into 4 partitions, each with the same approximate size of 256 candidate vectors[4]. As shown in Figure 2.5, 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.

[4]In this example, the last range has 961-768=193 elements.

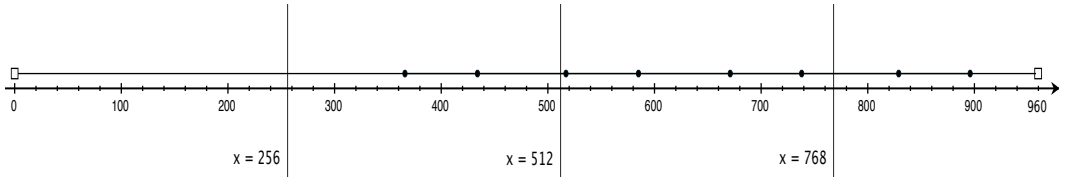

Figure 2.5: Candidates explored for *finRedBlackTree(4, 4, 4, 4)*.

The main pitfall with the SEQ-ON algorithm is that while it parallelizes the executions, the overall number of explored candidate vectors across all slaves remains the same as the sequential run. Since re-exploring invalid candidates is redundant, and they will not be generated by Korat, it is desirable to prune them from the search altogether if possible.

We introduce the concept of *infeasible range*, which is a sequence of consecutive invalid candidates considered by Korat search, i.e., repOK method returns $false$ for all these candidates and they appear in consecutive order. Our goal is to reduce exploration of infeasible ranges. We present three techniques to remove a bounded number of such ranges, prior to distribution among workers. Next, we briefly describe each technique and show how it performs for *finRedBlackTree(4, 4, 4, 4)*:

1. Our first technique, MKorat$_{inf}$, keeps track of the $m \geq 1$ largest infeasible ranges, in addition to using SEQ-ON algorithm to find the $k \geq 1$ equidistant candidates. To distribute, it removes an equidistant range if it is a subrange of any infeasible range. The key idea is that no worker needs to take an equidistant range, if it is an infeasible range, since it

contains no valid candidates. In our red-black tree example with 4 workers, MKorat$_{inf}$ results in 44.74% (430/961) *reduction*[5] for any $m \geq 1$, by running these ranges: [366, 512), [512, 768), and [768, 897).

2. Our second technique, MKorat$_{equ}$, calculates the $m \geq 1$ equidistant candidate vectors using SEQ-ON algorithm. In addition, for each equidistant candidate C_i, it finds a pair of valid candidates (V_s, V_t), if any, that (1) $V_s \leq C_i \leq V_t$, based on the exploration order, and (2) no valid candidate is in the range (V_s, V_t). By definition, this range is infeasible, and can be removed safely. Further, note that if C_i is a valid candidate, this range becomes (C_i, C_i), which is an empty range and will not be removed. For our red-black tree example with 4 workers, MKorat$_{equ}$, results in 62.64% (602/961) reduction by selecting these ranges to run: [366, 435), [517, 739), and [829, 897).

3. Our third technique, MKorat, removes the $m \geq 1$ largest infeasible ranges and distributes the remaining tasks among the workers with respect to $k \geq 1$ equidistant candidates maintained by SEQ-ON algorithm. In the red-black tree example with 4 workers, MKorat results in the highest reduction of 71.48% (687/961), for $m = 3$ infeasible ranges, by running the ranges: [366, 435), [517, 586), [671, 739), and [829, 897).

[5]Note that all our three techniques remove the *head* and *tail* infeasible ranges, if any. This optimization and a more formal definition of reduction, will be discussed later in Section 3.2.

Chapter 3

Techniques

In this chapter, we recall the original *SEQ-ON* algorithm [20] (Section 3.1) and present our three techniques MKorat$_{inf}$, MKorat$_{equ}$, and MKorat, which build on SEQ-ON (Section 3.2). Further, we discuss several key properties of our techniques and some implementation details (Section 3.3).

3.1 Equi-distancing for SEQ-ON

The key novelty of the original parallel test generation and execution algorithm using Korat [20], 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 the same number of candidates in any range $[C_i, C_{i+1})$ for $0 \leq i \leq n$, and the union of all ranges, is the entire search space $\bigcup_{i=0}^{n}[C_i, C_{i+1}) = [C_0, C_{n+1}]$, where C_0 and C_{n+1} are the initial and last candidate vector explored in Korat search respectively.

Figure 3.1 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 recursively continues, and at the end, the function returns the candidates to keep for the future parallel executions.

3.2 Techniques

This section presents our three main techniques that build on Korat's SEQ-ON algorithm. We first introduce the concept of an *infeasible range*, i.e., a sequence of consecutive candidate vectors $\langle C_i, C_{i+1}, ..., C_j \rangle$ explored in Korat search such that each C_k, where $i \leq k \leq j$, is invalid based on the user-provided repOK method. The motivation is to speed-up future executions of Korat search, by not running repOK method on known infeasible ranges, because it is known a-priori that the returned repOK result would be false for each candidate. More precisely, removing the infeasible range $\langle C_i, C_{i+1}, ..., C_j \rangle$ results in $j - i + 1$ fewer number of repOK executions.

The original SEQ-ON algorithm requires re-exploration of the entire space of candidate vectors explored by the standard Korat search. We introduce 3 extended versions of this technique that store a bounded number

```
// input: m is the maximum number of workers
// output: an array of equidistant candidates,
// with the array length between m and 2 * m

Candidate[] equiDistantCandidates(int m) {
  Candidate[] candidates = new Candidate[2 * m];
  int distance = 1;
  int index = 0;
  while (Korat.hasNext()) {
    for (int i = 0; i < distance; i++) {
      candidates[index] = Korat.next();
      if (!Korat.hasNext()) break;
    }
    index++;
    if (index == candidates.length) {
      // half the array and double the distance
      for (int j = 0; j < candidates.length / 2; j++)
        candidates[j] = candidates[2 * j + 1];
      distance = distance * 2;
      index = m;
    }
  }

  // resize the output length to valid indexes
  Candidate[] result = new Candidate[index];
  for (int i = 0; i < index; i++)
    result[i] = candidates[i];
  return result;
}
```

Figure 3.1: Equi-distancing for SEQ-ON [20].

of infeasible ranges during the first sequential run, and skip executing them for future runs. As a result, only a subset of the entire space of candidates explored by sequential Korat is re-explored. The three main techniques are built on top of the SEQ-ON algorithm. All three techniques share the same goal, which is removing infeasible ranges prior to distribution for future runs, and all techniques provide the same time and space complexity as the original

technique does. Moreover, each of the three techniques removes the *head* and *tail* infeasible ranges if they exist, where:

- *head infeasible 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. Note in case that C_0 is equal to C_v, the range $[C_0, C_v)$ contains no element, and *head infeasible range* does not exist.

- *tail infeasible 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. If C_w is equal to C_n, the range $(C_w, C_n]$ is empty and *tail infeasible range* does not exist.

3.2.1 MKorat$_{inf}$

This is the simplest technique presented, which stores the $m \geq 1$ largest infeasible ranges during the first sequential run. These infeasible ranges are maintained in addition to and independently of the $k \geq 1$ equidistant candidates that SEQ-ON maintains. For future Korat runs, any equidistant range that is a subrange of an infeasible range will be omitted. The remaining ranges will be distributed among workers.

3.2.2 MKorat$_{equ}$

This technique keeps the $k \geq 1$ equidistant candidate vectors that SEQ-ON maintains. In addition, for each equidistant candidate vector C_i, the technique finds the largest infeasible range $[C_s, C_t)$ containing C_i, if any,

and removes it prior to distribution. Note that MKorat$_{equ}$ does not take a separate argument for infeasible ranges; this number is determined based on the number of equidistant candidates.

3.2.3 MKorat

This technique removes the $m \geq 1$ largest infeasible ranges, if any, and finds the ranges $\langle r_1, r_2, ..., r_s \rangle$ that should be run. Further, the distribution will be with respect to the $k \geq 1$ equidistant candidate vectors maintained in SEQ-ON. Specifically, 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 recursively, until no range in the final collection of ranges $\langle q_1, q_2, ..., q_t \rangle$ contains an equidistant candidate vector.

3.2.4 Properties

Given a Korat search problem (repOK and finitization), we define the following general metrics, which enable us to compare our techniques with the original SEQ-ON algorithm:

- The effectiveness of a technique is defined by *reduction* as follows:

$$reduction = \frac{\#\ of\ invalid\ explored\ candidates\ skipped}{\#\ of\ explored\ candidates}$$

  The denominator of the equation above is the total number of candidates that workers need to explore in the SEQ-ON algorithm. The numerator is the number of invalid candidates our techniques skip for future runs.

Hence, conceptually, reduction is the subset of the explored space that can be pruned for the future runs in the original SEQ-ON algorithm. Further, $reduction_{opt}(EQU, INF)$ is the optimal reduction achievable given the number of equidistant candidates (EQU) and infeasible ranges (INF). $reduction_{max}$ is the maximum achievable reduction, given large enough values for input parameters EQU and INF.

- The average work each worker has to perform is:

$$AVG\ work\ per\ worker = \frac{Total\ length\ of\ the\ intervals\ to\ run}{\#\ of\ equidistant\ candidates}$$

- The maximum work each worker needs to undertake is the maximum length of the intervals selected by a technique for the future runs.

Our techniques have the following properties:

1. Maximum work performed by each worker, is no more than the maximum work in the original SEQ-ON algorithm. Correctness argument follows.

    - $MKorat_{inf}$ removes a range, with two consecutive equidistant candidates as end-points, if this range falls into a known infeasible range; hence, the remaining ranges to run are a subset of the ranges the original SEQ-ON selects, and the size of maximum range in our technique would not be larger than the maximum range selected in SEQ-ON.

- *MKorat*$_{equ}$ removes infeasible ranges surrounding equidistant candidates, and selects the remaining ranges to run, which makes each remaining range to be at most as big as the original range.
- *MKorat* removes the m-largest infeasible ranges from the explored space, and then breaks the remaining ranges with respect to equidistant candidates; so, this property holds here as well.

2. Average work performed by every worker, is at most the same as average work in the original SEQ-ON algorithm. The correctness argument for this property is similar to the Maximum work discussed above.

3. Our techniques work based on removing infeasible ranges for future runs. Specifically, in cases that the number of explored invalid candidates are larger than the number of valid ones, a higher reduction will be achieved. As an extreme case, for the constant returning repOK, i.e, returning *true* or *false*, all techniques achieve *reduction*$_{max}$, which is 0% and %100 respectively.

4. MKorat achieves *reduction*$_{opt}$, as it removes the given m largest infeasible ranges from the explored search space. Further, for large enough values of m, i.e., $m > \#\ of\ infeasible\ ranges$, this technique provides *reduction*$_{max}$.

3.3 Implementation

Figure 3.2 shows the core test generation algorithm of Korat. Within the *while (true)* loop, Korat explores the state space for valid candidates. We first extended integrated this algorithm with the SEQ-ON algorithm. Next, we implemented each technique using this extension. Our design maintains the time and space complexity of SEQ-ON and only requires one sequential run of Korat.

3.3.1 Command-line options

Table 3.1 shows the new run-time options we added to Korat. *--version* allows selecting between the original Korat and our 3 techniques. *--equi* represents the number of equidistant candidates. Note that this number cannot be greater than the number of total explored candidates. Our current implementations consider $min(equi, totalExplored)$ as the number of equidistant candidates, in which equi is the number provided by user and totalExplored is the total candidates explored in a sequential run. Finally, *--infeasible* is the number of infeasible ranges to consider. Similar to the *--equi* option, if this number exceeds the total number of infeasible ranges, the minimum of the two will be selected.

3.3.2 Limitations of parallel execution

Korat, provided with the command-line option *--cvWrite*, writes all explored candidate vectors to a serialized file *f*. Further, Korat has two other

```
void startTestGeneration(IFinitization fin) {
    /* Set up */
    IKoratSearchStrategy stateSpaceExplorer =
                              new StateSpaceExplorer(fin);
    StateSpace stateSpace = ((Finitization)fin).getStateSpace();
    initStartAndEndCVs(stateSpaceExplorer);

    long totalExplored = 0;
    long validCasesGenerated = 0;

    Object testCase = null;
    Class testCaseClass = fin.getFinClass();

    Method pred = getPredicateMethod(testCaseClass);
    IIntList accessedFields =
            stateSpaceExplorer.getAccessedFields();

    /* Main search */
    while (true) {
        testCase = stateSpaceExplorer.nextTestCase();
        if (testCase == null)
            break;

        totalExplored++;
        predicateOK = checkPredicate(testCase, pred);

        if (predicateOK) {
            validCasesGenerated++;
            stateSpaceExplorer.reportCurrentAsValid();
        }
    }
}
```

Figure 3.2: Korat test generation algorithm

options, *--cvStart <num1>* and *--cvEnd <num2>*, which limit the exploration from *<num1>-th* to *<num2>-th* candidate from file *f*. Our techniques generate executable Java commands based on this high level black-box API to be distributed among workers. However, we know that it might be infeasible

Option	Values
--version	0: Default, 1: MKorat$_{inf}$, 2: MKorat$_{equ}$, 3: MKorat
--equi	# of equidistant candidates
--infeasible	# of infeasible ranges

Table 3.1: Korat extended options

to generated tests of a single sequential run to a file. Hence, ideally, our executable Java commands should take actual *start* and *end* candidate vectors instead of indexes to them written in file *f*. While this is not an inherent limitation of our techniques, the current implementations work based on Korat API and require reading the candidate vectors from the file *f*.

Chapter 4

Evaluation

We designed a study to evaluate the effectiveness of our techniques, i.e., MKorat$_{inf}$, MKorat$_{equ}$, and MKorat, for a suite of standard subjects in Korat. This section describes the experiment procedure we designed to answer the following questions:

Q1. *How do our techniques compare in terms of effectiveness?*

Q2. *Does any of our techniques achieve optimal reduction?*

Q3. *Does the relation of # of valid to explored candidates, affect reduction?*

Q4. *How does the work each slave performs on average vary across our techniques and the SEQ-ON algorithm?*

4.1 Study

Table 4.1 shows the 7 subjects used in our study, which are taken from Korat's open-source repository[1]. Prior studies used similar subjects in their evaluation [1, 20, 26]. Due to the bounded exhaustive nature of Korat search, running these subjects does not scale for large finitization values. For

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

Subject
BinaryTree
BinomialHeap
DoublyLinkedList
HeapArray
RedBlackTree
SearchTree
SinglyLinkedList

Table 4.1: Subjects used in the study

instance, given $finRedBlackTree(12, 12, 12, 12)$ for the red-black tree example in Chapter 2, Korat explores 205,512,574 candidates in 4 minutes and finds 1,296 valid structures. We evaluated the effectiveness of our techniques for each subject, and compared it with the original SEQ-ON algorithm discussed in 3.1, with respect to the *reduction* definition from Subsection 3.2.4.

The tables we include in this chapter, present an illustrative subset of our experimental results; the full set of tables are in the Appendix A.

4.2 Results

Tables 4.2, 4.3, and 4.4 show some information for *BinaryTree*, *DoublyLinkedList*, and *RedBlackTree* subjects respectively. Each table contains number of candidates explored, number of valid instances found, and number of infeasible ranges for 5 different finitizations. Recall from section 3.2 that all the 3 presented techniques safely remove the head and tail infeasible ranges from the explored range; hence, the number of infeasible ranges in tables 4.2, 4.3, 4.4 excludes these two infeasible ranges. There are several key

	Finitization				
	2	4	6	8	10
Candidates explored	16	245	3653	54418	815100
Instances found	2	14	132	1430	16796
Infeasible ranges	1	13	131	1429	16795

Table 4.2: *BinaryTree* - Korat default

	Finitization				
	2	4	6	8	10
Candidates explored	27	94	776	17166	562823
Instances found	3	37	674	17007	562595
Infeasible ranges	0	0	0	0	0

Table 4.3: *DoublyLinkedList* - Korat default

	Finitization				
	2	4	6	8	10
Candidates explored	40	961	16487	322806	7530712
Instances found	2	8	20	64	260
Infeasible ranges	1	7	19	63	259

Table 4.4: *RedBlackTree* - Korat default

points obtained from these tables:

- The number of candidates explored grows considerably as the finitization increases, which shows Korat's bounded exhaustive testing technique does not scale with the finitization growth. For example, when finitization increases from 2 to 10, the number of explored candidates grows from 40 to 7,530,712 in *RedBlackTree*.

- Given a finitization, the number of infeasible ranges for *BinaryTree* and *RedBlackTree* is always one greater than instances found (including the head and tail infeasible ranges). We investigated this case and realized there is no consecutive valid candidates in the explored state. On the other hand, the same number is always 0 for *DoublyLinkedList*, and our further experiments showed that all the valid candidates are consecutive and the only 2 infeasible ranges are the head and tail. For instance, given finitization = 4, indexes of valid candidates are all the integers in this range: [31, 68), which makes the only infeasible ranges to be [0, 31) and [68, 94).

- For a fixed finitization, The number of infeasible ranges in each of these tables indicates the maximum ranges we can remove from the explored space. For instance, for the *BinaryTree* subject of size 6, at most 131 infeasible ranges can be removed and running our techniques for any number of infeasible ranges greater than 131, does not increase the pruned space. This threshold is called finitization$_{max}$.

- *RedBlackTree* has the smallest ratio of valid candidates to explored ones, which means most of the explored state is invalid based on the provided repOK. *DoublyLinkedList*, on the contrary, falls into the other extreme and has the largest ratio of valid to explored candidates. The *BinaryTree* example falls in between. Recall that our techniques aim to remove infeasible ranges; hence, we expect them to work more efficiently when there are more invalid explored candidates than valid ones.

	Equidistant candidates / *Infeasible ranges*	*Finitization* 2	4	6	8	10
MKorat$_{inf}$	1	56.25	12.24	1.72	0.19	0.02
	4	56.25	12.24	1.72	0.19	0.02
	16	87.50	28.57	1.72	0.19	0.02
	64	87.50	77.55	2.60	0.19	0.02
	256	87.50	94.28	45.96	0.19	0.02
	1024	87.50	94.28	86.42	6.25	0.02
MKorat$_{equ}$	1	87.50	16.73	2.35	0.31	0.02
	4	87.50	39.18	4.16	0.48	0.04
	16	87.50	85.30	13.82	1.49	0.11
	64	87.50	94.28	52.94	5.88	0.46
	256	87.50	94.28	96.38	19.03	1.78
	1024	87.50	94.28	96.38	69.65	6.79
MKorat	1	87.50	22.04	3.23	0.37	0.03
	4	87.50	46.12	7.33	0.88	0.09
	16	87.50	94.28	21.07	2.71	0.29
	64	87.50	94.28	61.10	8.91	1.01
	256	87.50	94.28	96.38	28.48	3.49
	1024	87.50	94.28	96.38	80.25	11.72

Table 4.5: *BinaryTree* reduction [%]

Tables 4.5, 4.6, and 4.7 show the reduction achieved for *BinaryTree*, *DoublyLinkedList*, and *RedBlackTree* subjects respectively, using our techniques MKorat$_{inf}$, MKorat$_{equ}$, and MKorat. Every table row shows the number of equidistant candidates and infeasible ranges (both set to the same value), and each column shows the size of the finitization. Note that in MKorat$_{equ}$, the number of infeasible ranges is determined based on the number of equidistant candidates. Hence, we considered these two parameters to be equal for the other two techniques as well, to make the comparison fair. In all the 3 tables:

- Given a finitization, reduction increases as the number of equidistant

	Equidistant candidates / *Infeasible ranges*	*Finitization* 2	4	6	8	10
MKorat$_{inf}$	1	88.88	60.63	13.14	0.92	0.04
	4	88.88	60.63	13.14	0.92	0.04
	16	88.88	60.63	13.14	0.92	0.04
	64	88.88	60.63	13.14	0.92	0.04
	256	88.88	60.63	13.14	0.92	0.04
	1024	88.88	60.63	13.14	0.92	0.04
MKorat$_{equ}$	1	88.88	60.63	13.14	0.92	0.04
	4	88.88	60.63	13.14	0.92	0.04
	16	88.88	60.63	13.14	0.92	0.04
	64	88.88	60.63	13.14	0.92	0.04
	256	88.88	60.63	13.14	0.92	0.04
	1024	88.88	60.63	13.14	0.92	0.04
MKorat	1	88.88	60.63	13.14	0.92	0.04
	4	88.88	60.63	13.14	0.92	0.04
	16	88.88	60.63	13.14	0.92	0.04
	64	88.88	60.63	13.14	0.92	0.04
	256	88.88	60.63	13.14	0.92	0.04
	1024	88.88	60.63	13.14	0.92	0.04

Table 4.6: *DoublyLinkedList* reduction [%]

candidates and infeasible ranges grow until it reaches a threshold, called $reduction_{max}$. For instance, in the *BinaryTree* subject, give finitization = 2, the maximum reduction is 87.50% and all the 3 techniques can achieve that for a large enough number of equidistant candidates and infeasible ranges.

- Given the same number of equidistant candidates and infeasible ranges, as finitization increases, reduction may increase or decrease for all techniques. For example, using MKorat for *BinaryTree*, with equidistant candidates and infeasible ranges equal to 16, reduction increases from

	Equidistant candidates / *Infeasible ranges*	*Finitization* 2	4	6	8	10
MKorat$_{inf}$	1	22.50	44.74	39.15	41.36	50.09
	4	70.00	44.74	39.15	41.36	50.09
	16	95.00	54.73	39.15	41.36	50.09
	64	95.00	88.86	69.33	52.78	58.80
	256	95.00	96.98	92.63	80.06	63.15
	1024	95.00	99.16	98.16	95.04	76.36
MKorat$_{equ}$	1	50.00	53.27	41.39	42.00	57.76
	4	95.00	69.92	52.26	42.75	50.37
	16	95.00	99.16	74.98	62.10	63.74
	64	95.00	99.16	99.87	81.70	68.14
	256	95.00	99.16	99.87	99.98	79.51
	1024	95.00	99.16	99.87	99.98	99.99
MKorat	1	95.00	54.11	49.20	48.34	57.76
	4	95.00	78.45	62.80	58.45	64.62
	16	95.00	99.16	93.04	69.22	66.63
	64	95.00	99.16	99.87	99.98	73.36
	256	95.00	99.16	99.87	99.98	99.59
	1024	95.00	99.16	99.87	99.98	99.99

Table 4.7: *RedBlackTree* reduction [%]

finitization 2 to 4, and decreases from finitization 4 to 6. This point indicates that the number of equidistant candidates and infeasible ranges should be adjusted properly based on the size of finitization to achieve the highest reduction.

Tables 4.8, 4.9, and 4.10 show the reduction for *BinaryTree*, *DoublyLinkedList*, and *RedBlackTree* subjects respectively. Each table is generated for finitization = 8. The rows are equidistant candidates and the columns are infeasible ranges. There are several points to discuss about these tables:

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	0.19	0.19	0.19	0.19	0.19	0.19
	4	0.19	0.19	0.19	0.19	0.19	0.19
	16	0.19	0.19	0.19	0.19	0.19	0.19
	64	0.19	0.19	0.19	0.19	0.19	0.19
	256	0.19	0.19	0.19	0.19	0.19	0.19
	1024	0.11	0.35	1.11	2.54	4.77	6.25
MKorat	1	0.37	0.88	2.71	8.91	28.48	80.25
	4	0.37	0.88	2.71	8.91	28.48	80.25
	16	0.37	0.88	2.71	8.91	28.48	80.25
	64	0.37	0.88	2.71	8.91	28.48	80.25
	256	0.37	0.88	2.71	8.91	28.48	80.25
	1024	0.37	0.88	2.71	8.91	28.48	80.25

Table 4.8: *BinaryTree* - reduction [%], Finitization = 8

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	0.92	0.92	0.92	0.92	0.92	0.92
	4	0.92	0.92	0.92	0.92	0.92	0.92
	16	0.92	0.92	0.92	0.92	0.92	0.92
	64	0.92	0.92	0.92	0.92	0.92	0.92
	256	0.92	0.92	0.92	0.92	0.92	0.92
	1024	0.63	0.92	0.92	0.92	0.92	0.92
MKorat	1	0.92	0.92	0.92	0.92	0.92	0.92
	4	0.92	0.92	0.92	0.92	0.92	0.92
	16	0.92	0.92	0.92	0.92	0.92	0.92
	64	0.92	0.92	0.92	0.92	0.92	0.92
	256	0.92	0.92	0.92	0.92	0.92	0.92
	1024	0.92	0.92	0.92	0.92	0.92	0.92

Table 4.9: *DoublyLinkedList* - reduction [%], Finitization = 8

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	41.36	41.36	41.36	41.36	41.36	41.36
	4	41.36	41.36	41.36	41.36	41.36	41.36
	16	41.36	41.36	41.36	41.36	41.36	41.36
	64	41.36	52.78	52.78	52.78	52.78	52.78
	256	40.75	55.66	62.32	79.45	80.06	80.06
	1024	40.75	56.38	66.69	94.44	95.04	95.04
MKorat	1	48.34	58.45	69.22	99.98	99.98	99.98
	4	48.34	58.45	69.22	99.98	99.98	99.98
	16	48.34	58.45	69.22	99.98	99.98	99.98
	64	48.34	58.45	69.22	99.98	99.98	99.98
	256	48.34	58.45	69.22	99.98	99.98	99.98
	1024	48.34	58.45	69.22	99.98	99.98	99.98

Table 4.10: *RedBlackTree* - reduction [%], Finitization = 8

- MKorat$_{equ}$ does not take the number of infeasible ranges into account; hence, the rows for this technique is not present in our tables.

- Given the number of infeasible ranges, MKorat$_{inf}$ and MKorat show different behavior as the number of equidistant candidates increase: In MKorat$_{inf}$, the reduction will increase, because equidistant ranges will have smaller size and more of them may fit in an infeasible range. However, MKorat is agnostic as to the number of infeasible ranges in terms of achieved reduction as it removes the given m-largest infeasible ranges.

- Given the number of equidistant candidates, for both techniques, reduction increases as the number of infeasible ranges grows, until it gets saturated.

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	3.5	107.5	1795.0	27155.0	407467.5
	4	1.4	43.0	718.0	10862.0	162987.0
	16	0.1	10.2	211.1	3194.7	47937.3
	64	0.1	0.8	54.7	835.5	12537.4
	256	0.1	0.0	7.6	211.3	3170.9
	1024	0.1	0.0	0.4	49.7	795.0
MKorat$_{equ}$	1	1.0	102.0	1783.5	27122.5	407443.5
	4	0.4	29.8	700.2	10830.4	162950.6
	16	0.1	2.1	185.1	3153.2	47889.6
	64	0.1	0.2	26.4	787.9	12481.4
	256	0.1	0.0	0.5	171.4	3114.9
	1024	0.1	0.0	0.1	16.1	741.1
MKorat	1	1.0	95.5	1767.5	27106.0	407391.0
	4	0.4	26.4	677.0	10786.8	162867.8
	16	0.1	0.8	169.5	3114.1	47806.2
	64	0.1	0.2	21.8	762.5	12413.2
	256	0.1	0.0	0.5	151.4	3060.8
	1024	0.1	0.0	0.1	10.4	702.0
	SEQ-ON	8.0	122.5	1826.5	27209.0	407550.0

Table 4.11: *BinaryTree* - *AVG* workload per worker

The next interesting study to perform is to measure the amount of work each worker has to perform on average, defined in Subsection 3.2.4, in each technique.

Tables 4.11, 4.12, 4.13 and tables 4.14, 4.15, and 4.16 show the average length of feasible ranges distributed among workers for our *BinaryTree*, *DoublyLinkedList*, and *RedBlackTree* subjects. The last row of each table shows the average workload in the original SEQ-ON algorithm. Note that not all the ranges in the SEQ-ON algorithm has the exact same length, as it may

	Equidistant candidates / *Infeasible ranges*	*Finitization* 2	4	6	8	10
MKorat$_{inf}$	1	1.5	18.5	337.0	8503.5	281297.5
	4	0.6	7.4	134.8	3401.4	112519.0
	16	0.1	2.1	39.6	1000.4	33093.8
	64	0.1	0.5	10.3	261.6	8655.3
	256	0.1	0.3	2.6	66.1	2189.0
	1024	0.1	0.3	0.8	16.5	548.8
MKorat$_{equ}$	1	1.5	18.5	337.0	8503.5	281297.5
	4	0.6	7.4	134.8	3401.4	112519.0
	16	0.1	2.1	39.6	1000.4	33093.8
	64	0.1	0.5	10.3	261.6	8655.3
	256	0.1	0.3	2.6	66.1	2189.0
	1024	0.1	0.3	0.8	16.5	548.8
MKorat	1	1.5	18.5	337.0	8503.5	281297.5
	4	0.6	7.4	134.8	3401.4	112519.0
	16	0.1	2.1	39.6	1000.4	33093.8
	64	0.1	0.5	10.3	261.6	8655.3
	256	0.1	0.3	2.6	66.1	2189.0
	1024	0.1	0.3	0.8	16.5	548.8
	SEQ-ON	13.5	47.0	388.0	8583.0	281411.5

Table 4.12: *DoublyLinkedList - AVG* workload per worker

be infeasible to break an arbitrary explored space into a set of desired ranges with equal length. There are several points to discuss about these tables:

- The average value is always lower in our techniques compared to the original SEQ-ON algorithm, which empirically shows our point from Subsection 3.2.4, that workers do less work on average in our techniques.
- In most of the cases, MKorat works better than MKorat$_{equ}$, and MKorat$_{equ}$ works better than MKorat$_{inf}$.

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	15.5	265.5	5016.0	94646.0	1878983.0
	4	2.4	106.2	2006.4	37858.4	751593.2
	16	0.1	25.5	590.1	11134.8	221056.8
	64	0.0	1.6	77.7	2345.0	47732.4
	256	0.0	0.1	4.7	250.4	10797.3
	1024	0.0	0.0	0.3	15.5	1736.1
MKorat$_{equ}$	1	10.0	224.5	4831.5	93607.5	1590464.5
	4	0.4	57.8	1574.0	36957.2	747464.6
	16	0.1	0.4	242.5	7195.4	160584.7
	64	0.0	0.1	0.3	908.4	36905.0
	256	0.0	0.0	0.0	0.2	6002.5
	1024	0.0	0.0	0.0	0.0	0.2
MKorat	1	1.0	220.5	4187.5	83379.5	1590464.5
	4	0.4	41.4	1226.6	26824.8	532835.0
	16	0.1	0.4	67.4	5843.8	147781.2
	64	0.0	0.1	0.3	0.9	30854.4
	256	0.0	0.0	0.0	0.2	118.5
	1024	0.0	0.0	0.0	0.0	0.2
	SEQ-ON	20.0	480.5	8243.5	161403.0	3765356.0

Table 4.13: *RedBlackTree* - *AVG* workload per worker

4.3 Answers to research questions

We started this chapter with several research questions. In this section we can answer them, based on the previous sections:

4.3.1 *Q1.* How do our techniques compare in terms of effectiveness?

MKorat has shown to be the most effective technique, followed by MKorat$_{equ}$ and MKorat$_{inf}$. Further, as the pruning of our approaches is based on removing infeasible ranges, the most achievable reduction, i.e., *reduction*$_{max}$ happens when all the infeasible ranges are removed.

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	27155.0	27155.0	27155.0	27155.0	27155.0	27155.0
	4	10862.0	10862.0	10862.0	10862.0	10862.0	10862.0
	16	3194.7	3194.7	3194.7	3194.7	3194.7	3194.7
	64	835.5	835.5	835.5	835.5	835.5	835.5
	256	211.3	211.3	211.3	211.3	211.3	211.3
	1024	53.0	52.9	52.5	51.7	50.5	49.7
MKorat	1	27106.0	26967.0	26470.0	24783.5	19457.5	5372.5
	4	10842.4	10786.8	10588.0	9913.4	7783.0	2149.0
	16	3188.9	3172.5	3114.1	2915.7	2289.1	632.0
	64	834.0	829.7	814.4	762.5	598.6	165.3
	256	210.9	209.8	205.9	192.8	151.4	41.8
	1024	52.8	52.6	51.6	48.3	37.9	10.4
	SEQ-ON	27209.0	27209.0	27209.0	27209.0	27209.0	27209.0

Table 4.14: *BinaryTree* - *AVG* workload per worker, $Finitization = 8$

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	8503.5	8503.5	8503.5	8503.5	8503.5	8503.5
	4	3401.4	3401.4	3401.4	3401.4	3401.4	3401.4
	16	1000.4	1000.4	1000.4	1000.4	1000.4	1000.4
	64	261.6	261.6	261.6	261.6	261.6	261.6
	256	66.1	66.1	66.1	66.1	66.1	66.1
	1024	16.6	16.5	16.5	16.5	16.5	16.5
MKorat	1	8503.5	8503.5	8503.5	8503.5	8503.5	8503.5
	4	3401.4	3401.4	3401.4	3401.4	3401.4	3401.4
	16	1000.4	1000.4	1000.4	1000.4	1000.4	1000.4
	64	261.6	261.6	261.6	261.6	261.6	261.6
	256	66.1	66.1	66.1	66.1	66.1	66.1
	1024	16.5	16.5	16.5	16.5	16.5	16.5
	SEQ-ON	8583.0	8583.0	8583.0	8583.0	8583.0	8583.0

Table 4.15: *DoublyLinkedList* - *AVG* workload per worker, $Finitization = 8$

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	94646.0	94646.0	94646.0	94646.0	94646.0	94646.0
	4	37858.4	37858.4	37858.4	37858.4	37858.4	37858.4
	16	11134.8	11134.8	11134.8	11134.8	11134.8	11134.8
	64	2912.1	2345.0	2345.0	2345.0	2345.0	2345.0
	256	744.1	556.8	473.1	258.0	250.4	250.4
	1024	186.5	137.3	104.9	17.4	15.5	15.5
MKorat	1	83379.5	67062.0	49672.5	32.0	32.0	32.0
	4	33351.8	26824.8	19869.0	12.8	12.8	12.8
	16	9809.3	7889.6	5843.8	3.7	3.7	3.7
	64	2565.5	2063.4	1528.3	0.9	0.9	0.9
	256	648.8	521.8	386.5	0.2	0.2	0.2
	1024	162.6	130.8	96.9	0.0	0.0	0.0
	SEQ-ON	161403.0	161403.0	161403.0	161403.0	161403.0	161403.0

Table 4.16: *RedBlackTree* - *AVG* workload per worker, $Finitization = 8$

4.3.2 *Q2*. Does any of our techniques achieve optimal reduction?

MKorat provides the optimal reduction ($reduction_{opt}(EQU, INF)$ from Subsection 3.2.4), by removing the m-largest infeasible ranges. Neither MKorat$_{inf}$, nor MKorat$_{equ}$ achieve the optimal reduction, as they do not necessarily remove the largest infeasible ranges.

4.3.3 *Q3*. Does the relation of # of valid to explored candidates, affect reduction?

Our techniques are more effective when the space explored is sparse, i.e., the number of invalid candidate vectors explored is considerably larger than valid ones. As an extreme case, a repOK method returning a constant, achieves 0% or 100% reduction for the constant values *true* and *false* respectively, in all the 3 proposed techniques.

4.3.4 *Q4.* How does the work each slave performs on average vary across our techniques and the SEQ-ON algorithm?

Our experiments show that the amount of average work performed by each slave in our techniques, is at most the same as the original SEQ-ON algorithm, and in most cases considerably smaller. Further, in most cases, MKorat performs better than MKorat$_{equ}$, and MKorat$_{equ}$ outperforms MKorat$_{inf}$.

4.4 Execution platform

We run all the experiments on a machine with 4-cores, Intel® Core™ i7-4770HQ CPU at 2.20GHz, with 16GB of RAM, running OS X 10.10.5. We used Java 1.8.0 60 from Oracle®.

4.5 Threats to validity

4.5.1 External

The subjects used in our study may not be representative. To mitigate this threat, we considered 7 subjects shipped with Korat source code that vary in code size, and complexity of repOK. Some of these projects have also been used in prior studies on Korat. The results may vary for different finitizations, number of equidistant candidates, and number of infeasible ranges. Exploring all the combinations is not feasible. Yet, to attenuate 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 [1, 20, 26].

4.5.2 Internal

Korat, our extended version of Korat that implements the techniques, and our automation scripts, may contain bugs that can impact our conclusions. We are mostly confident in the correctness of Korat, as is a robust tool used in several prior studies. To increase the confidence in our implementations, we developed the core parts of our techniques twice with two different algorithms: (1) efficient and (2) inefficient, in terms of time and space complexity, and observed that they produce the same result for several examples. Further, to increase the confidence in our scripts, we reviewed our code, tested it on a number of subjects manually, and inspected several results.

Chapter 5

Potential applications

The chapter first introduces an abstract search problem, called *online next-valid-neighbor problem* (Section 5.1) and describes how our approach solves this problem, and then describes two concrete domains – constraint-driven game play (Section 5.2) and constraint-driven data structure repair (Section 5.3) – where this problem arises and discusses the potential our approach holds for providing more efficient solutions in these domains.

5.1 Online next-valid-neighbor problem

5.1.1 Problem

Consider input generation based on input constraint repOK and finitization f. Let S denote the set of all structures (valid or invalid) that are explored by the Korat search with respect to f. Given a candidate structure $s \in S$, the next-valid-neighbor problem is the problem to generate structure t, if any, such that either (1) s is valid and $s = t$ or (2) s is invalid and running Korat search starting at the candidate vector that represents s creates t as the first valid structure.

The online next-valid-neighbor problem is the problem where the next-

valid-neighbor is repeatedly solved for different input structures with respect to fixed finitization f and input constraint repOK. Thus, Korat search is repeatedly run starting at different initial candidate vectors.

5.1.2 Solution

A straightforward solution to the online next-valid-neighbor problem is to directly apply the problem definition and run the Korat search on each input structure and return the first valid structure created by Korat.

Our approach of using infeasible ranges provides the basis for defining a novel solution that is likely more efficient. Figure 5.1 contains the pseudocode of our solution:

5.2 Game play

Consider a 2-player game played on a $k \times k$ grid where $k \geq 1$ is a fixed integer, such as *tic-tac-toe*, where:

- Each grid cell is empty or contains one token from a finite set of tokens
- One player starts the game by making a legal move
- The 2 players take turns making legal moves
- The game ends when some player wins or there is a tie

```
// infeasible ranges maintained in program state
Map<Structure, Structure> ranges =
    new HashMap<Structure, Structure>();

// assuming s is an invalid structure
Structure onlineNextValidNeighbor(Structure s) {
  for (Map.Entry<Structure, Structure> e: ranges.entrySet())
    // if structure s falls in a known infeasible range
    // e=(p, n), then the next-valid-neighbor of s is n
    if (inRange(s, e))
      return e.getValue(); // n

  Structure n = Korat.findFirstValid(s);

  // if no valid structure found starting from s
  if (n == null) return null;

  // if our map has an infeasible range x=(s', n), we need
  // to expand the beginning of the range by updating it to
  // x=(s, n). Note that s < s' as the function has not yet
  // returned and the if branch is taken
  if (ranges.values().contains(n))
    updateRanges(ranges, s, n);

  // else a new infeasible range is found and should be
  // added in the map
  else
    ranges.put(s, n);

  return n;
}
```

Figure 5.1: Next-valid-neighbor

5.2.1 Constraint-driven game play

We introduce *constraint-driven game play*, an approach to 2-player games where the game rules are defined by two logical constraints, namely *gameOver* and *validMove*, which define the rules that govern when the game is over and the rules that govern legal moves respectively, and are written in

```
1    void play() {
2      Game g = new Game(); // empty board
3      boolean turn = toss(); // choose turn randomly
4
5      while (!g.gameOver()) {
6        if (turn)
7          // auto-generated next move
8          g = Korat.findFirstValidMove(g, turn);
9        else
10         // ask human user for a gameplay
11         g = getHumanMove(g, turn);
12
13       turn = !turn;
14     }
15   }
```

Figure 5.2: Automated game play

the spirit of repOK methods.

To illustrate, consider modeling tic-tac-toe using two user-provided predicates, *gameOver* and *validMove*, that show if a game state is final, and if a transition between two game states is valid respectively. A direct application of Korat provides a solution for automated game play. As an example, consider the scenario where computer is playing tic-tac-toe, with a human player. Figure 5.2 provides the game play pseudocode for this scenario:

Now, consider *repeated* game play where the game is played multiple times, each time starting at the game's initial state and terminating when the game is over. A straightforward solution to this problem is provided by invoking the *play* method repeatedly. Our approach of utilizing infeasible ranges as embodied in the *onlineNextValidNeighbor* algorithm provides the basis for a novel solution that is likely more efficient. Figure 5.3 presents our

```
1   // infeasible ranges maintained in the program state
2   Map<Game, Game> ranges = HashMap<Game, Game>();
3
4   // finds the first valid game play coming after g
5   Game findFirstValidMove(Game g, boolean turn) {
6     // if the next game state is known from prior game plays
7     if (ranges.values().contains(g)) {
8       return ranges.get(g);
9     }
10
11    // use Korat to search for the next valid game play
12    Game updated = Korat.findFirstValidMove(g, turn);
13    ranges.put(g, updated); // update the cache
14
15    return updated;
16  }
17
18  void play() {
19    Game g = new Game();
20    boolean turn = toss();
21
22    while (!g.gameOver()) {
23      if (turn)
24        g = findFirstValidMove(g, turn);
25      else
26        g = getHumanMove(g, turn);
27
28      turn = !turn;
29    }
30  }
```

Figure 5.3: Automated game play using infeasible range caching

approach.

5.3 Data structure repair

Data structure repair [5, 6, 8, 13, 14, 21] is an approach for error recovery, where the error may be in program state, such as memory, or persistent data,

```
1    void repair(Structure s) {
2      // build the candidate vector that corresponds
3      // to the broken data structure s
4      CandidateVector cv = canonicalizeWRTRepOk(s);
5
6      // find the next valid candidate using Korat search
7      cv = Korat.findFirstValidCV(cv);
8
9      // update structure s based on the new valid
10     // candidate vector found
11     updateStructure(s, cv);
12   }
```

Figure 5.4: Data structure repair

such as the file system.

5.3.1 Constraint-driven data structure repair

Constraint-driven data structure repair [5, 6] uses given logical constraints as specifications for repairing erroneous program states. Juzi [8, 14] introduced the use of imperative constraints, in the form of repOK methods, in data structure repair.

The constraint-driven data structure repair problem is defined as follows [8]. Given a repOK method that describes expected structural constraints and an erroneous structure s such that *!s.repOk()*, mutate s into t such that *t.repOK()* and t is "similar" to s. Similarity is a heuristic notion and is intended to restrict repair to avoid unnecessary mutations and to preserve as much of the original structure as possible while satisfying the structural constraints. Figure 5.4 shows the basis that Korat provides to solve this problem:

The method *canonicalizeWRTRepOk* translates the structure s to a candidate vector, which has all fields set to 0 except for the ones accessed by repOK, which are set to *canonical* values that are created using linearization [30]. The *repair* algorithm borrows the spirit of Juzi but differs in the way that *repair* is faithful to the Korat search whereas Juzi modifies the Korat search. Specifically, when Korat backtracks, the last field accessed in the candidate vector gets the *next* value according to field domain ordering, however, Juzi considers *all* values other than the original value of that field when backtracking.

Now, consider *repeated* data structure repair where the repair is performed multiple times, possibly on *different* erroneous states as inputs conform to a fixed finitization. A straightforward solution to this problem is provided by invoking the *repair* method repeatedly. Our approach of utilizing infeasible ranges as embodied in the *onlineNextValidNeighbor* algorithm provides the basis for a novel solution that is likely more efficient. Figure 5.5 presents our algorithm.

5.3.2 Example

Recall Figure 2.3 which shows the candidates explored in Korat search given *finRedBlackTree(2, 2, 2, 2)*. The range $(10, 29)$ is one infeasible range in the explored space. Figure 5.7 shows two invalid structures a and b, within this range, corresponding to candidates in lines 14 and 22 respectively. Figure 5.8 shows the valid structure c, our repair algorithm in Figure 5.4, selects for

```
// infeasible ranges maintained in program state
Map<CandidateVector, CandidateVector> ranges =
    new HashMap<CandidateVector, CandidateVector>();

CandidateVector findFirstValidCV(CandidateVector cv) {
  for (Map.Entry<CandidateVector, CandidateVector> e:
      ranges.entrySet())
    // if candidate cv falls in a known infeasible range
    // e=(p, n), then the next-valid-neighbor of cv is n
    if (inRange(cv, e))
      return e.getValue(); // n

  CandidateVector n = Korat.findFirstValidCV(cv);

  // if no valid structure found starting from cv
  if (n == null) return null;

  // if our map has an infeasible range x=(cv', n), we need
  // to expand the beginning of the range by updating it to
  // x=(cv, n). Note that cv < cv' as the function has not yet
  // returned and the if branch is taken
  if (ranges.values().contains(n))
    updateRanges(ranges, cv, n);

  // else a new infeasible range is found and should be
  // added in the map
  else
    ranges.put(cv, n);

  return n;
}

void repair(Structure s) {
  CandidateVector cv = canonicalizeWRTRepOk(s);
  cv = findFirstValidCV(cv);
  updateStructure(s, cv);
}
```

Figure 5.5: Data structure repair using infeasible range caching

Figure 5.7: Red-black trees a and b

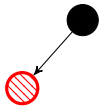

Figure 5.8: Red-black tree c

both of these invalid structures, as it corresponds to the first valid candidate explored after them (line 29 of Figure 2.3). Imagine the following two repair scenarios using our algorithm described in Figure 5.5:

1. *repair(a); repair(b);* The first call, *repair(a)*, makes a direct invocation to Korat search and finds the valid structure c. In addition, it records the explored infeasible range *r=(CV(a), CV(c))*. Next, *repair(b)* does not invoke Korat search as the first valid structure for b is already known, i.e., *inRange(CV(b), r)* returns *true*.

2. *repair(b); repair(a);* The first *repair(b)* finds c by directly invoking Korat, plus storing the range *r=(CV(b), CV(c))*. Next, *repair(a)* cannot find *CV(a)* in r, and makes another direct call to Korat search starting from a, and it updates the the range r to *(CV(a), CV(c))*.

Chapter 6

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 [1], and one white-box testing technique, namely *symbolic execution* [16], and one approach, namely *model checking* [3], for testing multi-threaded programs.

6.1 Parallel Korat

The idea of parallel test generation and execution in the context of Korat was introduced by Misailovic et al. [20]. The idea of infeasible ranges is rooted in their discussion on potential optimizations [20] where they observe the potential usefulness of creating sub-ranges that start and end at valid candidates. Our second technique, MKorat$_{equ}$, embodies this observation whereas our other two techniques build on it.

PKorat [26] 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 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 infeasible ranges and re-explores all candidates that sequential Korat explores by default. 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.

6.2 Parallel symbolic execution

ParSym [24] 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* [27] 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.

Ranged symbolic execution [25] 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 [23] 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 infeasible 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 case 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 infeasible ranges in symbolic execution.

6.3 Parallel model checking

Funes et al. [9] introduced the idea of ranging for software model checking using *Java PathFinder (JPF)* [29], 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 errorneous state faster than the sequential run of the model checker [7]. One of the earliest techniques for parallel search for explicit state checking was parallel *Murϕ*, introduced by Stern and Dill [28], and shown to provide approximately linear speedups.

Chapter 7

Conclusion

We introduced a novel approach for memoizing Korat – a systematic testing technique based on backtracking search that explores large spaces of candidate inputs to find desired inputs that are described by imperative predicates given by the user. Our approach builds on previous work on parallel test generation using Korat, specifically the SEQ-ON algorithm, which allows efficient re-execution of Korat for input generation using parallel workers with evenly distributed workloads. Our key insight is that the Korat search typically encounters and inspects many *consecutive* candidates that are all invalid inputs, and such *invalid ranges* of candidates can be *memoized* succinctly to optimize re-execution of Korat, so it can simply prune those candidates when the search re-executes. We presented three new techniques that embody our insight and build on SEQ-ON, evaluated the algorithms using a standard suite of subjects to show the efficacy of our approach, and showed how it enables Korat to be applied in two new application domains, namely for game play and data structure repair.

We believe our work opens a promising new direction to optimize solving of imperative constraints. In future work, we plan to explore the use of

feasible ranges [23] for memoization in Korat, develop techniques for utilizing infeasible and feasible ranges for incremental solving [17, 22, 31] in Korat, e.g., when it is re-run after a change to finitization or repOK, as well as other novel applications of memoized constraint solving.

Appendices

Appendix A

Evaluation Appendix

	Finitization				
	2	4	6	8	10
Candidates explored	17	139	2194	52567	1702171
Instances found	2	15	203	4140	115975
Infeasible ranges	1	14	202	4139	115974

Table A.1: *SinglyLinkedList* - Korat default

	Finitization				
	2	4	6	8	10
Candidates explored	58	1666	42815	1323194	150727471
Instances found	6	120	7602	603744	117157172
Infeasible ranges	3	23	941	33555	6628009

Table A.2: *BinomialHeap* - Korat default

	Finitization				
	2	4	6	8	10
Candidates explored	22	875	45233	2606968	155455872
Instances found	2	14	132	1430	16796
Infeasible ranges	1	13	131	1429	16795

Table A.3: *SearchTree* - Korat default

	Finitization				
	2	4	6	8	10
Candidates explored	45	1425	64533	5231385	583317405
Instances found	15	320	13139	1005075	111511015
Infeasible ranges	8	179	6082	423977	TBA

Table A.4: *HeapArray* - Korat default

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat_{inf}	1	47.05	17.98	1.91	0.11	0.00
	4	70.58	17.98	1.91	0.11	0.00
	16	88.23	35.25	1.91	0.11	0.00
	64	88.23	79.85	1.91	0.11	0.00
	256	88.23	89.20	23.06	0.11	0.00
	1024	88.23	89.20	79.48	0.24	0.00
MKorat_{equ}	1	70.58	21.58	2.59	0.17	0.00
	4	88.23	44.60	5.24	0.22	0.00
	16	88.23	78.41	11.48	0.52	0.01
	64	88.23	89.20	37.51	1.78	0.06
	256	88.23	89.20	88.83	7.18	0.24
	1024	88.23	89.20	90.74	26.46	0.95
MKorat	1	88.23	29.49	3.41	0.22	0.00
	4	88.23	53.23	7.24	0.50	0.02
	16	88.23	89.20	18.64	1.45	0.06
	64	88.23	89.20	46.71	4.54	0.23
	256	88.23	89.20	90.74	13.84	0.80
	1024	88.23	89.20	90.74	38.79	2.70

Table A.5: *SinglyLinkedList* reduction [%]

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	20.68	7.08	10.17	1.55	0.66
	4	75.86	22.44	10.17	1.55	0.66
	16	82.75	72.38	38.87	11.45	0.66
	64	89.65	84.87	63.98	31.26	0.66
	256	89.65	88.95	71.45	41.79	2.05
	1024	89.65	92.19	73.10	44.42	5.88
MKorat$_{equ}$	1	84.48	45.61	10.17	1.55	0.66
	4	84.48	72.92	52.24	15.82	0.66
	16	87.93	89.91	64.92	38.46	1.87
	64	89.65	90.75	73.92	45.73	1.75
	256	89.65	92.13	74.33	45.75	6.09
	1024	89.65	92.79	75.84	45.79	7.35
MKorat	1	84.48	45.61	28.06	12.77	1.87
	4	87.93	90.03	67.01	34.21	4.73
	16	87.93	92.31	74.19	45.76	7.35
	64	87.93	92.73	75.53	45.84	7.35
	256	87.93	92.73	78.61	46.10	7.36
	1024	87.93	92.73	82.24	46.79	7.38

Table A.6: *BinomialHeap* reduction [%]

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	18.18	7.20	0.73	0.06	0.00
	4	68.18	7.20	0.73	0.06	0.00
	16	90.90	14.51	0.73	0.06	0.00
	64	90.90	77.59	0.73	0.06	0.00
	256	90.90	94.28	42.90	0.06	0.00
	1024	90.90	98.40	85.55	0.06	0.00
MKorat$_{equ}$	1	50.00	13.94	1.49	0.13	0.01
	4	90.90	35.31	3.71	0.35	0.02
	16	90.90	98.40	12.87	1.18	0.10
	64	90.90	98.40	49.21	4.56	0.38
	256	90.90	98.40	99.70	18.01	1.52
	1024	90.90	98.40	99.70	71.81	6.10
MKorat	1	90.90	17.14	1.80	0.17	0.01
	4	90.90	40.57	4.65	0.43	0.03
	16	90.90	98.40	14.87	1.40	0.12
	64	90.90	98.40	51.86	5.01	0.43
	256	90.90	98.40	99.70	18.81	1.63
	1024	90.90	98.40	99.70	72.54	6.31

Table A.7: *SearchTree* reduction [%]

	Equidistant candidates / *Infeasible ranges*	*Finitization* 2	4	6	8	10
MKorat$_{inf}$	1	0.00	0.00	0.00	0.00	0.00
	4	0.00	0.00	0.00	0.00	0.00
	16	44.44	0.00	0.00	0.00	0.00
	64	66.66	0.00	0.00	0.00	0.00
	256	66.66	26.38	0.00	0.00	0.00
	1024	66.66	71.08	0.00	0.00	0.00
MKorat$_{equ}$	1	15.55	0.35	0.01	0.00	0.00
	4	40.00	1.54	0.06	0.00	0.00
	16	66.66	8.07	0.28	0.00	0.00
	64	66.66	29.26	0.96	0.01	0.00
	256	66.66	73.26	3.98	0.05	0.00
	1024	66.66	77.54	15.82	0.20	0.00
MKorat	1	17.77	1.54	0.06	0.00	0.00
	4	46.66	5.05	0.23	0.00	0.00
	16	64.44	16.98	0.88	0.01	0.00
	64	64.44	45.19	3.01	0.07	0.00
	256	64.44	77.47	9.54	0.24	0.00
	1024	64.44	77.47	28.52	0.88	0.01

Table A.8: *HeapArray* reduction [%]

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	0.11	0.11	0.11	0.11	0.11	0.11
	4	0.11	0.11	0.11	0.11	0.11	0.11
	16	0.11	0.11	0.11	0.11	0.11	0.11
	64	0.11	0.11	0.11	0.11	0.11	0.11
	256	0.11	0.11	0.11	0.11	0.11	0.11
	1024	0.01	0.11	0.18	0.24	0.24	0.24
MKorat	1	0.22	0.50	1.45	4.54	13.84	38.79
	4	0.22	0.50	1.45	4.54	13.84	38.79
	16	0.22	0.50	1.45	4.54	13.84	38.79
	64	0.22	0.50	1.45	4.54	13.84	38.79
	256	0.22	0.50	1.45	4.54	13.84	38.79
	1024	0.22	0.50	1.45	4.54	13.84	38.79

Table A.9: *SinglyLinkedList* - reduction [%], Finitization = 8

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	1.55	1.55	1.55	1.55	1.55	1.55
	4	1.55	1.55	1.55	1.55	1.55	1.55
	16	11.45	11.45	11.45	11.45	11.45	11.45
	64	11.45	26.31	31.26	31.26	31.26	31.26
	256	11.14	30.33	41.79	41.79	41.79	41.79
	1024	11.14	32.03	44.42	44.42	44.42	44.42
MKorat	1	12.77	34.21	45.76	45.84	46.10	46.79
	4	12.77	34.21	45.76	45.84	46.10	46.79
	16	12.77	34.21	45.76	45.84	46.10	46.79
	64	12.77	34.21	45.76	45.84	46.10	46.79
	256	12.77	34.21	45.76	45.84	46.10	46.79
	1024	12.77	34.21	45.76	45.84	46.10	46.79

Table A.10: *BinomialHeap* - reduction [%], Finitization = 8

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	0.06	0.06	0.06	0.06	0.06	0.06
	4	0.06	0.06	0.06	0.06	0.06	0.06
	16	0.06	0.06	0.06	0.06	0.06	0.06
	64	0.06	0.06	0.06	0.06	0.06	0.06
	256	0.06	0.06	0.06	0.06	0.06	0.06
	1024	0.06	0.06	0.06	0.06	0.06	0.06
MKorat	1	0.17	0.43	1.40	5.01	18.81	72.54
	4	0.17	0.43	1.40	5.01	18.81	72.54
	16	0.17	0.43	1.40	5.01	18.81	72.54
	64	0.17	0.43	1.40	5.01	18.81	72.54
	256	0.17	0.43	1.40	5.01	18.81	72.54
	1024	0.17	0.43	1.40	5.01	18.81	72.54

Table A.11: *SearchTree* - reduction [%], Finitization = 8

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	0.00	0.00	0.00	0.00	0.00	0.00
	4	0.00	0.00	0.00	0.00	0.00	0.00
	16	0.00	0.00	0.00	0.00	0.00	0.00
	64	0.00	0.00	0.00	0.00	0.00	0.00
	256	0.00	0.00	0.00	0.00	0.00	0.00
	1024	0.00	0.00	0.00	0.00	0.00	0.00
MKorat	1	0.00	0.00	0.01	0.07	0.24	0.88
	4	0.00	0.00	0.01	0.07	0.24	0.88
	16	0.00	0.00	0.01	0.07	0.24	0.88
	64	0.00	0.00	0.01	0.07	0.24	0.88
	256	0.00	0.00	0.01	0.07	0.24	0.88
	1024	0.00	0.00	0.01	0.07	0.24	0.88

Table A.12: *HeapArray* - reduction [%], Finitization = 8

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	4.5	57.0	1076.0	26252.0	851041.5
	4	1.0	22.8	430.4	10500.8	340416.6
	16	0.1	5.2	126.5	3088.4	100122.5
	64	0.1	0.4	33.1	807.7	26185.8
	256	0.1	0.1	6.5	204.3	6622.8
	1024	0.1	0.1	0.4	51.1	1660.5
MKorat$_{equ}$	1	2.5	54.5	1068.5	26238.0	851036.0
	4	0.4	15.4	415.8	10489.8	340407.8
	16	0.1	1.7	114.2	3076.0	100108.7
	64	0.1	0.2	21.0	794.2	26170.5
	256	0.1	0.1	0.9	189.8	6606.8
	1024	0.1	0.1	0.2	37.7	1644.7
MKorat	1	1.0	49.0	1059.5	26225.0	851002.0
	4	0.4	13.0	407.0	10460.6	340356.4
	16	0.1	0.8	105.0	3047.1	100057.6
	64	0.1	0.2	17.9	771.9	26125.4
	256	0.1	0.1	0.7	176.2	6569.9
	1024	0.1	0.1	0.2	31.3	1615.7
	SEQ-ON	8.5	69.5	1097.0	26283.5	851085.5

Table A.13: *SinglyLinkedList* - *AVG* workload per worker

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	23.0	774.0	19230.0	651335.5	74861949.0
	4	2.8	258.4	7692.0	260534.2	29944779.6
	16	0.5	27.0	1539.5	68917.5	8807288.1
	64	0.1	3.8	237.2	13991.6	2303444.5
	256	0.1	0.7	47.5	2996.8	574423.1
	1024	0.1	0.1	11.2	717.4	138399.5
MKorat$_{equ}$	1	4.5	453.0	19230.0	651335.5	74861949.0
	4	1.8	90.2	4089.2	222764.0	29944778.8
	16	0.4	9.8	883.2	47899.0	8699784.3
	64	0.1	2.3	171.7	11045.6	2278270.6
	256	0.1	0.5	42.7	2793.1	550727.5
	1024	0.1	0.1	10.0	699.8	136230.4
MKorat	1	4.5	453.0	15399.0	577065.5	73948185.5
	4	1.4	33.2	2824.4	174084.2	28719235.8
	16	0.4	7.5	649.9	42215.2	8213947.1
	64	0.1	1.8	161.1	11024.2	2148216.5
	256	0.1	0.4	35.6	2774.6	543284.6
	1024	0.1	0.1	7.4	686.8	136188.3
	SEQ-ON	29.0	833.0	21407.5	661597.0	75363735.5

Table A.14: *BinomialHeap* - *AVG* workload per worker

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	9.0	406.0	22450.0	1302607.5	77723578.5
	4	1.4	162.4	8980.0	521043.0	31089431.4
	16	0.1	44.0	2641.1	153247.9	9143950.4
	64	0.0	3.0	690.7	40080.2	2391494.7
	256	0.0	0.1	100.5	10137.0	604852.7
	1024	0.0	0.0	6.3	2541.6	151655.7
MKorat$_{equ}$	1	5.5	376.5	22278.5	1301692.0	77718956.0
	4	0.4	113.2	8710.6	519559.2	31082023.4
	16	0.1	0.8	2318.0	151536.2	9135226.2
	64	0.0	0.2	353.4	38275.0	2382398.3
	256	0.0	0.0	0.5	8316.1	595637.3
	1024	0.0	0.0	0.1	716.8	142407.9
MKorat	1	1.0	362.5	22207.5	1301253.0	77716779.0
	4	0.4	104.0	8625.2	519117.4	31079611.0
	16	0.1	0.8	2264.8	151195.7	9133443.9
	64	0.0	0.2	335.0	38096.6	2381223.3
	256	0.0	0.0	0.5	8235.1	594967.6
	1024	0.0	0.0	0.1	698.2	142085.9
	SEQ-ON	11.0	437.5	22616.5	1303484.0	77727936.0

Table A.15: *SearchTree* - *AVG* workload per worker

	Equidistant candidates	*Finitization*				
	Infeasible ranges	2	4	6	8	10
MKorat$_{inf}$	1	22.5	712.5	32266.5	2615692.5	291658702.5
	4	9.0	285.0	12906.6	1046277.0	116663481.0
	16	1.4	83.8	3796.0	307728.5	34312788.5
	64	0.3	21.9	992.8	80482.8	8974113.9
	256	0.3	4.0	251.1	20355.5	2269717.5
	1024	0.3	0.4	62.9	5103.7	569090.1
MKorat$_{equ}$	1	19.0	710.0	32263.0	2615689.0	291658690.0
	4	5.4	280.6	12898.6	1046267.8	116663467.2
	16	0.8	77.0	3785.0	307718.6	34312775.0
	64	0.3	15.5	983.2	80472.9	8974100.7
	256	0.3	1.4	241.0	20344.7	2269704.1
	1024	0.3	0.3	53.0	5093.3	569076.8
MKorat	1	18.5	701.5	32244.5	2615655.5	291658646.5
	4	4.8	270.6	12876.0	1046223.6	116663398.4
	16	0.9	69.5	3762.4	307668.3	34312694.3
	64	0.3	12.0	962.8	80425.6	8974021.2
	256	0.3	1.2	227.1	20304.9	2269631.2
	1024	0.3	0.3	45.0	5058.5	569010.8
	SEQ-ON	22.5	712.5	32266.5	2615692.5	291658702.5

Table A.16: *HeapArray - AVG* workload per worker

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	26252.0	26252.0	26252.0	26252.0	26252.0	26252.0
	4	10500.8	10500.8	10500.8	10500.8	10500.8	10500.8
	16	3088.4	3088.4	3088.4	3088.4	3088.4	3088.4
	64	807.7	807.7	807.7	807.7	807.7	807.7
	256	204.3	204.3	204.3	204.3	204.3	204.3
	1024	51.2	51.2	51.1	51.1	51.1	51.1
MKorat	1	26225.0	26151.5	25900.5	25088.5	22643.5	16087.5
	4	10490.0	10460.6	10360.2	10035.4	9057.4	6435.0
	16	3085.2	3076.6	3047.1	2951.5	2663.9	1892.6
	64	806.9	804.6	796.9	771.9	696.7	495.0
	256	204.0	203.5	201.5	195.2	176.2	125.1
	1024	51.1	51.0	50.5	48.9	44.1	31.3
	SEQ-ON	26283.5	26283.5	26283.5	26283.5	26283.5	26283.5

Table A.17: *SinglyLinkedList* - *AVG* workload per worker, $Finitization = 8$

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	651335.5	651335.5	651335.5	651335.5	651335.5	651335.5
	4	260534.2	260534.2	260534.2	260534.2	260534.2	260534.2
	16	68917.5	68917.5	68917.5	68917.5	68917.5	68917.5
	64	18024.6	14999.8	13991.6	13991.6	13991.6	13991.6
	256	4574.8	3586.7	2996.8	2996.8	2996.8	2996.8
	1024	1147.0	877.3	717.4	717.4	717.4	717.4
MKorat	1	577065.5	435210.5	358829.5	358287.0	356535.5	352019.5
	4	230826.2	174084.2	143531.8	143314.8	142614.2	140807.8
	16	67890.0	51201.2	42215.2	42151.4	41945.3	41414.0
	64	17755.8	13391.0	11040.9	11024.2	10970.3	10831.3
	256	4490.7	3386.8	2792.4	2788.2	2774.6	2739.4
	1024	1125.9	849.1	700.1	699.1	695.6	686.8
	SEQ-ON	661597.0	661597.0	661597.0	661597.0	661597.0	661597.0

Table A.18: *BinomialHeap* - *AVG* workload per worker, $Finitization = 8$

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	1302607.5	1302607.5	1302607.5	1302607.5	1302607.5	1302607.5
	4	521043.0	521043.0	521043.0	521043.0	521043.0	521043.0
	16	153247.9	153247.9	153247.9	153247.9	153247.9	153247.9
	64	40080.2	40080.2	40080.2	40080.2	40080.2	40080.2
	256	10137.0	10137.0	10137.0	10137.0	10137.0	10137.0
	1024	2541.6	2541.6	2541.6	2541.6	2541.6	2541.6
MKorat	1	1301253.0	1297793.5	1285164.0	1238141.0	1058220.0	357867.0
	4	520501.2	519117.4	514065.6	495256.4	423288.0	143146.8
	16	153088.5	152681.5	151195.7	145663.6	124496.4	42102.0
	64	40038.5	39932.1	39543.5	38096.6	32560.6	11011.2
	256	10126.4	10099.5	10001.2	9635.3	8235.1	2784.9
	1024	2539.0	2532.2	2507.6	2415.8	2064.8	698.2
	SEQ-ON	1303484.0	1303484.0	1303484.0	1303484.0	1303484.0	1303484.0

Table A.19: *SearchTree* - *AVG* workload per worker, $Finitization = 8$

	Equidistant candidates	*Infeasible ranges* 1	4	16	64	256	1024
MKorat$_{inf}$	1	2615692.5	2615692.5	2615692.5	2615692.5	2615692.5	2615692.5
	4	1046277.0	1046277.0	1046277.0	1046277.0	1046277.0	1046277.0
	16	307728.5	307728.5	307728.5	307728.5	307728.5	307728.5
	64	80482.8	80482.8	80482.8	80482.8	80482.8	80482.8
	256	20355.5	20355.5	20355.5	20355.5	20355.5	20355.5
	1024	5103.7	5103.7	5103.7	5103.7	5103.7	5103.7
MKorat	1	2615655.5	2615559.0	2615181.0	2613835.0	2609186.0	2592521.5
	4	1046262.2	1046223.6	1046072.4	1045534.0	1043674.4	1037008.6
	16	307724.1	307712.8	307668.3	307510.0	306963.0	305002.5
	64	80481.7	80478.7	80467.1	80425.6	80282.6	79769.8
	256	20355.3	20354.5	20351.6	20341.1	20304.9	20175.2
	1024	5103.7	5103.5	5102.7	5100.1	5091.0	5058.5
	SEQ-ON	2615692.5	2615692.5	2615692.5	2615692.5	2615692.5	2615692.5

Table A.20: *HeapArray* - *AVG* workload per worker, $Finitization = 8$

References

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[2] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, pages 2–23, 2005.

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

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[5] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 78–95, 2003.

[6] Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 176–185, 2005.

[7] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. Parallel randomized state-space search. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, ICSE '07, pages 3–12, 2007.

[8] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 64–73, 2007.

[9] Diego Funes, Junaid Haroon Siddiqui, and Sarfraz Khurshid. Ranged model checking. *ACM SIGSOFT Software Engineering Notes*, pages 1–5, 2012.

[10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 225–234, 2010.

[11] Patrice Godefroid. 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, Paris, France, 15-17 January 1997*, pages 174–186, 1997.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[13] G. Haugk, F. M. Lax, R. D. Royer, and J. R. Williams. The 5ess switching system: Maintenance capabilities. *AT T Technical Journal*, pages 1385–1416, 1985.

[14] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, pages 123–138, 2005.

[15] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 553–568, 2003.

[16] James C. King. Symbolic execution and program testing. *Commun. ACM*, pages 385–394, 1976.

[17] Xiaoming Li, Daryl Shannon, Jabari Walker, Sarfraz Khurshid, and Darko Marinov. Analyzing the uses of a software modeling tool. *Electr. Notes Theor. Comput. Sci.*, pages 3–18, 2006.

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

[19] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*, page 22, 2001.

[20] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. Parallel test generation and execution with Korat. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 135–144, 2007.

[21] Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating. *IEEE Trans. Software Eng.*, pages 1135–1139, 1987.

[22] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 504–515, 2011.

[23] Rui Qiu. Scaling and certifying symbolic execution. Unpublished PhD thesis proposal, 2016.

[24] J. H. Siddiqui and S. Khurshid. ParSym: Parallel symbolic execution. pages V1–405–V1–409, Oct.

[25] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, OOPSLA '12, pages 523–536.

[26] Junaid Haroon Siddiqui and Sarfraz Khurshid. PKorat: Parallel generation of structurally complex test inputs. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*, pages 250–259, 2009.

[27] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, ISSTA '10, pages 183–194.

[28] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. *Formal Methods in System Design*, pages 117–129, 2001.

[29] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *The Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11-15, 2000*, pages 3–12, 2000.

[30] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 196–205, 2004.

[31] Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. Memoized symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 144–154, 2012.

Vita

Nima Dini was born in Tehran, Iran. He received his Bachelors of Science degree in Software Engineering from University of Tehran. He applied to The University of Texas at Austin graduate program and started his graduate studies in Software Engineering in Fall 2014.

Email address: nima.dini@gmail.com

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.