The Thesis Committee for Cagdas Yelen
certifies that this is the approved version of the following thesis:

Backward-Korat: Improving Korat Search to Enable Backward Input Space Exploration

APPROVED BY

SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Christine L. Julien

Backward-Korat: Improving Korat Search to Enable Backward Input Space Exploration

by

Cagdas Yelen, 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 2017

Dedicated to my dear parents, Gulhan and Ismail Yelen,

and my beloved sister, Elif.

Acknowledgments

I would like to express the deepest appreciation to my supervisor Dr. Sarfraz Khurshid, who continuously supported and encouraged me in working on this thesis with great patience and extensive knowledge. Without his guidance and help, this thesis would not have been possible. Also, a thank you to Nima Dini for his great support and being a great coworker. Further, I would like to thank Dr. Christine Julien for reviewing this work and contributing with valuable feedback.

In addition, I would like to thank The University of Texas at Austin Cockrell School of Engineering and for their encouragement and support.

This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1319688 and CNS-1239498).

Backward-Korat: Improving Korat Search to Enable Backward Input Space Exploration

Cagdas Yelen, M.S.E.
The University of Texas at Austin, 2017

Supervisor: Sarfraz Khurshid

Automated test input generation plays an important role in increasing software quality. Exhaustively testing a program for all test inputs within a given bound helps check many corner cases that are easy to miss otherwise. *Korat* is a constraint solver and an automated testing framework for bounded exhaustive testing of Java programs. Korat uses a predicate method that describes desired inputs and a finitization bound to explore the space of all candidate inputs and generates the desired ones. Korat performs a systematic backtracking search for input space exploration based on pruning and isomorphism breaking. The Korat search gains part of its efficiency by monitoring executions of the given predicate on candidate inputs and creating new candidates based on the object fields accessed by the predicate during its execution. The Korat search has a default order for exploring the candidate inputs – the search always performs the same exploration for the same predicate and finitization.

Our thesis is that a different search order for the Korat search can enhance the efficacy of Korat. Specifically, we introduce the backward Korat, a novel approach to enable Korat to go backward in the search space. Our technique is built on the core of the traditional Korat search. The backward Korat can be applied to a variety of existing techniques including constraint-based data structure repair, parallel Korat, etc. We evaluate our approach using a standard suite of data structures. The experimental results show that the backward search works well and generates the same test inputs as the traditional search produces even though it performs slower compared to the forward search. Using the backward search and the traditional Korat search in tandem enables a new set of possible applications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software Testing is a technique for validating and verifying that a software or an application meets the requirements. It is usually a manual process that accounts for more than half of the total development and maintenance cost. Therefore, automated testing has been an interesting area for researchers. *Korat* is a constraint solver and a framework for constraint-based generation of structurally complex test inputs for Java programs, where the constraints are written as *imperative predicates* that characterize the desired properties of the generated test inputs [4, 23].

The foundation of our work is the Korat search for test input generation and constraint solving. Korat uses an imperative predicate, termed ***`repOK`*** [20], that specifies the desired properties of test inputs and a *finitization* that sets a bound to the input space [18]. Korat generates all *non-isomorphic* predicate inputs within the given finitization for which the `repOK` returns true. To perform the input generation, Korat searches the predicate's input space by exploring all non-isomorphic candidates (Section 3.1 will provide more details on how the traditional Korat search works and non-isomorphic data structures).

For the exploration of the input state space, Korat maps each test input to a *candidate vector* which consists of integers. The search starts with the first candidate vector and explores the search space until it invokes `repOK` on all non-isomorphic candidate vectors [4]. Korat explores the candidate vectors in a specific order for the repeated execution of the same input structure and size, which we call *forward Korat search.* The candidate vectors are lexicographically ordered based on the order of the values in the field domain and the `repOK` executions. The fact that the search is only able to run forward restricts Korat from being powerful in some applications. For example, in the context of data structure repair [7], the traditional Korat search fails to find a valid candidate if the search start from an invalid candidate that drops into the last *infeasible range* (Range of consecutive infeasible candidates) of the state space.

Our thesis is that the traditional Korat search can be improved to deliver the most efficient or desired results in some applications using *bidirectional* searching capabilities. We introduce the idea of *backward Korat search*, which improves Korat's state space exploration capability. The backward Korat search starts from a given candidate and explores the state space in the reverse order of the original Korat search, such that for any two consecutive candidates c_1 and c_2, such that in the original Korat search c_2 comes after c_1, the backward Korat would explore them in reverse order. A full backward search starts from the last candidate the original Korat would explore, and terminates at a candidate vector with all elements equal to zero, i.e., the candidate at which the original full Korat search would start. It uses a similar backtracking approach as the traditional Korat search to find the previous candidate vector at every step.

We make the following contributions:

- **Backward Korat search.** We introduce a novel approach for Korat to go backwards in the state space so that Korat is able to explore the candidates in the reverse direction and has an improved search capability as it gains bidirectional searching ability. Using the backward search and the traditional Korat search in tandem opens a whole new set of possible applications including improved constraint-based data structure repair, infeasible range construction and neighborhood search.

- **Technique.** We introduce multi-stage backtracking approach based on the traditional forward Korat search [4].

- **Evaluation.** We use a set of data structures to compare the backward search and the forward search. Evaluation results show that the backward Korat search generates the same test inputs as the traditional search produces although the cost of going backward is higher than going forward.

- **Potential applications.**

  - ***Improved constraint-based data structure repair.*** We enhance previous work [7] on data structure repair using Korat.

– ***Infeasible range construction.*** We utilize both the forward and the backward search to grow an infeasible range from a given infeasible candidate for pruning the state space exploration for the next execution of Korat [26].

– ***Neighborhood search.*** We introduce the idea of a neighborhood search using Korat. Given the number of valid structures to be found n, Korat starts from an initial candidate vector and explores the candidates in both directions until it finds n valid structures.

Chapter 2

Traditional Korat Example

In this chapter, we will explain a simple example which is taken from Korat's source code[1] and present the motivation of the backward Korat search.

```java
public class BinaryTree {
    public static class Node {
        Node left;
        Node right;
    }
    private Node root;
    private int size;
}
```

Figure 2.1: *BinaryTree* example.

Figure 2.1 shows the class declaration for binary tree example. Korat requires a predicate method, also called `repOK`, to check the validity of the generated structures and a finitization to define how to bound the input space. The sample `repOK` and finitization methods for the binary tree example are shown in Figures 2.2 and 2.3. For instance, to generate all non-isomorphic binary tree structures of size 3, we run Korat with finitization 3. For a given *finBinaryTree(3)*, Korat considers 63 candidate vectors and creates 5 valid ones in Figure 2.4.

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

```java
public boolean repOK() {
    if (root == null)
        return size == 0;
    // checks that tree has no cycle
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        if (current.left != null) {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    // checks that size is consistent
    return (visited.size() == size);
}
```

Figure 2.2: Predicate method repOK for the *BinaryTree* example.

```java
public static IFinitization finBinaryTree(int nodesNum, int minSize,
        int maxSize) {
    IFinitization f = FinitizationFactory.create(BinaryTree.class);
    IObjSet nodes = f.createObjSet(Node.class, nodesNum, true);
    f.set("root", nodes);
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    IIntSet sizes = f.createIntSet(minSize, maxSize);
    f.set("size", sizes);
    return f;
}
```

Figure 2.3: Finitization description for the *BinaryTree* example.

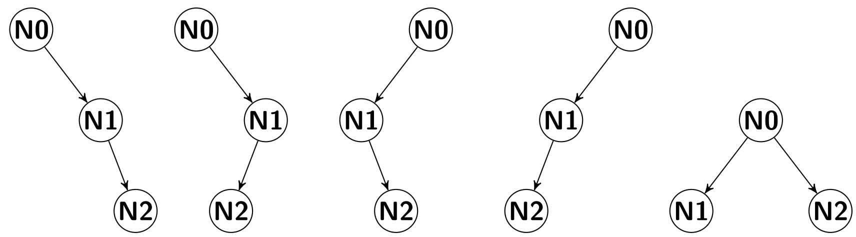


Figure 2.4: Valid binary trees with 3 nodes

For the binary tree example with 3 nodes, Korat creates the candidate vectors by mapping structures to an integer array by indexing their class fields in the following order: *T0.root, T0.size, N0.left, N0.right, N1.left, N1.right, N2.left, N2.right*. The search starts from the candidate vector set to all zeros, which is the first candidate based on the ordering of the field domain values, and Korat invokes the predicate method `repOK` to every candidate vector during the exploration. The fields accessed by the `repOK` invocation is monitored by Korat to find the next candidate by backtracking on the last accessed fields. The whole state space explored by Korat is shown in Figure 2.5 for binary tree of size 3. Candidate vectors and the field indexes accessed by `repOK` are shown in the figure for every candidate explored during the search. Valid structures that are generated by Korat are marked with *** and highlighted with green color. The direction of the exploration is from candidate vector 1, (0, 0, 0, 0, 0, 0, 0, 0), to candidate vector 63, (1, 0, 2, 3, 3, 0, 0, 0), which we call the forward Korat search.

We introduce the concept of the backward Korat search, which performs the search in the reverse direction. The backward Korat search makes it possible to start from any candidate that would be explored by the original Korat search within the search space and explore the state space backward. For example, if the backward Korat search start from candidate vector 34, (1, 0, 2, 0, 0, 3, 1, 0), the next candidate that Korat considers is candidate vector 33, (1, 0, 2, 0, 0, 3, 0, 3) and the search continues until it is terminated. The backward search stops when Korat reaches to the end candidate. The end candidate is the candidate vector set to all zeros unless it is specified by the user.

2.1 Valid candidate search

Korat supports bounding the search to specific start and end candidate vectors [23]. Given a Java predicate, a finitization, and an invalid candidate for which `!repOK()` holds, Korat starts the state space exploration until it finds a valid candidate, if any, within the finitization bound and it stops. However, in some certain cases, the backward search becomes more computationally efficient compared to the traditional Korat search. For example, if the search starts from the candidate 17 in Figure 2.5, Korat needs to consider 13 candidates to be able to find a valid one by going forward. On the other hand, going backward would decrease this number to only 1 candidate to be considered.

2.2 Search completeness

The valid candidate search problem becomes more than an inefficiency and a performance issue for some cases. Dini [7] used this approach in the

	Candidate vector	::	Index of fields accessed in repOK
1	0 0 0 0 0 0 0 0	::	0 1
2	1 0 0 0 0 0 0 0	::	0 2 3 1
3	1 0 0 1 0 0 0 0	::	0 2 3
4	1 0 0 2 0 0 0 0	::	0 2 3 4 5 1
5	1 0 0 2 0 1 0 0	::	0 2 3 4 5
6	1 0 0 2 0 2 0 0	::	0 2 3 4 5
7	1 0 0 2 0 3 0 0	::	0 2 3 4 5 6 7 1 ***
8	1 0 0 2 0 3 0 1	::	0 2 3 4 5 6 7
9	1 0 0 2 0 3 0 2	::	0 2 3 4 5 6 7
10	1 0 0 2 0 3 0 3	::	0 2 3 4 5 6 7
11	1 0 0 2 0 3 1 0	::	0 2 3 4 5 6
12	1 0 0 2 0 3 2 0	::	0 2 3 4 5 6
13	1 0 0 2 0 3 3 0	::	0 2 3 4 5 6
14	1 0 0 2 1 0 0 0	::	0 2 3 4
15	1 0 0 2 2 0 0 0	::	0 2 3 4
16	1 0 0 2 3 0 0 0	::	0 2 3 4 5 6 7 1 ***
17	1 0 0 2 3 0 0 1	::	0 2 3 4 5 6 7
18	1 0 0 2 3 0 0 2	::	0 2 3 4 5 6 7
19	1 0 0 2 3 0 0 3	::	0 2 3 4 5 6 7
20	1 0 0 2 3 0 1 0	::	0 2 3 4 5 6
21	1 0 0 2 3 0 2 0	::	0 2 3 4 5 6
22	1 0 0 2 3 0 3 0	::	0 2 3 4 5 6
23	1 0 0 2 3 1 0 0	::	0 2 3 4 5
24	1 0 0 2 3 2 0 0	::	0 2 3 4 5
25	1 0 0 2 3 3 0 0	::	0 2 3 4 5
26	1 0 1 0 0 0 0 0	::	0 2
27	1 0 2 0 0 0 0 0	::	0 2 3 4 5 1
28	1 0 2 0 0 1 0 0	::	0 2 3 4 5
29	1 0 2 0 0 2 0 0	::	0 2 3 4 5
30	1 0 2 0 0 3 0 0	::	0 2 3 4 5 6 7 1 ***
31	1 0 2 0 0 3 0 1	::	0 2 3 4 5 6 7
32	1 0 2 0 0 3 0 2	::	0 2 3 4 5 6 7
33	1 0 2 0 0 3 0 3	::	0 2 3 4 5 6 7
34	1 0 2 0 0 3 1 0	::	0 2 3 4 5 6
35	1 0 2 0 0 3 2 0	::	0 2 3 4 5 6
36	1 0 2 0 0 3 3 0	::	0 2 3 4 5 6
37	1 0 2 0 1 0 0 0	::	0 2 3 4
38	1 0 2 0 2 0 0 0	::	0 2 3 4
39	1 0 2 0 3 0 0 0	::	0 2 3 4 5 6 7 1 ***
40	1 0 2 0 3 0 0 1	::	0 2 3 4 5 6 7
41	1 0 2 0 3 0 0 2	::	0 2 3 4 5 6 7
42	1 0 2 0 3 0 0 3	::	0 2 3 4 5 6 7
43	1 0 2 0 3 0 1 0	::	0 2 3 4 5 6
44	1 0 2 0 3 0 2 0	::	0 2 3 4 5 6
45	1 0 2 0 3 0 3 0	::	0 2 3 4 5 6
46	1 0 2 0 3 1 0 0	::	0 2 3 4 5
47	1 0 2 0 3 2 0 0	::	0 2 3 4 5
48	1 0 2 0 3 3 0 0	::	0 2 3 4 5
49	1 0 2 1 0 0 0 0	::	0 2 3
50	1 0 2 2 0 0 0 0	::	0 2 3
51	1 0 2 3 0 0 0 0	::	0 2 3 4 5 6 7 1 ***
52	1 0 2 3 0 0 0 1	::	0 2 3 4 5 6 7
53	1 0 2 3 0 0 0 2	::	0 2 3 4 5 6 7
54	1 0 2 3 0 0 0 3	::	0 2 3 4 5 6 7
55	1 0 2 3 0 0 1 0	::	0 2 3 4 5 6
56	1 0 2 3 0 0 2 0	::	0 2 3 4 5 6
57	1 0 2 3 0 0 3 0	::	0 2 3 4 5 6
58	1 0 2 3 0 1 0 0	::	0 2 3 4 5
59	1 0 2 3 0 2 0 0	::	0 2 3 4 5
60	1 0 2 3 0 3 0 0	::	0 2 3 4 5
61	1 0 2 3 1 0 0 0	::	0 2 3 4
62	1 0 2 3 2 0 0 0	::	0 2 3 4
63	1 0 2 3 3 0 0 0	::	0 2 3 4

Figure 2.5: Candidates explored for *finBinaryTree(3)*

context of data structure repair. Given a faulty structure, Korat starts from the corresponding candidate vector and runs the forward search until it finds a valid structure. However, finding a valid candidate becomes impossible if the faulty structure drops to the last infeasible range, which is the range containing candidates [52, 63) in Figure 2.5. In scenarios similar to this, the backward search comes into play and becomes an essential feature for Korat as it enables Korat to perform the search in both directions.

Chapter 3

Technique

In this chapter, first of all, we will take a deeper look into the traditional Korat search (Forward Korat search) and how Korat backtracks using the field accesses of the predicate method. Thereafter, we will explain the backward Korat search algorithm and discuss how it differs from the forward search. Further, a fast-forwarding technique for the traditional Korat search and how the backward search benefits from the fast-forwarding will be discussed.

3.1 Forward Korat search

As it is stated in the previous chapters, Korat uses a predicate method and finitization to generate all non-isomorphic test inputs for which the predicate returns true. In this section, we will illustrate the forward Korat search algorithm in more detail.

Korat requires a finitization to be able to generate a bounded set of test inputs. In this way, Korat knows the set of values to be considered for a specific field of the target class. If we continue using the same example from chapter 2, there are two sets Korat considers for the *BinaryTree* of size 3 as they are shown in Figure 3.1.

$$fd(\text{BinaryTree.root}) = [null, N0, N1, N2]$$
$$fd(\text{Node.right}) = fd(\text{Node.left}) = fd(\text{BinaryTree.root})$$
$$fd(\text{BinaryTree.size}) = [\ 3\ ]$$

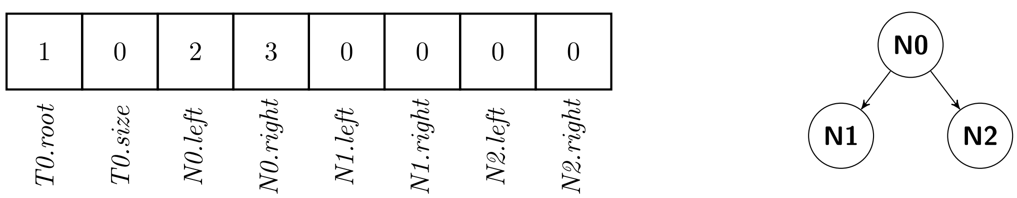


Figure 3.1: An example candidate vector with its corresponding tree structure, and the field domains for finitization 3.

These set of values for the corresponding fields are called field domains. A candidate vector is formed by using the index of an instance in its field domain for every field. In our example, the candidate vector has eight fields: the *BinaryTree* object has two fields, root and size, and each of the three Node objects have two fields, left and right. The field domain for the root field of the *BinaryTree* object and the both left and right fields of the Node object has four elements: *null, N0, N1, N2*. On the other hand, the field domain for the size field of the *BinaryTree* object has only one element, which is *3*. The state space of all potential candidates consists of $4 \cdot 1 \cdot (4 \cdot 4)^3 = 2^{14}$ candidates. Figure 3.1 shows one of the candidate vectors with its corresponding tree structure.

The exploration of the state space starts with the candidate vector set to all zeros. Korat sets the fields of the objects by mapping the values in vector to the corresponding elements. For every candidate, Korat invokes `repOK` to check if the current candidate is valid, or not. Korat monitors the fields that `repOK` accesses during the execution. These fields are ordered based on the order that `repOK` accesses them. In Figure 2.5, the right side of the figure shows the indices of fields accessed during `repOK` execution in an ordered manner.

After the execution of `repOK` is completed for the current candidate, Korat generates the next candidate by backtracking on the accessed fields during the previous execution. The basic idea is that Korat tries to increment the last accessed field to be able to generate the next candidate. If the domain index of the last accessed field exceeds the maximum domain index, Korat resets that index to zero and backtracks to the previous field in the ordered fields. This is repeated until the next candidate is found or the search is completed. Thanks to using backtracking, Korat prunes a huge portion of the state space. For example, for *BinaryTree* of size 3, Korat only considers 63 candidates instead of all 2^{14} possible candidates.

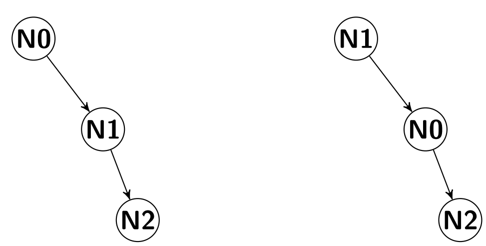


Figure 3.2: BinaryTree objects that are isomorphic to each other

	Candidate vector	::	Index of fields accessed in repOK
25	1 0 0 2 3 3 0 0	::	0 2 3 4 5
	1 0 0 3 0 0 0 0	::	0 2 3 6 7 1
	1 0 0 3 0 0 0 1	::	0 2 3 6 7
	1 0 0 3 0 0 0 2	::	0 2 3 6 7 4 5 1 ***
	1 0 0 3 0 1 0 2	::	0 2 3 6 7 4 5
	1 0 0 3 0 2 0 2	::	0 2 3 6 7 4 5
	1 0 0 3 0 3 0 2	::	0 2 3 6 7 4 5
	1 0 0 3 1 0 0 2	::	0 2 3 6 7 4
	1 0 0 3 2 0 0 2	::	0 2 3 6 7 4
	1 0 0 3 3 0 0 2	::	0 2 3 6 7 4
	1 0 0 3 0 0 0 3	::	0 2 3 6 7
	1 0 0 3 0 0 1 0	::	0 2 3 6
	1 0 0 3 0 0 2 0	::	0 2 3 6 7 4 5 1 ***
	1 0 0 3 0 1 2 0	::	0 2 3 6 7 4 5
	1 0 0 3 0 2 2 0	::	0 2 3 6 7 4 5
	1 0 0 3 0 3 2 0	::	0 2 3 6 7 4 5
	1 0 0 3 1 0 2 0	::	0 2 3 6 7 4
	1 0 0 3 2 0 2 0	::	0 2 3 6 7 4
	1 0 0 3 3 0 2 0	::	0 2 3 6 7 4
	1 0 0 3 0 0 2 1	::	0 2 3 6 7
	1 0 0 3 0 0 2 2	::	0 2 3 6 7
	1 0 0 3 0 0 2 3	::	0 2 3 6 7
	1 0 0 3 0 0 3 0	::	0 2 3 6
26	1 0 1 0 0 0 0 0	::	0 2

Avoided isomorphic copies

Figure 3.3: An example showing the non-isomorphism pruning

The search is optimized further by eliminating candidates that are isomorphic to each other. Figure 3.2 shows two isomorphic *BinaryTree* objects. Korat avoids generating isomorphic structures like these by applying non-isomorphism breaking. The detailed description of how Korat defines non-isomorphism and handles isomorphic candidates is given elsewhere [4]. Korat would explore 364 candidates for *BinaryTree* of size 3 if non-isomorphism breaking is disabled. However, the search only considers 63 candidates as shown in Figure 2.5. If we consider the 25^{th} and 26^{th} in that figure, there are 22 more candidates that Korat avoids to consider. Figure 3.3 illustrates avoided isomorphic copies between these two candidate vectors.

Pseudo-code for the forward Korat search algorithm is provided in Figure 3.4 [3]. The search starts with initVector, which is the candidate vector set to all zero, and continues until the whole state space is explored. Based on the `repOK` execution, backtracking on the accessed fields determine the next candidate. The algorithm shows how Korat exhaustively explores the search space of the predicate in an efficient way by pruning large portions of the search space and generating only non-isomorphic structures.

```
function forwardKorat(){

  int [] current = initVector;
  Stack accessedFields = new Stack();
  boolean isRepOK;

  do{
    (isRepOK, accessedFields) = current.repOK();

    if(isRepOK){
      reportValidCandidate(current);
    }

    int lastAccessedField = accessedFields.pop();
    while(!accessedFields.isEmpty()
      && current[lastAccessedField] >=
        nonIsoMax(current,accessedFields,lastAccessedField) ){
      current[lastAccessedField] = 0;
      lastAccessedField = accessedFields.pop();
    }

    if(!accessedFields.isEmpty()){
      current[lastAccessedField]++;
    }
  } while(current != lastVector && !accessedFields.isEmpty())
}
```

Figure 3.4: Forward Korat search algorithm pseudocode [3].

3.2 Backward Korat search

This section presents the backward Korat search, a novel approach for Korat to exhaustively explore the search space in the reverse direction. We introduce multi-stage backtracking, which is the foundation of the backward search and is inspired by backtracking that Korat uses for the traditional forward search. The main motivation of going in the reverse direction is to enable Korat to have a bidirectional search capability. Thus, Korat can be used in a more powerful way for some certain applications such as constraint-driven data structure repair [7] and distributed test input generation [23].

3.2.1 Multi-stage backtracking

We previously explained how Korat uses backtracking to exhaustively explore the search space for the traditional search algorithm in the forward direction. As it is also explained in the related paper [4], forward Korat backtracks on the accessed fields of the previous candidate to be able to find the next candidate. On the other hand, the backward Korat applies backtracking multiple times on the accessed fields of the previous candidate and the intermediate candidates, which are used for the intermediate steps of the algorithm, to generate the next candidate.

Figure 3.5 presents step by step how the next candidate is found by using multi-stage backtracking. To be consistent with the previous examples, we use the same data structure, a *BinaryTree* with 3 nodes, for the illustration of the approach. In the figure, candidate vector numbering is kept consistent

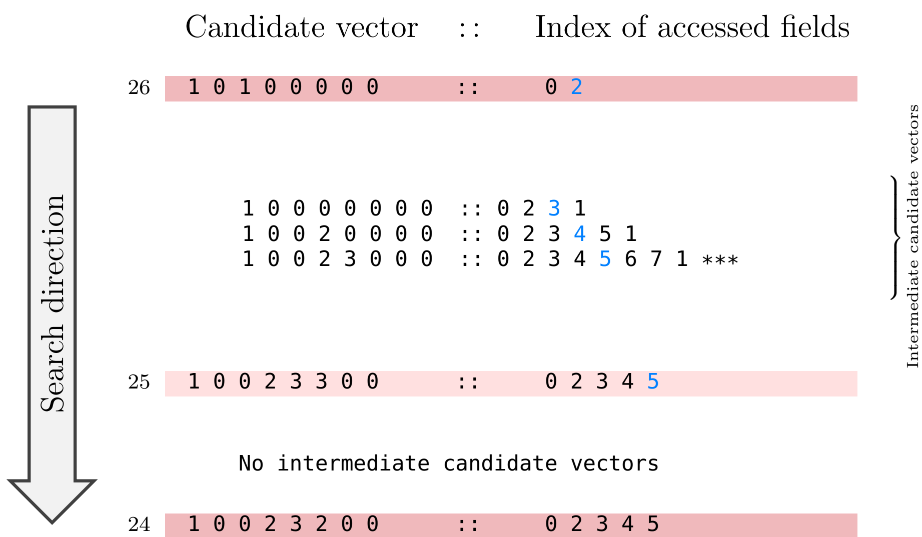


Figure 3.5: An example showing multi-stage backtracking

with the forward search space exploration. That is to say, Korat goes backward in the search space when it starts from the 26^{th} candidate and finds the 25^{th} and 24^{th} candidate vectors respectively.

The reason why it is called multi-stage backtracking is that there are intermediate candidates involved in the search. Therefore, Korat invokes the `repOK` and does backtracking on the fields `repOK` access to find the next candidate in the search space. In some cases, the next candidate is found without having any intermediate candidate. The difference between having intermediate candidates and not having them will be clear once we explain the example in Figure 3.5.

The basic idea behind the backward search is decrementing the last field access of the current candidate and incrementally setting the next field access of the intermediate candidates to its max value if they exist (By considering *isomorphism breaking* to avoid the isomorphic candidates). This is repeated until the next candidate is found based on the termination condition which will be explained later in this section. Once Korat decrements the last accessed field of the current candidate vector, it invokes the `repOK` on the new candidate. If the last accessed field of the new candidate is the same as the last accessed field of the previous candidate, then the next candidate is found. Otherwise, the new candidate is called an intermediate candidate since the termination condition is not satisfied for finding the next candidate. In this case, Korat sets the next accessed field of the intermediate candidate to its max value in accordance with the isomorphism breaking.

Figure 3.5 illustrates both cases of the backward search. If the search starts from the 26^{th} candidate, its last accessed field index is 2, which is colored with blue. Since the last accessed field is non-zero, the second field of the current candidate vector is decremented by one and the resulting candidate vector becomes (1, 0, 0, 0, 0, 0, 0, 0). Since the `repOK` execution on the resulting candidate touches more fields compared to the previous candidate, the resulting candidate is an intermediate candidate. The termination condition for finding the next candidate in the search space is that the accessed fields of the resulting candidate should be the same as the previous candidate. However, this is not the case for (1, 0, 0, 0, 0, 0, 0, 0) as it has more fields that are accessed after indices 0 and 2. Consequently, Korat sets the next accessed

```
function backwardKorat(){

  int [] current = initVector;
  List accessedFields = new ArrayList();
  boolean isRepOK;

  do{
    (isRepOK, accessedFields) = current.repOK();

    if(isRepOK){
      reportValidCandidate(current);
    }

    currentFieldIndex = accessedFields.removeLast();
    while(!accessedFields.isEmpty()
        && current[currentFieldIndex] == 0 ){
      currentFieldIndex = accessedFields.removeLast();
    }

    if(accessedFields.isEmpty()){
      break;
    }

    int lastAccessedField = accessedFields.getLast();
    current[lastAccessedField]--;

    int numberOfAccessedFields = accessedFields.size();
    while(true){
      accessedFields = current.repOK();

      if(numberOfAccessedFields == accessedFields.size()){
        break;
      }

      int accessedFieldIndex = acessedFields.get(numberOfAccessedFields);
      current[accessedFieldIndex] =
        nonIsoMax(current,accessedFields,accessedFieldIndex);
      numberOfAccessedFields++;
    }
  } while(current != lastVector && !accessedFields.isEmpty())
}
```

Figure 3.6: Backward Korat search algorithm pseudocode.

field of this candidate, which is index 3, to its max value. This process is repeated until (1, 0, 0, 2, 3, 3, 0, 0), the 25^{th} candidate, is found as it satisfies the termination condition. On the other hand, the search does not involve any intermediate candidates when it goes from the 25^{th} to the 24^{th} candidate. This is because the resulting candidate satisfies the termination condition after the last accessed field of the 25^{th} candidate is decremented. The number of `repOK` calls for going from the 26^{th} candidate to 25^{th} candidate is 4 while this number is only 1 for going from the 25^{th} candidate to 24^{th} candidate vector.

3.2.2 Implementation

Figure 3.6 shows the core algorithm of the backward Korat search. Within the *while* loop at line 7, Korat searches the candidates until it finds the end candidate. The algorithm is similar to the forward Korat search algorithm until line 25. At this point, Korat decrements the last accessed field and the multi-stage backtracking process begins. Since this procedure is explained in the previous section, we will not go into the details.

When the backward search encounters an intermediate candidate, Korat sets the next accessed field to its max value according to the isomorphism breaking to generate only non-isomorphic structures. This is operation is shown at line 36 of the same figure and *`nonIsoMax`* is the function that determines the max value for the accessed field to be set. The algorithm for this function is provided in Figure 3.7. The function checks all of the previously accessed fields to find the maximum of those that are from the same field do-

```
function nonIsoMax(currentCandidate, accessedFields, accessedFieldIndex){

  int maxInstanceIndex = accessedFieldIndex.getFieldDomain().size() - 1;

  int nonIsoMaxInstanceIndex = 0;
  for(int i=0; i< accessedFields.indexOf(accessedFieldIndex); i++){
    int currentAccessedFieldIndex = accessedFields.get(i);
    int activeInstanceIndex = currentCandidate[currentAccessedFieldIndex];

    if (nonIsoMaxInstanceIndex < activeInstanceIndex){
      nonIsoMaxInstanceIndex = activeInstanceIndex;
    }
  }

  if(nonIsoMaxInstanceIndex < maxInstanceIndex){
    return nonIsoMaxInstanceIndex + 1;
  }
  else{
    return maxInstanceIndex;
  }
}
```

Figure 3.7: nonIsoMax function for the backward Korat search

main with the current accessed field. If the maximum instance index of the previously accessed fields (`nonIsoMaxInstanceIndex`) is smaller than the maximum instance index of the field domain (maxInstanceIndex), the function returns `nonIsoMaxInstanceIndex + 1`. Otherwise, it returns the maximum instance index of the field domain. In this way, Korat avoids generating test inputs that are isomorphic to each other by keeping the difference less than 1 for the fields that belong to the same domain and setting them to the next instance in the field domain.

3.2.3 Fast-forwarding for Korat

The traditional forward Korat search starts exploring the search space from the candidate vector set to all zeros. In addition, the programmer optionally can provide a start candidate if the search needs to be started from a specific candidate vector. On the other hand, the backward Korat search requires a start candidate to begin the search as we do not have information about the state space prior to the exploration. For a complete exploration of the search space, the start candidate for the backward search should be the last candidate of the forward search. Thus, we need to execute a full forward search to get the next candidate so that we can run the backward search by using it as the start candidate. However, this approach is not feasible as we already have a full exploration of the search space when we go forward.

```
function fastForwardKorat(){

  int [] current = initVector;
  Stack accessedFields = new Stack();

  do{
    accessedFields = current.repOK();

    int lastAccessedField = accessedFields.pop();
    while(!accessedFields.isEmpty()
        && current[lastAccessedField] >=
          nonIsoMax(current, lastAccessedField) ){
      current[lastAccessedField] = 0;
      lastAccessedField = accessedFields.pop();
    }

    if(!accessedFields.isEmpty()){
      current[lastAccessedField]=nonIsoMax(current, lastAccessedField);
    }
  } while(current != lastVector && !accessedFields.isEmpty())
}
```

Figure 3.8: Fast-forwarding for Korat search pseudocode.

We introduce fast-forwarding for Korat, which is a novel approach to solve this problem. The algorithm pseudo-code is shown in Figure 3.8. The core of the algorithm is setting the last accessed field to its `nonIsoMax` value instead of incrementing it as the forward Korat search does. The rest of the algorithm is the same as the forward Korat search algorithm. Our approach targets to find the last candidate of the search space. The fast-forwarding provides huge amount of speedup over the execution cost of the forward search as it skips a lot of candidates that the forward search would explore since we are only interested in finding the last candidate.

Figure 3.9 show the state space exploration for *BinaryTree* of size 3 in fast-forwarding mode. As we can see from the figure, the fast-forwarding explores almost three times less candidates to find the last candidate as the search considers only 23 candidates compared to full forward search which considers 63 candidates for this specific subject.

	Candidate vector	::	Index of fields accessed in repOK
1	0 0 0 0 0 0 0 0	::	0 1
2	1 0 0 0 0 0 0 0	::	0 2 3 1
3	1 0 0 2 0 0 0 0	::	0 2 3 4 5 1
4	1 0 0 2 0 3 0 0	::	0 2 3 4 5 6 7 1 ***
5	1 0 0 2 0 3 0 3	::	0 2 3 4 5 6 7
6	1 0 0 2 0 3 3 0	::	0 2 3 4 5 6
7	1 0 0 2 3 0 0 0	::	0 2 3 4 5 6 7 1 ***
8	1 0 0 2 3 0 0 3	::	0 2 3 4 5 6 7
9	1 0 0 2 3 0 3 0	::	0 2 3 4 5 6
10	1 0 0 2 3 3 0 0	::	0 2 3 4 5
11	1 0 2 0 0 0 0 0	::	0 2 3 4 5 1
12	1 0 2 0 0 3 0 0	::	0 2 3 4 5 6 7 1 ***
13	1 0 2 0 0 3 0 3	::	0 2 3 4 5 6 7
14	1 0 2 0 0 3 3 0	::	0 2 3 4 5 6
15	1 0 2 0 3 0 0 0	::	0 2 3 4 5 6 7 1 ***
16	1 0 2 0 3 0 0 3	::	0 2 3 4 5 6 7
17	1 0 2 0 3 0 3 0	::	0 2 3 4 5 6
18	1 0 2 0 3 3 0 0	::	0 2 3 4 5
19	1 0 2 3 0 0 0 0	::	0 2 3 4 5 6 7 1 ***
20	1 0 2 3 0 0 0 3	::	0 2 3 4 5 6 7
21	1 0 2 3 0 0 3 0	::	0 2 3 4 5 6
22	1 0 2 3 0 3 0 0	::	0 2 3 4 5
23	1 0 2 3 3 0 0 0	::	0 2 3 4

Figure 3.9: Candidates explored in fast-forwarding mode for *finBinaryTree(3)*

3.2.4 Command-line options

The command-line options that are added to Korat for the backward search and fast-forwarding are shown in Table 3.1. *--back* allows Korat to execute the search in the backward direction. This option requires a start candidate, *--cvStart*[1], which is not mandatory for the forward Korat search. *--findEnd* activates the fast-forwarding mode for Korat and it does not require any other command-line options. It can only be used for the traditional forward Korat search.

Option	Description
--back	Backward Korat search mode
--findEnd	Fast-forwarding mode

Table 3.1: Backward-Korat command-line options

[1]http://korat.sourceforge.net/manual.html

Chapter 4

Evaluation

We evaluate the effectiveness of fast-forwarding for Korat and performance of the backward Korat search compared to the forward search on a suite of standard subjects that are chosen from Korat's default examples. This section describes the experiment procedure we designed to answer the following research questions:

RQ1. *What is the cost of fast-forwarding to the last candidate?*

RQ2. *How does backward Korat search performs compared to forward search?*

RQ3. *How effective fast-forwarding is compared to forward search?*

4.1 Study

We used seven subjects that are taken from Korat's open-source repository [1] as shown in Table 4.1. Some former studies on Korat used the same set of subjects in their evaluation [4, 7, 23, 31]. For each subject, we used the largest finitization for which the execution of forward Korat search is terminated within 30 seconds to create the tables that we show in this chapter.

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

These tables present an illustrative subset of our experimental results. We also provide another set of experiments with all subjects for finitizations 6, 8, and 10 to compare the results for different finitizations.

Subject(fin)
BinaryTree (BT)
BinomialHeap (BH)
DoublyLinkedList (DLL)
HeapArray (HA)
RedBlackTree (RBT)
SearchTree (ST)
SinglyLinkedList (SLL)

Table 4.1: Subjects used in the study

4.1.1 Execution platform

We run all the experiments on a machine with 2-cores, Intel® Core™ i5-4278U CPU at 2.60GHz, with 8GB of RAM, running OS X 10.11.6. We used Java 1.8.0 121 from Oracle®.

4.2 Results

Recall from Section 3.2.3 that the design goal of the fast-forwarding for Korat is to find the last candidate of the search space in order to execute backward Korat search. Tables 4.2 and 4.3 show the experiment results that compares fast-forwarding and forward Korat search in terms of the total number of candidates explored and the execution time by using the finitizations for which the forward Korat execution terminated within 30 seconds. In both cases, the search start from the candidate vector set to all zeros and stops when the last candidate of the search space is found. There are several points that are inferred from these tables:

		Fast-forwarding	Forward search
Subject(fin)	*BinaryTree(12)*	1,033,412	12,284,830
	BinomialHeap(9)	41	11,778,107
	DoublyLinkedList(11)	5	3,535,294
	HeapArray(9)	22	51,460,480
	RedBlackTree(10)	7	7,530,712
	SinglyLinkedList(11)	26	10,639,556
	SearchTree(9)	43,162	20,086,300

Table 4.2: Fast-forwarding vs. forward search: Total explored.

		Fast-forwarding	Forward search
Subject(fin)	*BinaryTree(12)*	1.343	8.492
	BinomialHeap(9)	0.19	15.48
	DoublyLinkedList(11)	0.154	5.424
	HeapArray(9)	0.186	10.906
	RedBlackTree(10)	0.205	9.709
	SinglyLinkedList(11)	0.148	6.298
	SearchTree(9)	0.327	13.281

Table 4.3: Fast-forwarding vs. forward search: Execution time.

- The amount of pruning that the fast-forwarding provides over the traditional forward Korat search is highly dependent on the subject type. For example, the fast-forwarding provides 12X reduction in terms of the total number of candidates explored for *BinaryTree* subject while the reduction is over 1,000,000X for *RedBlackTree*. The same thing can be stated for the execution time as the amount of speedup varies by subject.

- Given the execution times for fast-forwarding and forward search, the minimum speedup is more than 6X. Also, execution time reduction is proportional to the reduction in the number of candidates explored.

Tables 4.4 and 4.5 show the same set of experiments conducted with different finitizations. This time, the same subjects with finitizations 6, 8, and 10 are used for comparing the fast-forwarding (*ff*) and the forward search (*fw*) to see how the approach scales. The resulting key points from these tables:

- The fast-forwarding scales perfectly for some subjects, such as *DoublyLinkedList* and *RedBlackTree* as the total number of candidates explored stays the same for different finitizations.

- The amount of reduction in terms of total explored candidates increases as the finitization gets larger. For instance, the reduction is around 137X for *SinglyLinkedList* of size 6 while it is more than 70,000X for finitization 10.

	Finitization					
	6		8		10	
	ff	*fw*	*ff*	*fw*	*ff*	*fw*
BT	626	3,653	6,918	54,418	82,500	815,100
BH	28	42,815	36	1,323,194	44	150,727,471
DLL	5	776	5	17,166	5	562,823
HA	16	64,533	20	5,231,385	24	583,317,405
RBT	7	16,487	7	322,806	7	7,530,712
SLL	16	2,194	20	52,567	24	1,702,171
ST	1,154	45,233	12,638	2,606,968	149,684	155,455,872

Table 4.4: Fast-forwarding vs. forward search: Total explored.

	Finitization					
	6		8		10	
	ff	*fw*	*ff*	*fw*	*ff*	*fw*
BinaryTree	0.146	0.287	0.175	0.457	0.46	1.024
BinomialHeap	0.19	0.389	0.187	1.805	0.184	202.499
DoublyLinkedList	0.154	0.189	0.154	0.273	0.198	0.953
HeapArray	0.131	0.326	0.131	1.583	0.133	106.742
RedBlackTree	0.197	0.435	0.249	1.314	0.205	9.709
SinglyLinkedList	0.212	0.403	0.15	0.367	0.155	1.437
SearchTree	0.182	0.311	0.221	1.807	0.443	103.353

Table 4.5: Fast-forwarding vs. forward search: Execution time.

- For small finitizations, the execution time does not differ much as the internal Korat overhead becomes comparable to the state space exploration time. But, we can clearly see speedup when the finitizations increase.

Subject(fin)		Execution time	Total explored	New found
BinaryTree(12)	*fw*	8.492	12,284,830	208,012
	bw	15.132	13,608,740	
BinomialHeap(9)	*fw*	15.48	11,778,107	8,746,120
	bw	26.412	13,218,171	
DoublyLinkedList(11)	*fw*	5.424	3,535,294	3,535,027
	bw	9.916	4,213,909	
HeapArray(9)	*fw*	10.906	51,460,480	10,391,382
	bw	26.856	56,606,518	
RedBlackTree(10)	*fw*	9.709	7,530,712	260
	bw	16.013	11,750,872	
SinglyLinkedList(11)	*fw*	6.298	10,639,556	678,570
	bw	10.958	12,423,950	
SearchTree(9)	*fw*	13.281	20,086,300	4,862
	bw	20.966	22,601,403	

Table 4.6: The forward search vs the backward search for various finitizations.

Tables 4.6 and 4.7 shows the experiment results for comparing the traditional forward Korat search (*fw*) with the backward Korat search (*bw*) in terms of execution time, the total number of candidates explored and the new valid instances found. Similar to the fast-forwarding experiments, we used the finitizations for which forward Korat terminated within 30 seconds for Table 4.6 so that the internal overhead does not have a major impact on the execution time. On the other side, Table 4.7 shows the results of the experiments that are conducted by using finitizations 6, 8 and 10 with the same

		Finitization					
		6		8		10	
		fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	*BT*	0.287	0.247	0.457	0.602	1.024	1.671
	BH	0.389	0.453	1.805	3.224	202.499	332.218
	DLL	0.189	0.181	0.273	0.618	0.953	1.731
	HA	0.326	0.47	1.583	4.42	106.742	281.781
	RBT	0.435	0.377	1.314	1.336	9.709	16.013
	SLL	0.403	0.256	0.367	0.485	1.437	2.274
	ST	0.311	0.341	1.807	3.545	103.353	175.56
Total explored	*BT*	3,653	4,468	54,418	63,382	815,100	921,302
	BH	42,815	50,454	1,323,194	1,502,625	150,727,471	167,364,609
	DLL	776	1,004	17,166	21,339	562,823	678,839
	HA	64,533	73,745	5,231,385	5,812,641	583,317,405	636,346,249
	RBT	16,487	23,015	322,806	472,346	7,530,712	11,750,872
	SLL	2,194	2,828	52,567	64,315	1,702,171	2,013,450
	ST	45,233	54,364	2,606,968	2,980,582	155,455,872	172,744,382
New found	*BT*	132		1,430		16,796	
	BH	7,602		603,744		117,157,172	
	DLL	674		17,007		562,595	
	HA	13,139		1,005,075		111,511,015	
	RBT	20		64		260	
	SLL	203		4,140		115,975	
	ST	132		1,430		16,796	

Table 4.7: The forward search vs the backward search for finitizations 6, 8, 10.

set of subjects. For all experiments, the *Total explored* numbers are higher for backward search since we included the intermediate candidates for these counts. There are several points to discuss about these two tables:

- In all cases, the forward search and the backward search both find the same number of valid structures (*New found*), which proves the soundness of the backward search.

- The backward search is usually 2X slower than the forward search. This number gets close to 3X in some rare cases such as *HeapArray* of size 9. The slowdown is mostly caused by the fact that the backward Korat search explores more candidates than the forward search due to the intermediate candidates. Another important factor is that the number of fields considered during backtracking and non-isomorphism checking which is explained in Section 3.2.2.

- The amount of slowdown of the backward search over the forward search depends on the subject type and the finitization. The slowdown increases as the finitization becomes larger. This is an expected result and stems from the exhaustive bounded nature of Korat since the search space grows exponentially as the finitization increases.

		Fast-forwarding cost
Subject(fin)	*BinaryTree(12)*	1.343
	BinomialHeap(9)	0.19
	DoublyLinkedList(11)	0.154
	HeapArray(9)	0.186
	RedBlackTree(10)	0.205
	SinglyLinkedList(11)	0.148
	SearchTree(9)	0.327

Table 4.8: The cost of the fast forwarding for various finitizations.

		Finitization		
		6	8	10
Subject	*BinaryTree*	0.146	0.175	0.46
	BinomialHeap	0.19	0.187	0.184
	DoublyLinkedList	0.154	0.154	0.198
	HeapArray	0.131	0.131	0.133
	RedBlackTree	0.197	0.249	0.205
	SinglyLinkedList	0.212	0.15	0.155
	SearchTree	0.182	0.221	0.443

Table 4.9: The cost of the fast forwarding for finitizations 6, 8, 10.

Since the backward Korat needs the last candidate of the search space for a full exploration, we separately illustrate the cost of fast-forwarding to the last candidate. Tables 4.9 and 4.8 shows the results in seconds for our both experiment cases. By looking at these tables:

- The fast-forwarding to the last candidate of the search space takes less than 0.5 seconds except *BinaryTree* of finitization 12.
- Compared to the corresponding backward Korat search execution timings, the fast-forwarding cost becomes more insignificant as the finitization increases.

4.3 Answers to research questions

4.3.1 *RQ1.* What is the cost of fast-forwarding to the last candidate?

The overall cost of fast-forwarding technique is insignificant to a large extent compared to the execution time of the backward Korat search. It becomes even more insignificant as the finitization increases.

4.3.2 *RQ2.* How does backward Korat search performs compared to forward search?

The backward Korat search performs slower than the traditional forward Korat search as we expected due to the fact that it explores more candidates and performs more field accesses. The amount of slowdown is usually less than 2X while this number can go up to 3X for some cases.

4.3.3 *RQ3.* How effective fast-forwarding is compared to forward search?

The fast-forwarding technique has shown that it provides reduction over the forward Korat search in terms of the total number of candidates explored and the execution time to find the last candidate vector.

Chapter 5

Potential Applications

It is shown in Sections 2.1 and 2.2 that how the backward search improves the valid candidate search problem and enables Korat to perform the search in both directions. This chapter presents three potential applications that benefits from these improvements.

5.1 Improved constraint-based data structure repair

Data structure repair is a technique for recovering faulty data structures to enable the programs to execute successfully [5–8, 13, 16, 24–26]. Constrain-based data structure repair utilizes logical constraints to recover the faulty program state [5, 6]. *Juzi* tool [8, 9] introduced the idea of using a predicate method for repairing faulty structures during the program execution. Another prior work, *MKorat*, improved on this idea by memoizing infeasible ranges and repaired structure in the case of repeated repair scenarios [7].

Both Juzi and MKorat are based on the traditional forward Korat search as to find the next valid structure in the search space to repair the faulty structure. However, these approaches fail or perform in an inefficient way for some cases. Figure 5.1 illustrates how the backward search can improve

	Candidate vector	::	Index of accessed fields
35	1 0 2 0 0 3 2 0	::	0 2 3 4 5 6
36	1 0 2 0 0 3 3 0	::	0 2 3 4 5 6
37	1 0 2 0 1 0 0 0	::	0 2 3 4
38	1 0 2 0 2 0 0 0	::	0 2 3 4
39	1 0 2 0 3 0 0 0	::	0 2 3 4 5 6 7 1 ***
40	1 0 2 0 3 0 0 1	::	0 2 3 4 5 6 7
41	1 0 2 0 3 0 0 2	::	0 2 3 4 5 6 7
42	1 0 2 0 3 0 0 3	::	0 2 3 4 5 6 7
43	1 0 2 0 3 0 1 0	::	0 2 3 4 5 6
44	1 0 2 0 3 0 2 0	::	0 2 3 4 5 6
45	1 0 2 0 3 0 3 0	::	0 2 3 4 5 6
46	1 0 2 0 3 1 0 0	::	0 2 3 4 5
47	1 0 2 0 3 2 0 0	::	0 2 3 4 5
48	1 0 2 0 3 3 0 0	::	0 2 3 4 5
49	1 0 2 1 0 0 0 0	::	0 2 3
50	1 0 2 2 0 0 0 0	::	0 2 3
51	1 0 2 3 0 0 0 0	::	0 2 3 4 5 6 7 1 ***
52	1 0 2 3 0 0 0 1	::	0 2 3 4 5 6 7
53	1 0 2 3 0 0 0 2	::	0 2 3 4 5 6 7
54	1 0 2 3 0 0 0 3	::	0 2 3 4 5 6 7
55	1 0 2 3 0 0 1 0	::	0 2 3 4 5 6
56	1 0 2 3 0 0 2 0	::	0 2 3 4 5 6
57	1 0 2 3 0 0 3 0	::	0 2 3 4 5 6
58	1 0 2 3 0 1 0 0	::	0 2 3 4 5
59	1 0 2 3 0 2 0 0	::	0 2 3 4 5
60	1 0 2 3 0 3 0 0	::	0 2 3 4 5
61	1 0 2 3 1 0 0 0	::	0 2 3 4
62	1 0 2 3 2 0 0 0	::	0 2 3 4
63	1 0 2 3 3 0 0 0	::	0 2 3 4

Figure 5.1: Candidate vectors representing constraint-based data structure repair for *finBinaryTree(3)*

these cases. Given the data structures that are canonicalized with respect to `repOK`, the yellow candidate vectors represent the faulty structures that appear during the program execution and need to be repaired. For the 41^{st} candidate, the traditional repair algorithm runs the forward search and considers 10 candidate vectors until it finds a valid structure. On the other hand, the backward Korat only considers 2 candidates to provide a valid structure. Moreover, a candidate that is closer to the faulty structure in the search space will have a better approximation to the original structure and the repair process will have a higher chance of success. The situation is even worse when the faulty structure appears in the last infeasible range of the search space. For instance, the forward search is not able to find any valid structures if the erroneous data structure is canonicalized to the 57^{th} candidate since it falls in to the last infeasible range. The backward Korat search becomes very useful to solve cases similar to these.

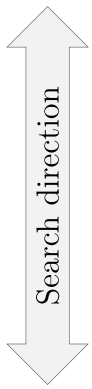
5.2 Infeasible range construction

An infeasible range is a range of consecutive infeasible candidates in the search space [7] as mentioned in the previous chapters. Prior studies showed that the concept of infeasible ranges can be useful for some set of applications that is built on Korat [7, 25, 26]. With the introduction of the backward Korat search, an infeasible range can be constructed by starting a bidirectional search, which is running Korat in the both directions, until the search encounters valid candidates (or it finds the first/last candidate vector) in the both directions.

```
function constructInfeasibleRange(startCandidate, predicate, fin){

  /* startCandidate is already a valid candidate, return */
  if(predicate.invoke(startCandidate))
    return;

  IKoratSearchStrategy stateSpaceExplorer =  new StateSpaceExplorer(fin);
  Object testCase = null;

  /* Going forward in the search space starting from the startCandidate */

  boolean isRepOK = false;
  stateSpaceExplorer.initStartCV(startCandidate);
  while(!isRepOK && !testCase.isLastCandidate()){
    testCase = stateSpaceExplorer.getNextCandidate();
    isRepOK = predicate.invoke(testCase);
    if(isRepOK)
      stateSpaceExplorer.reportCurrentAsValid();
  }


  /* Going backward in the search space starting from the startCandidate */

  isRepOK = false;
  stateSpaceExplorer.initStartCV(startCandidate);
  while(!isRepOK && !testCase.isFirstCandidate()){
    testCase = stateSpaceExplorer.getPrevCandidate();
    isRepOK = predicate.invoke(testCase);
    if(isRepOK)
      stateSpaceExplorer.reportCurrentAsValid();
  }
}
```

Figure 5.2: constructInfeasibleRange function for constracting an infeasible range

The algorithm pseudo-code is shown in Figure 5.2. Given an infeasible start candidate, *constructInfeasibleRange* function performs both forward and backward search to find all consecutive infeasible candidates. Figure 5.3 shows a visual example of the procedure. Assume that the start candidate is 20^{th} candidate, which is highlighted with yellow color. Forward search



	Candidate vector	::	Index of accessed fields
16	1 0 0 2 3 0 0 0	::	0 2 3 4 5 6 7 1 ***
17	1 0 0 2 3 0 0 1	::	0 2 3 4 5 6 7
18	1 0 0 2 3 0 0 2	::	0 2 3 4 5 6 7
19	1 0 0 2 3 0 0 3	::	0 2 3 4 5 6 7
20	1 0 0 2 3 0 1 0	::	0 2 3 4 5 6
21	1 0 0 2 3 0 2 0	::	0 2 3 4 5 6
22	1 0 0 2 3 0 3 0	::	0 2 3 4 5 6
23	1 0 0 2 3 1 0 0	::	0 2 3 4 5
24	1 0 0 2 3 2 0 0	::	0 2 3 4 5
25	1 0 0 2 3 3 0 0	::	0 2 3 4 5
26	1 0 1 0 0 0 0 0	::	0 2
27	1 0 2 0 0 0 0 0	::	0 2 3 4 5 1
28	1 0 2 0 0 1 0 0	::	0 2 3 4 5
29	1 0 2 0 0 2 0 0	::	0 2 3 4 5
30	1 0 2 0 0 3 0 0	::	0 2 3 4 5 6 7 1 ***

Figure 5.3: Constructed infeasible range by going forward and backward in the search space for *finBinaryTree(3)*.

and backward search stop when they hit 16^{th} and 30^{th} candidates, respectively. In this way, Korat can start from any infeasible candidate and find the boundaries of the corresponding infeasible range.

5.3 Neighborhood search

We talked about constraint-based data structure repair and how the backward Korat search improves on it in Section 5.1. Replacing an erroneous data structure with the next or previous valid candidate in the search space does not guarantee that it is the data structure that the program expects. Even though the repaired data structure is a valid candidate with respect to the `repOK` specification, the structure that the program expects can be different. This can cause a further faulty state in the program execution.

```
function neighborhoodSearch(n, startCandidate, predicate, fin){

  if(n <= 0)
    return;

  IKoratSearchStrategy stateSpaceExplorerFw =  new StateSpaceExplorer(fin);
  IKoratSearchStrategy stateSpaceExplorerBw =  new StateSpaceExplorer(fin);

  Object testCase = null;
  int count = 0;
  boolean isRepOK = false;

  stateSpaceExplorerFw.initStartCV(startCandidate);
  stateSpaceExplorerBw.initStartCV(startCandidate);

  while(true){

    /* Forward search checks the next candidate */

    testCase = stateSpaceExplorerFw.getNextCandidate();
    isRepOK = predicate.invoke(testCase);
    if(isRepOK){
      count++;
      stateSpaceExplorerFw.reportCurrentAsValid();
      if(count >= n || testCase.isLastCandidate())
        break;
    }

    /* Backward search checks the previous candidate */

    testCase = stateSpaceExplorerBw.getNextCandidate();
    isRepOK = predicate.invoke(testCase);
    if(isRepOK){
      count++;
      stateSpaceExplorerBw.reportCurrentAsValid();
      if(count >= n || testCase.isFirstCandidate())
        break;
    }
  }
}
```

Figure 5.4: neighborhoodSearch function for finding n valid candidates within the neighborhood of an initial candidate.

Candidate vector :: Index of accessed fields

Search direction

```
 7  1 0 0 2 0 3 0 0  :: 0 2 3 4 5 6 7 1 ***
 8  1 0 0 2 0 3 0 1  :: 0 2 3 4 5 6 7
 9  1 0 0 2 0 3 0 2  :: 0 2 3 4 5 6 7
10  1 0 0 2 0 3 0 3  :: 0 2 3 4 5 6 7
11  1 0 0 2 0 3 1 0  :: 0 2 3 4 5 6
12  1 0 0 2 0 3 2 0  :: 0 2 3 4 5 6
13  1 0 0 2 0 3 3 0  :: 0 2 3 4 5 6
14  1 0 0 2 1 0 0 0  :: 0 2 3 4
15  1 0 0 2 2 0 0 0  :: 0 2 3 4
 8  1 0 0 2 3 0 0 0  :: 0 2 3 4 5 6 7 1 ***
17  1 0 0 2 3 0 0 1  :: 0 2 3 4 5 6 7
18  1 0 0 2 3 0 0 2  :: 0 2 3 4 5 6 7
19  1 0 0 2 3 0 0 3  :: 0 2 3 4 5 6 7
20  1 0 0 2 3 0 1 0  :: 0 2 3 4 5 6
21  1 0 0 2 3 0 2 0  :: 0 2 3 4 5 6
22  1 0 0 2 3 0 3 0  :: 0 2 3 4 5 6
23  1 0 0 2 3 1 0 0  :: 0 2 3 4 5
24  1 0 0 2 3 2 0 0  :: 0 2 3 4 5
25  1 0 0 2 3 3 0 0  :: 0 2 3 4 5
26  1 0 1 0 0 0 0 0  :: 0 2
27  1 0 2 0 0 0 0 0  :: 0 2 3 4 5 1
28  1 0 2 0 0 1 0 0  :: 0 2 3 4 5
29  1 0 2 0 0 2 0 0  :: 0 2 3 4 5
30  1 0 2 0 0 3 0 0  :: 0 2 3 4 5 6 7 1 ***
31  1 0 2 0 0 3 0 1  :: 0 2 3 4 5 6 7
32  1 0 2 0 0 3 0 2  :: 0 2 3 4 5 6 7
33  1 0 2 0 0 3 0 3  :: 0 2 3 4 5 6 7
34  1 0 2 0 0 3 1 0  :: 0 2 3 4 5 6
35  1 0 2 0 0 3 2 0  :: 0 2 3 4 5 6
36  1 0 2 0 0 3 3 0  :: 0 2 3 4 5 6
37  1 0 2 0 1 0 0 0  :: 0 2 3 4
38  1 0 2 0 2 0 0 0  :: 0 2 3 4
39  1 0 2 0 3 0 0 0  :: 0 2 3 4 5 6 7 1 ***
```

Figure 5.5: Neighborhood search with n=*4* starting from the candidate vector 30 for *finBinaryTree(3)*.

One solution can be providing n different valid candidates to the program instead of one and let the program try all possibilities. In this way, the chance that the program terminates successfully becomes higher. This type of repair is achieved by performing a neighborhood search starting from the canonicalized invalid structure until n valid candidates are found. Figure 5.4 shows the algorithm pseudo-code. The `neighborhoodSearch` function applies the valid candidate search in both directions, forward and backward, and it terminates the search when n valid candidates are found. Figure 5.5 illustrates an example neighborhood search exploration for BinaryTree of size 3 with n = 4. In this example, the faulty structure corresponds to the 20^{th} candidate vector. The bi-directional search stops when it encounters 7^{th} and 39^{th} candidate vectors as the search is looking to find 4 valid structures.

Chapter 6

Related Work

Parallel Korat [23] introduced the first approach for parallel test generation and execution using Korat. Parallel Korat performs a sequential run of the complete search to create *equi-distant* candidate vectors, which allow creation of *ranged* problems that can be solved by independent workers in the future when the Korat search needs to be performed for the same search problem as before. PKorat [31] introduced an alternative parallel approach based on a work list that consists of work items that Korat search must explore. Dini et al. [7, 26] built on Parallel Korat [23] and introduced infeasible ranges that allow the re-execution of the Korat search to skip known ranges of consecutive invalid candidates to further optimize re-execution. The backward search can improve the way that Korat is run in distributed setting by letting individual workers start from the same candidate and execute the search in the reverse direction.

Generation of complex data structures has received much attention for bounded-exhaustive testing. TestEra [21] was among the first to generate tests up to the given bound based on declarative predicates written in Alloy. Korat [4] enabled a user to write (declarative) predicates in an imperative language. UDITA [10] supports predicates written in mixed-style (declarative and imperative). More recently, Kuraj et al. [19] introduced SciFe that uses an algebra of enumerators to make the generation incremental and parallelizable. The backward search enables Korat to have more flexible search capabilities based on user and domain needs.

Data structure repair is a technique for error recovery for errors in memory or persistent storage [1, 2, 13, 24]. While traditional techniques used dedicated repair routines, Demsky and Rinard [5] introduced the idea of using data structure integrity constraints as a basis for repair. The Juzi framework [16] introduced the use of imperative predicates as constraints for data structure repair using generalized symbolic execution [17] for systematic search. DSDSR [14] used dynamic symbolic (or concolic) execution [12, 28] for data structure repair. Tarmeem [36] and PBnJ [27] leveraged the SAT-based Alloy tool-set [15] to enable repair with respect to richer specifications. While rich post conditions allow more accurate repairs (than just `repOK` methods), they require check-pointing pre-states and generally admit less scalable solutions due to the higher complexity of the underlying constraint solving problem. Our application of Korat follows the spirit of Juzi but differs in that Korat does not require building or solving path conditions that are required in symbolic execution. Moreover, it further improves the repair efficiency and solves some corner cases with the introduction of backward search.

Fast-forwarding for Korat is a technique to explore the search space without considering all candidates. Misailovic et al. proposed fast-forwarding for PAR-OFF, to find a number of initial candidates for workers to run Korat in a distributed setting [23]. Our work differs in terms of the purpose of fast-forwarding and the technique that is used. While fast-forwarding for PAR-OFF targets to find random initial candidates for the workers to start individual executions, our algorithm's design goal is to find the end candidate. On the other part, their algorithm uses a random number of normal Korat steps and randomly truncates the field-access stack to find required number of candidates. Our algorithm does not involve any randomness and deterministically prunes a large number of candidates to find the last candidate vector of the search space.

Chapter 7

Conclusion

We presented the backward Korat, a novel approach to enable Korat to have an improved state space exploration capability. Our technique is built on the traditional forward Korat search by applying the backtracking approach in multiple stages to explore the search space in the reverse direction. The backward Korat can be used in a variety of applications including constraint-based data structure repair, PKorat, etc. We evaluated our algorithms, including the fast-forwarding approach in two different experimental settings. First, we compared the backward search with the traditional forward search to see the slowdown which is caused caused by intermediate candidates and additional field accesses that the backward search needs. Second, we evaluated the fast-forwarding algorithm to observe the cost of finding the last candidate of the search space. Our results showed that the backward search generates the same test inputs as the traditional search produces even though it is 2-3X slower compared to the forward search due to the reason we discussed.

Appendices

Appendix A

Evaluation Appendix

		Execution time	Total explored	New found
BinaryTree(12)	*fw*	8.492	12,284,830	208,012
	bw	15.132	13,608,740	

Table A.1: *BinaryTree* forward vs. backward search for finitization 12.

		Execution time	Total explored	New found
BinomialHeap(9)	*fw*	15.48	11,778,107	8,746,120
	bw	26.412	13,218,171	

Table A.2: *BinomialHeap* forward vs. backward search for finitization 9.

		Execution time	Total explored	New found
DoublyLinkedList(11)	*fw*	5.424	3,535,294	3,535,027
	bw	9.916	4,213,909	

Table A.3: *DoublyLinkedList* forward vs. backward search for finitization 11.

		Execution time	Total explored	New found
HeapArray(9)	*fw*	10.906	51,460,480	10,391,382
	bw	26.856	56,606,518	

Table A.4: *HeapArray* forward vs. backward search for finitization 9.

		Execution time	Total explored	New found
RedBlackTree(10)	*fw*	9.709	7,530,712	260
	bw	16.013	11,750,872	

Table A.5: *RedBlackTree* forward vs. backward search for finitization 10.

		Execution time	Total explored	New found
SinglyLinkedList(11)	*fw*	6.298	10,639,556	678,570
	bw	10.958	12,423,950	

Table A.6: *SinglyLinkedList* forward vs. backward search for finitization 11.

		Execution time	Total explored	New found
SearchTree(9)	*fw*	13.281	20,086,300	4,862
	bw	20.966	22,601,403	

Table A.7: *SearchTree* forward vs. backward search for finitization 9.

	Finitization					
	6		8		10	
	fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	0.287	0.247	0.457	0.602	1.024	1.671
Total explored	3,653	4,468	54,418	63,382	815,100	921,302
New found	132		1,430		16,796	

Table A.8: *BinaryTree* forward vs. backward Korat search for finitizations 6, 8, 10.

	Finitization					
	6		8		10	
	fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	0.389	0.453	1.805	3.224	202.499	332.218
Total explored	42,815	50,454	1,323,194	1,502,625	150,727,471	167,364,609
New found	7,602		603,744		117,157,172	

Table A.9: *BinomialHeap* forward vs. backward Korat search for finitizations 6, 8, 10.

	Finitization					
	6		8		10	
	fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	0.189	0.181	0.273	0.618	0.953	1.731
Total explored	776	1,004	17,166	21,339	562,823	678,839
New found	674		17,007		562,595	

Table A.10: *DoublyLinkedList* forward vs. backward Korat search for finitizations 6, 8, 10.

	Finitization					
	6		8		10	
	fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	0.326	0.47	1.583	4.42	106.742	281.781
Total explored	64,533	73,745	5,231,385	5,812,641	583,317,405	636,346,249
New found	13,139		1,005,075		111,511,015	

Table A.11: *HeapArray* forward vs. backward Korat search for finitizations 6, 8, 10.

	Finitization					
	6		8		10	
	fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	0.435	0.377	1.314	1.336	9.709	16.013
Total explored	16,487	23,015	322,806	472,346	7,530,712	11,750,872
New found	20		64		260	

Table A.12: *RedBlackTree* forward vs. backward Korat search for finitizations 6, 8, 10.

	Finitization					
	6		8		10	
	fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	0.403	0.256	0.367	0.485	1.437	2.274
Total explored	2,194	2,828	52,567	64,315	1,702,171	2,013,450
New found	203		4,140		115,975	

Table A.13: *SinglyLinkedList* forward vs. backward Korat search for finitizations 6, 8, 10.

	Finitization					
	6		8		10	
	fw	*bw*	*fw*	*bw*	*fw*	*bw*
Execution time	0.311	0.341	1.807	3.545	103.353	175.56
Total explored	45,233	54,364	2,606,968	2,980,582	155,455,872	172,744,382
New found	132		1,430		16,796	

Table A.14: *SearchTree* forward vs. backward Korat search for finitizations 6, 8, 10.

Bibliography

[1] Ext2 fsck manual page. `http://e2fsprogs.sourceforge.net`.

[2] Microsoft chkdsk manual page.

[3] Nazareno M. Aguirre, Valeria S. Bengolea, Marcelo F. Frias, and Juan P. Galeotti. Incorporating coverage criteria in bounded exhaustive black box test generation of structural inputs. In *Proceedings of the 5th International Conference on Tests and Proofs*, pages 15–32, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[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 Engi-*

neering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, pages 176–185, 2005.

[7] Nima Dini. MKorat: A novel approach for memoizing the korat search and some potential applications, 2016.

[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] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 855–858, New York, NY, USA, 2008. ACM.

[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] Ishtiaque Hussain and Christoph Csallner. Dynamic symbolic data structure repair. In *International Conference on Software Engineering*, pages 215–218, New York, NY, USA, 2010. ACM.

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

[16] 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.

[17] 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.

[18] Korat home page. `http://korat.sourceforge.net/index.html`.

[19] Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. Programming with enumerable sets of structures. In *Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 37–56, New York, NY, USA, 2015. ACM.

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

[21] 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.

[22] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *29th International Conference on Software Engineering*, pages 771–774, Washington, DC, USA, May 2007. IEEE Computer Society.

[23] 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.

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

[25] Milos Gligoric Sarfraz Khurshid Nima Dini, Cagdas Yelen. Incrementally solving imperative predicates using memoization. Under submission, 2017.

[26] Sarfraz Khurshid Nima Dini, Cagdas Yelen. Optimizing parallel korat using invalid ranges. In *The 24th International SPIN Symposium on Model Checking of Software*, 2017.

[27] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *European Conference on Object-oriented Programming*, pages 552–576, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *International Symposium on Foundations of Software Engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

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

[30] 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, Lan-*

guages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012, OOPSLA '12, pages 523–536.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. Contract-based data structure repair using Alloy. In *European Conference on Object-Oriented Programming*, pages 577–598, Berlin, Heidelberg, Jun 2010. Springer-Verlag.

Vita

Cagdas Yelen was born in Turkey. He received the Bachelor of Science degree in Electrical and Electrionics Engineering from Bogazici University. He applied to The University of Texas at Austin graduate program and started his graduate studies in Software Engineering in Fall 2015.

Email address: cagdasyelen@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.