# Efficient Solving of Structural Constraints

Bassem Elkarablieh
University of Texas at Austin
Austin, TX 78712
elkarabl@ece.utexas.edu

Darko Marinov
University of Illinois at Urbana
Urbana, IL 61801
marinov@cs.uiuc.edu

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

## ABSTRACT

Structural constraint solving is being increasingly used for software reliability tasks such as systematic testing or error recovery. For example, the Korat algorithm provides constraint-based test generation: given a Java predicate that describes desired input constraints and a bound on the input size, Korat systematically searches the bounded input space of the predicate to generate all inputs that satisfy the constraints. As another example, the STARC tool uses a constraint-based search to repair broken data structures. A key issue for these approaches is the efficiency of search.

This paper presents a novel approach that significantly improves the efficiency of structural constraint solvers. Specifically, most existing approaches use backtracking through code re-execution to explore their search space. In contrast, our approach performs checkpoint-based backtracking by storing partial program states and performing abstract undo operations. The heart of our approach is a light-weight search that is performed purely through code instrumentation. The experimental results on Korat and STARC for generating and repairing a set of complex data structures show an order to two orders of magnitude speed-up over the traditionally used searches.

**Categories and Subject Descriptors:** D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

**General Terms:** Algorithms, Reliability

**Keywords:** Backtracking, model checking, systematic testing

## 1. INTRODUCTION

Constraint solving lies at the heart of several approaches which are used increasingly effectively to improve software reliability, including symbolic execution [2, 11, 14, 17, 24], systematic testing [1, 4, 21], and data structure repair [6, 7, 13]. Modern software operates on complex data, so of increasing importance are approaches that can handle such data. We use the term *structural constraints* to refer to the constraints on the structure of object graphs that arise during a program's execution.

Two examples of solvers for such structural constraints are Korat [1, 20], for systematic test generation, and STARC [8], for data structure repair. A primary application of Korat is to enumerate test inputs from constraints that define properties of desired inputs. Such test inputs enable *bounded-exhaustive testing* that checks the code under test exhaustively within given bounds. Such testing was used previously to reveal faults in several real-world applications, including a fault-tree analyzer developed for NASA [26], an XPath compiler at Microsoft [25], and a web traversal application at Google [22]. The same solving also enables other reliability tasks, such as data structure repair, where violated assertions are used as a basis for mutating and repairing a program's erroneous states [7].

To generate/repair data structures, the solvers analyze a given imperative constraint, i.e., a Java predicate that represents constraints that define properties of the desired object graphs, and use a systematic search to explore the bounded space of the predicate to determine object graphs that satisfy the given constraint. For example, to generate data structures, Korat takes a bound on the size of the structure, along with the structural constraints, and generates all graphs (within the given bound) for which the predicate returns true. To explore the state space, Korat implements a search where program states are not stored for backtracking; instead, the state at a backtracking control point is re-created by re-executing the program from its beginning. This contrasts with stateful searches, such as those used in some model checkers, e.g., SPIN [12] and Java Pathfinder (JPF) [27], which store program states and retrieve them for backtracking. Both approaches have complementary strengths and traditionally model checkers are based on one of the two approaches [3, 10, 12, 27].

Efficiency of constraint solving is a key issue for wider adoption of solvers for structural constraints. For example, the current search in STARC enables repairing structures with up to a hundred thousand nodes. Improving the efficiency of STARC enables repairing larger structures that are typically found in real applications.

We present a new approach that significantly improves the efficiency of structural constraint solving. Our work builds on the Korat search [1, 20], but rather than using a backtracking that re-builds the structure and re-checks all the integrity constraints with every search candidate, our approach uses a checkpoint-based backtracking that employs efficient state manipulations based on (1) selective storing of program components and (2) abstract undo operations for retrieving the program state.

Our approach is based on two key insights: (1) the predicates check desired properties by traversing the given structures without mutating them; and (2) the traversals are over object graphs and often use standard worklist-based algorithms that track sets of visited nodes to prevent infinite traversals. The first insight allows us to define a minimal part of state to store, which reduces storage overhead. The second insight allows us to use our own library classes in place of the standard Java libraries, such as sets and lists, that are

*ISSTA'08,* July 20–24, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-59593-904-3/08/07 ...$5.00.

```
class SinglyLinkedList {
  Node header;
  Node getHeader() {...}

  int size;
  int getSize() {...}

  static class Node {
    Node next;
    Node getNext() {...}
  }

   // class invariant
   boolean repOk() {
    // If header is null, then size must be 0
    if (getHeader() == null) {
      return getSize() == 0;
    }

    // Acyclicity
    Set<Node> visited = new HashSet<Node>();
    Node p = getHeader();
    while (p != null) {
      if (!visited.add(p)) {
        return false;
      }
      p = p.getNext();
    }
    // Number of reachable nodes cached in size
    return getSize() == visited.size();
   }
```

**Figure 1: Class invariant for the `SinglyLinkedList`.**

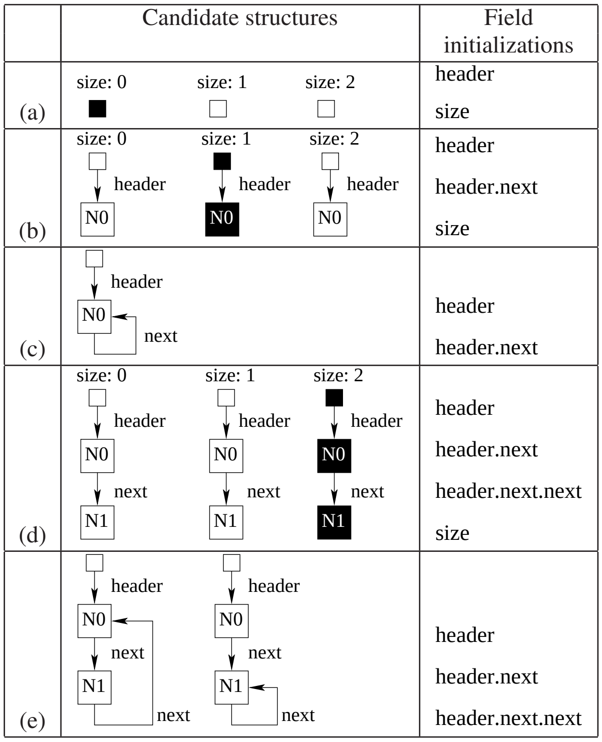

**Figure 2: The candidate structures considered by Korat for generating lists with up to two nodes. Valid lists are displayed in bold. The candidates are grouped according to the field initializations that Korat performs when generating them. For example, all the lists in row (a) require initializing the `header` and the `size` fields. The small boxes represent `SinglyLinkedList` objects and the large boxes represent `Node` objects.**

commonly used in graph traversals; in contrast with the standard libraries that optimize program execution, our libraries optimize backtracking and thus speedup the search.

This paper makes the following contributions:

- **Hybrid search for structural constraint solving:** We show how to combine the benefits of a re-execution-based search with the benefits of a checkpoint-based search.

- **Abstract undo operations:** We introduce undo operations that are performed at an abstract data type level to enable efficient backtracking.

- **Implementation:** We present an implementation for performing backtracking based on code instrumentation that executes on a standard Java Virtual Machine (JVM).

- **Evaluation:** We evaluated our approach on generation/repair of a suite of commonly used data structures. The experimental results show speedups of up to two orders of magnitude over Korat and an order of magnitude over STARC.

# 2. EXAMPLE

To illustrate our approach, we present the example of generating singly linked lists using Korat. This example illustrates how integrating our checkpoint-based search in Korat optimizes its performance and enables more efficient data structure generation. We start by describing the Korat search algorithm and present in detail all the steps for generating all singly linked lists with up to two nodes using Korat. We then illustrate how to generate the same lists more efficiently using our approach.

## 2.1 Singly linked list

Consider the skeleton of a singly linked list class in Figure 1. The inner class `Node` models the entries in a list. Each `Node` object has a `next` field which points to the next node in the list. Each singly linked list object has a `header` node which points to the first node in the list, and a `size` field which caches the number of unique nodes reachable from the header node along the `next` field. Each field is associated with an accessor method that returns its value.

The structural integrity constraints are: (1) acyclicity along `next` and (2) number of nodes reachable from the `header` field following `next` is cached in `size`.

The structural integrity constraints of a `SinglyLinkedList` can be represented as a Java predicate method which takes the structure as input and returns `true` if and only if the structure satisfies all the integrity constraints. Following Liskov [19], we term such a predicate method *repOk*, and for object-oriented programs, we term structural invariants as class invariants. The class invariant for the `SinglyLinkedList` class is shown in Figure 1.

## 2.2 Korat generation

To generate lists, Korat takes as inputs the class declaration, the `repOk` predicate, and a *finitization*, i.e., a bound on the number of objects of each type and a *domain* of values for each field. Korat starts by allocating the values of each field domain, and creates a *default* structure, i.e., a structure where all the fields are *uninitialized*. Korat then invokes `repOk` on the structure and initializes the fields to values from their domains as they are accessed during the execution of `repOk`. (Korat instruments the accessor methods and uses them to non-deterministically mutate the values of the fields.) When `repOk` terminates, Korat backtracks, mutates the last field

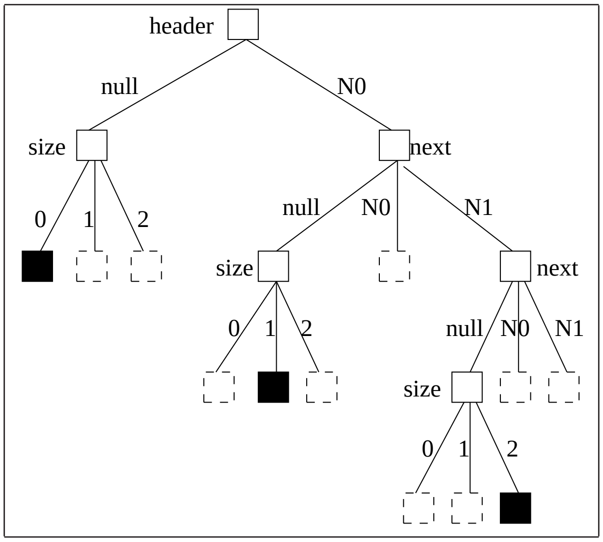

**Figure 3: The Korat search tree for generating all non-isomorphic lists with up to two nodes. The vertices represent field initializations, the edges represent values from the field domains. A path from the root to a leaf represents a candidate structure. Valid lists end at a bold leaf and invalid ones end at a dashed leaf.**

accessed by `repOk` to the next value in its domain, and re-executes `repOk`. Korat terminates when all the search is exhausted, i.e., all the values in the field domains are considered, and reports all the structures that resulted from a positive execution of `repOk`. Korat employs efficient pruning techniques to reduce the search space and only generates non-isomorphic structures [1].

To illustrate, consider generating all the singly linked lists with up to two nodes using Korat. The domains of the types are the sets [0, 1, 2] for the `size` field, and [null, N0, N1] for the `header` and `next` fields, where N0 and N1 are labels assigned for two objects of type `Node`. Figure 2 shows all the candidate structures and field initializations considered by Korat while generating the lists. The valid candidates are displayed in bold. Following the execution of `repOk`, Korat first accesses (and initializes) the `header` field to `null`—the first value in `header`'s domain— then initializes the `size` field to 0 and reports the structure as valid. Korat then re-executes `repOk` from the beginning, backtracking on the last field initialized during the execution of `repOk`, i.e., the `size` field, and sets it to 1—the second value of its domain. Once all the values of the `size` field are considered, Korat backtracks and assigns the `header` field to N0, which enables extending the list to generate the rest of the graphs. Note how Korat backtracks directly after `repOk` returns `false` which enables efficient pruning of the search space. Also note that Korat only considers N0, and not N1, for `header` to avoid generating isomorphic structures [1]. Korat's backtracking approach re-initializes all the fields that are accessed by `repOk` with every candidate structure. We next describe our backtracking approach using the same example.

## 2.3 Checkpoint-based generation

Another way to look at the Korat framework is by considering the search (execution) tree that Korat builds while generating the lists. Figure 3 shows the search tree. Each vertex in the tree represents a field initialization. Each edge represents a value from the field domain. For example, the root node corresponds to the `header` field, and the values considered for `header` are `null` and N0. Throughout the rest of the paper, we term the vertices as *choice points* as they represent expressions in the code where non-deterministic choices are performed, and the edges as *choices* as they represent the possible alternatives for field values.

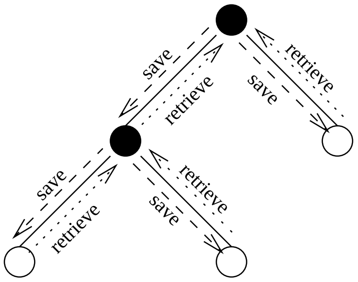

**Figure 4: The backtracking process. Black nodes represent choice points. White nodes represent termination points. Upon execution, the program state is saved. Upon backtracking, the program state is retrieved.**

Each path from the root to a leaf represents an execution of `repOk` and thus, a candidate structure. The leafs in bold represent executions of `repOk` that resulted in valid structures. In Korat, every execution of `repOk` traverses the tree from the root to a leaf. While this approach is sufficient to consider all the candidates, it (1) adds unnecessary field initializations with every path traversal, and (2) performs redundant checks of the structure properties. To illustrate, consider the three candidates at row (d) in Figure 2. These candidates only differ by the value of the `size` field. Since the last accessed field is the `size` field, (1) there is no need to re-initialize the list before changing the value of `size`, and (2) there is no need to check for the acyclicity of the list since a change in the `size` field only affects the check performed at line (`L16`) in `repOk`.

Our generation approach combines lightweight backtracking (Section 3.2) with incremental state storage and retrieval (Section 3.1) to optimize the performance of Korat. Rather than re-executing `repOk` (traversing the search tree from the root) for each candidate structure, our approach backtracks to the last choice point in `repOk` (the nearest parent in the tree), retrieves the program state at that statement, and proceeds with `repOk`'s execution (the next path in the tree starting at the nearest parent). Using this approach, only the last accessed field is re-initialized, and the only constraint checks performed are those checked after that field access. To illustrate, let us again consider the candidates at row (d) in Figure 2. Our algorithm backtracks to the check at line `L16`, mutates `size`, and only checks the size property of the structure.

For the singly linked list example, the checkpoint-based approach requires only 17 field initializations (which is equivalent to the number of edges in the search tree), whereas Korat requires 35 (which is equivalent to the sum of the number of edges for each path from the root vertex to a leaf). This difference in field initialization increases with the size of the structure. For example, for a list with 100 nodes, Korat performs 873,852 field initializations and generates the list in 532ms, whereas the checkpoint-based approach performs 15,452 in 54ms which is 10 times faster.

# 3. ALGORITHM

This section describes our approach for efficient backtracking. There are two key operations for backtracking: (1) switching the *execution control* to specific statements in a program and (2) storing/retrieving the *program state* at those statements. While this backtracking is advantageous for avoiding the overhead of rebuilding the program state as in re-execution-based backtracking, its performance is highly dependent on the efficiency of these two operations. For instance, our experiments with Korat show that the overhead imposed by a naive approach for storing/retrieving, which takes a snapshot of the heap at every choice point, is similar to that imposed by rebuilding the program state through re-execution for most structures.

We next describe our backtracking algorithm. We first show how to maintain the program state (but program counter), i.e., the stack, static, and heap memory, and then show our technique for efficiently maintaining the program counter when backtracking. We use Korat as an illustrative application of our approach, yet we point out that the approach is not limited to it (Section 6).

## 3.1 State storage and retrieval algorithm

Systematic search algorithms [1, 8, 14, 24] employ *choice points*, i.e., program statements where nondeterministic choices are performed on the search variables, and *termination points*, i.e., program statements that specify the end of a search path or a choice. For instance, the choice points in Korat are the field access statements in `repOk` where the search variables are the fields of the structure and the choices are the members of the field domains. The termination points in Korat are the `return` statements of `repOk` that declare a structure as valid or not.

Backtracking occurs between a termination point and a choice point. To maintain the correctness of a program execution, a backtracking approach must store the program state at each choice point, and upon backtracking, must retrieve the stored state and proceed with the next choice. To illustrate, Figure 4 gives an abstraction of the search process. Black nodes represent choice points, and white nodes represent termination points. As the program executes (following the dashed arrows in Figure 4), the search algorithm stores the state between choice points. Once a termination point is reached, the search algorithm backtracks to the last choice point (following the dotted arrows in Figure 4), retrieves the stored state and proceeds with the next choice.

Several approaches exist for state storage/retrieval [16, 31]. A simple, yet expensive, approach for storing the state is taking a snapshot of the heap at every choice point. This approach is not only expensive in terms of memory requirements but also inefficient as it stores a lot of redundant copies of parts of states. A more efficient approach for state storage is by using state comparisons [18, 30]. This approach efficiently stores the heap at the first choice point, and then incrementally updates it by comparing the state at the current choice point with the stored one.

We propose an alternative approach for state storage/retrieval. Rather than taking a snapshot of the heap at the choice point, or conceptually performing state comparisons to update the snapshot, we incrementally store the program state as the changes occur during execution. To enable efficient state retrieval, along with every stored change, we save a corresponding *undo command* [9] that enables retrieving the original state when the command is executed. Undo commands are implementations of the "Command" design pattern [9] where each command object saves the necessary information for undoing the effect of an action performed on the object. Undo commands have been previously used in software model checkers [23]. However, our use of the undo commands is different. Rather than performing the operations at the concrete heap level, we introduce *abstract undo commands* which perform these operations at an abstract object level. We elaborate on this idea later in this section.

```
// An interface for undo commands
public interface UndoCommand {
  public void execute();
}

// Declaration of the stack used for storing
// the undo commands
Stack<UndoCommand> undoStack = new Stack<UndoCommand>();

// Store method for saving the undo commands
// at a choice point
public void store() {
  saveUndoStack(undoStack);
  undoStack = new Stack<UndoCommand>();
}

// Retrieve method for restoring the program state
// at the backtracking target
public void retrieve() {
  undoStack = getLastUndoStack();
  while (!undoStack.isEmpty()) {
    UndoCommand uc = undoStack.pop();
    uc.execute();
  }
}
```

**Figure 5: Components for maintaining the program state. The `UndoCommand` interface defines the common behavior of all undo commands. The `undoStack` saves the undo commands that occur between two choice points in the program. The `store` method saves the undo stack and clears it while the `retrieve` method retrieves the last undo stack and executes its commands to restore the program state.**

The search algorithm maintains the undo commands in a stack which we term *undo stack*. As the program executes, the undo stack is populated with undo commands. At each choice point, the undo stack is saved. Upon backtracking, the last saved undo stack is retrieved, and its commands are executed to restore the state to the previous choice point. To illustrate, Figure 5 shows the implementation of the `store` method which is invoked at choice points, and the `retrieve` method which is invoked when backtracking.

### *3.1.1 Undo commands*

We next describe the undo commands. Undo commands are considered in the methods of interest for the search algorithm. For instance, in Korat's search, undo commands are inserted in `repOk` and any helper method invoked by `repOk` that accesses the target structure fields, i.e., contains choice points. Undo commands are inserted at method statements that cause a change in the program state. The statements of interest for inserting the undo commands are the following:

- store operations on the local variables,
- store operations on instance or static fields of a class, and
- method invocations.

We create undo command that (1) saves the original value of the modified object, and (2) enables retrieving the state of the modified object when the undo command is executed. We next describe the undo operation for each of above statements.

**Local variable stores**: Store operations on local variables are treated as field store operations. Since Java doesn't support pointer creations to elements in the JVM registers' stack, our approach replaces local variables with static fields and uses field undo commands (described in the next paragraph) to undo local changes.

```
// A static field to replace the local variable p
public static Node p;

// RepOk method with undo commands, and undoable containers
boolean repOk() {
  ...
  // The HashSet is replaced with an undoable hash set
  Set<Node> visited = new UndoableHashSet<Node>(undoStack);
  ...

  while (p != null) {
    ...

    // Undo command added to retrieve the value of p
    undoStack.push(new PUndoCommand(this, p));
    p = p.getNext();
  }
  ...
}
```

(a)

```
// The undo command for field accesses
public class PUndoCommand implements UndoCommand {
  SinglyLinkedList list;
  Node value;

  public PUndoCommand(SinglyLinkedList list, Node value) {
    this.list = list;
    this.value = value;
  }

  public void execute() {
    list.p = value;
  }
}
```

(b)

**Figure 6: (a) Undo commands inserted for the `repOk` method of the `SinglyLinkedList` class. Undo commands are added before variable changes, and containers are replaced with undoable containers. (b) An undo command class to restore the value of the variable `p` in `repOk`. The `execute` method reassigns `p` to its original value.**

This approach adds some overhead as *X*STORE instructions which access variables from stack memory are replaced with PUTSTATIC instructions which access variables from static memory. Note that our transformation replaces stack frames with static fields and as such cannot support recursive methods; to support recursion, our transformation would need to replace stack frames with (appropriately linked) heap objects

**Field stores**: Field store operations are the simplest to save and undo. Before each field store operation, we create an undo command object that takes as input the field's owner object and the field's value. When the command is executed, it reassigns the field to the saved value. To illustrate, consider the example of the `repOk` method for the `SignlyLinkedList` class in Figure 6(a). The method has one field to store, `p` (the `visited` variable is never re-assigned and thus it is not saved). To store `p`, we push a new instance of the `PUndoCommand` class (Figure 6(b)) which takes the list object and the field's value, onto the undo stack. Upon backtracking, when the `execute` method of the `PUndoCommand` is invoked, the field `p` retrieves its old value. The example in Figure 6(b) describes the general implementation of an undo command to restore the value of a field. We point out, however, that there is no need to save the owner object when restoring static fields.

Note that executing the undo commands upon backtracking restores the heap, static, and stack memory since local variables are transformed into static fields.

```
// A snippet of the UndoableHashSet class
public class UndoableHashSet<T> implements Set<T> {
  Stack<UndoCommand> undoStack;
  Set<T> container;
  ....

  public boolean add(T e) {
    if (container.add(e)) {
      undoStack.push(new AddUndoCommand<T>(container, e));
      return true;
    }
    return false;
  }
}
```

(a)

```
// Implementation for the abstract add undo command
public class AddUndoCommand<T> implements UndoCommand {
  Set<T> container;
  T val;

  public AddUndoCommand(Set<T> container, T val) {
    this.container = container;
    this.val = val;
  }

  public void execute() {
    container.remove(val);
  }
}
```

(b)

**Figure 7: Abstract undo commands on sets. (a) An `UndoableHashSet` class that associates each operation with its corresponding undo command. (b) An undo command for reversing the effect of the add operation on sets. If an item is added to the set, the undo command removes it when executed.**

**Method invocations**: A straightforward way to handle method invocations is to instrument the invoked method's code and add undo commands before changes to its local variables and fields accesses. We use this approach on `repOk` (the method of interest of Korat's search algorithm) and any helper method invoked by `repOk` which contains choice points. However, we treat other method invocations differently depending on the type of the method, its effect on the caller object, and the type of its caller object. We first check if the method is *pure*, i.e., does not mutate the state of its caller, and if so, there is no need to instrument the method's code. We then check the method's caller object type. If the caller object's type is that of a container type, i.e., its class implements the `java.util.Collection` or the `java.util.Map` interfaces, we use *abstract undo* commands to reverse the effect of the method on the container (Section 3.1.2). If the method's caller object type is not a container, we use the straightforward approach, i.e., instrument the method and add undo operations on its field accesses, but not the local variables.

### *3.1.2 Abstract undo operations*

Container types are widely used in Java programs. For example, `repOk` predicates are typically implemented as standard work-list algorithms that traverse the object graph, keep track of visited nodes, and check for the validity of the structural integrity constraints [20]. Collection classes provide powerful utilities for performing such traversals and checks, for example, a `LinkedList` object can be used for the work-list and a `HashSet` object can be used for saving the visited items. These classes maintain complex data structures to enable efficient operations, such as adding, removing, or checking the occurrence of an element. This makes

```
public boolean repOk() {
// L0:
//   store();
     Node header = getHeader();

// L1:
//   store();
     int size = getSize();

     System.out.println(header + "  " + size);

//   retrieve();
//   int index = Search.getTargetId();
//   TABLESWITCH \\(index)
//        0 : L0
//        1 : L1
//        2 : L2

// L2:
     return true;
}
```

**Figure 8: An example of the backtracking implementation. At each choice point, a label is added in addition to a call to the `store` method. At the branch target, a call to the `retrieve` method is added as well as a `TABLESWITCH` to enable backtracking to the program choice points.**

it expensive to store and retrieve their states using standard approaches. To illustrate, a snapshot approach requires iterating over the container elements at each choice point to save the state. A state comparison approach requires traversing the container to perform state comparisons. Even the undo approach that we presented in the previous section may be expensive due to the complex implementation of the operations on containers. For example, a `HashSet` implementation uses a `HashMap` which saves its elements in an internal array. Therefore adding undo commands for all the internal state changes involves several operations, especially for operations that dynamically resize the containers which include copying all the container elements into newly allocated memory.

We present an efficient way for undoing changes on containers. We perform the undo operations at the abstract level of the container rather than at the concrete container implementation. For example, instead of adding field undo commands in the implementation of the `addFirst` method of a `LinkedList` class, we add one undo command that reverses the effect of the `addFirst` method, i.e., the undo command calls the `removeFirst` method on the `LinkedList` object.

To apply this abstraction, we implement undoable versions of the container classes and replace all the instances of the concrete versions with the new ones, e.g., the `visited` variable in the `repOk` method in Figure 6(a). The undoable versions are simple adapters for the original containers where the methods' implementations push the appropriate undo command to the program undo stack. To illustrate, consider the code snippet of the `UndoableHashSet` class in Figure 7(a). The add method of this class adds an object to the internal wrapped `HashSet` object. If the `add` operation is successful, an `AddUndoCommand` object is created and pushed onto the undo stack. The implementation of the `AddUndoCommand` class is displayed in Figure 7(b). Instances of this class are constructed using the container and the object added to the container. The execute method simply removes the added object from the container.

Abstract undo operations achieve their efficiency by providing a way to undo the effect of complex operations that are frequently invoked and that involve large state changes.

## 3.2 Backtracking implementation

We next describe an implementation for efficient backtracking. We describe how to maintain the program counter and change its value between choice points to automatically switch the program control without special JVM support.

We start by identifying the *backtracking sources*, i.e., the program statements to backtrack from, and the *backtracking targets*, i.e., program statements to backtrack to. We then instrument the program to enable branching from the backtracking sources to the targets while restoring the state of the program at those targets.

The backtracking sources are the termination points of the program. For instance, the return statements in `repOk`. The backtracking targets are the choice points of the program. In Korat, the choice points are the accessor methods that perform nondeterministic choices to initialize the fields.

To enable efficient backtracking, we instrument the method under analysis, e.g., `repOk`, by adding labels at the backtracking targets, and `TABLESWITCH` instructions at the backtracking sources. The branch targets for the `TABLESWITCH` instructions are the labels inserted before the backtracking targets. The `TABLESWITCH` condition checks an integer value returned by search algorithm that identifies the label of the target choice point (this information is already maintained by the search algorithm). Note that this is a non-trivial use of table switches as the targets of the `TABLESWITCH` instructions occur at arbitrary points in the method code.

At the backtracking sources and targets, we also add a call to the `retrieve` and `store` methods described in Section 3.1 to maintain the program state when backtracking.

To illustrate the backtracking approach, consider the example in Figure 8 of a simplified `repOk` method that accesses two fields from the `SinglyLinkedList` and always returns `true`. The instructions added by the instrumentation are displayed in the commented portion of the code. Our use of the `TABLESWITCH` statements cannot be expressed in Java source and therefore, they are expressed in Java bytecode. The method in Figure 8 is simple and doesn't require adding undo commands.

The code example has two choice points. A label is added (L0 and L1) before each choice point, as well as a call to the `store` method which is used to save any undo commands performed before the choice point (in this case none). The added labels are the backtrack targets.

The method has one backtracking source which is the `return` statement. Before this statement, a label is added (L2) in addition to a call to the `retrieve` method which is used to execute the saved undo commands. A `TABLESWITCH` is also added before the return statement. The branching labels are L0 and L1, with the default label L2. The field domains we use in this example are [null, N0] for the `header` field, and [0, 1] for the `size` field. The output of executing `repOk` is as follows:

```
null  0
null  1
N0  0
N0  1
```

The execution works as follows. The first pass on `repOk` assigns `header` and `size` to `null` and 0 respectively. Before the method returns, the search algorithm returns 1 as the id for the last choice point, and the `TABLESWITCH` branches to label L1, assigns `size` to 1, and prints the values. At the next encounter of the return statement, the search algorithm returns 0 as the last field initialization since all the values in the `size` field domain are considered. The program backtracks to label L0, assigns `header` to `N0`, assigns `size` to 0, prints the values. The program then backtracks to label L1 and assigns `size` to 1. When all the choices are considered, the search algorithm returns 2 as the branch target, which causes the TABLESWITCH to branch to label L2, and the method's execution then terminates.

The above discussion illustrated backtracking within a single repOk method. However, normally as the complexity of the structural constraints increases, it is typical to represent the class invariant as multiple small helper methods with one executive repOk method that invokes the helper methods. Such cases might require backtracking to choice points that reside in the helper methods from the return statements in repOk. To handle such scenarios, the call sites of the helper methods are considered backtracking targets, and TABLESWITCH statements are added at the entry points of the helper methods to enable branching to the destination choice point. Upon backtracking from repOk, the control point is changed to the helper method's call site, the method is invoked, and then the TABLESWITCH at the entry of the method directs the control to the target choice point. Note that there is no need to restore the local variables at the target choice point, since restoring the values is automatically handled by executing the undo commands.

The described backtracking mechanism adds minimal overhead since it primarily adds table switches at method entries and return statements. Backtracking within repOk requires one switch, while backtracking for the cases of helper methods, requires two switches per invocation to reach the target choice point.

# 4. EVALUATION

## 4.1 Methodology

We evaluate the efficiency of our search by implementing it in both Korat, the test input generation tool described in Section 2, and STARC [8], a framework for data structure repair which employs a backtracking search based on static analysis.

We conduct two experiments using Korat. In the first experiment, we use Korat to exhaustively enumerate all the non-isomorphic structures of a small size, ranging from 6 to 15 nodes. In the second experiment, we use Korat to generate the first 100 non-isomorphic structures of a larger size, up to a few hundred nodes (generating all structures is infeasible due to the large number of such structures). For both experiments, we compare the generation time taken by the checkpoint-based approach with that taken by the re-execution-based approach originally used in Korat. The comparison with Korat shows up to 8.3X speedup in execution time when using the checkpoint-based approach for generating small structures, and up to 117X speedup for generating large structures.

To demonstrate the efficiency of the backtracking technique, in addition to the generation time, we compare the number of field initializations performed by the checkpoint-based approach with those performed by Korat.

For data structure repair, we use STARC for repairing structurally complex subjects with tens of thousands of nodes. We perform the experiment using the checkpoint-based approach and using the original search implemented in STARC. A comparison of the results shows up to 56X speedup for the checkpoint-based search when repairing structures with 100,000 nodes and 100 faults.

## 4.2 Subjects

The evaluation is performed on ten subject structures that have been previously used to evaluate various approaches in testing and error recovery [8, 20]. The structures are the following: singly linked list, the acyclic structure described in Section 2; sorted list, structurally identical to a singly linked list but with sorted elements; binary and n-ary trees; search trees, similar to binary trees but with ordered elements; red-black and avl trees, implementations of balanced search trees, with red-black trees having complex constraints on the colors of the nodes along the paths from the root [5]; doubly linked list, an implementation of the java.util.LinkedList library; disjoint set, a linked-based implementation of the fast union-find data structure [5]; and heap array, an implementation of a priority queue. For each of the subjects, we implemented the repOk methods that represent the structural constraints.

## 4.3 Experimental results

We next describe the experimental results. All experiments used a 1.7 GHZ Pentium M processor with 512 MB of RAM.

### *4.3.1 Generation using Korat*

**Experiment 1: Generating all small structures**

In this experiment, we study the efficiency of the checkpoint-based backtracking approach by using it to generate all the non-isomorphic structures within a given small scope (up to 15 nodes). We refer to the original implementation as Korat. Table 1 shows the generation results. The table displays the generation time in milliseconds taken by Korat and the checkpoint-based approach for generating eight of the subject structures. We do not show the results for the singly and doubly linked lists as both approaches can generate all the lists with less than 20 nodes within 50ms. The table also contains a comparison between the number of field initializations performed by Korat and the checkpoint-based approach when generating the structures. We use this comparison to illustrate the efficiency of the backtracking algorithm. In addition to the generation time and the number of field initializations, the table reports the number of undo commands performed by the checkpoint-based approach to maintain the program state.

The generation time results show that for the reported subjects, the checkpoint-based approach is faster than Korat with speedup ranging from 1.2X for heap array, to 8.3X for search tree. The results also show that the speedup factors increase with the size of the structure. For example, in the red-black tree example, the speedup ratio increases from 3.2X when generating trees with 6 nodes to 6.6X when generating those with 9 nodes.

To study the speedups obtained by using the checkpoint-based backtracking approach, we evaluate the reduction in the number of field initializations, as well as the number of undo operations required by the backtracking approach to maintain the program state. The field initialization results in Table 1 show that for all the studied subjects the checkpoint-based approach reduces the number of field initializations required by Korat to generate the structures. The reduction ratio ranges from 6.3X for heap array to 28X for red-black tree. In comparison with the number of field initializations performed by Korat, the number of undo operations is smaller by approximately one order of magnitude.

Note that the speedup factors vary among different subjects with the same size. For example, the speedup factor for generating a heap array with 10 nodes is 2.6X whereas that of the avl tree is 6.4X. This variation is due to the field initialization ratio and the number of undo operations performed. For example, the field initialization ratio for generating the heap array is 9.5X whereas that of the avl tree is 22X. Similarly, the number of undo operations performed in the heap array example is approximately 8X less than the number of field initializations of Korat, whereas in the avl tree it is approximately 18X less.

The above results show that as the structure size increases, the reduction ratio in the number of field initializations increases which leads to an increase in the speedup factor. The next experiment shows how this result enables our approach to achieve up to two orders of magnitude speedup factors over Korat.

| *Subject structure* | *Size* | *Candidate structures* | *Valid structures* | *Time(ms)* | | *Speedup* | *Field initializations* | | *Ratio* | *Undo operations* |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | *Korat* | *checkpoint-based* | | *Korat* | *checkpoint-based* | | |
| Binary tree | 9 | 210,444 | 4,862 | 901 | 330 | **2.7X** | 3,556,640 | 241,074 | **14.7X** | 282,525 |
| | 10 | 815,100 | 16,796 | 3,404 | 892 | **3.8X** | 15,366,812 | 921,312 | **16.6X** | 1,064,599 |
| | 11 | 3,162,018 | 58,786 | 13,590 | 2,784 | **4.8X** | 65,809,076 | 3,535,028 | **18.6X** | 4,036,961 |
| | 12 | 12,284,830 | 208,012 | 57,263 | 10,055 | **5.7X** | 279,823,708 | 13,608,752 | **20.5X** | 15,386,477 |
| Avl tree | 7 | 43,485 | 136 | 350 | 110 | **3.2X** | 801,754 | 50,364 | **15.9X** | 77,540 |
| | 8 | 182,930 | 288 | 1,021 | 262 | **3.8X** | 3,840,910 | 207,966 | **18.4X** | 309,971 |
| | 9 | 611,592 | 440 | 3,615 | 650 | **5.5X** | 14,013,240 | 686,794 | **20.4X** | 980,053 |
| | 10 | 2,036,700 | 660 | 11,627 | 1,802 | **6.4X** | 50,271,572 | 2,265,072 | **22.1X** | 3,107,883 |
| Red black | 6 | 23,327 | 140 | 341 | 108 | **3.2X** | 623,331 | 30,876 | **20.1X** | 69,096 |
| | 7 | 101,104 | 280 | 931 | 232 | **4.0X** | 3,130,680 | 135,274 | **23.1X** | 315,071 |
| | 8 | 449,270 | 576 | 3,681 | 657 | **5.6X** | 15,812,723 | 614,107 | **25.7X** | 1,510,765 |
| | 9 | 2,061,202 | 1,220 | 19,885 | 3,014 | **6.6X** | 81,246,102 | 2,899,485 | **28.0X** | 7,578,187 |
| Search tree | 6 | 98,693 | 924 | 661 | 161 | **4.1X** | 1,798,082 | 115,354 | **15.5X** | 202,942 |
| | 7 | 755,833 | 3,432 | 3,862 | 791 | **4.8X** | 15,782,074 | 864,476 | **18.2X** | 1,445,712 |
| | 8 | 5,797,298 | 12,870 | 30,261 | 4,779 | **6.3X** | 136,077,730 | 6,524,130 | **20.8X** | 10,459,953 |
| | 9 | 44,537,298 | 48,620 | 276,551 | 32,960 | **8.3X** | 1,159,010,940 | 49,493,134 | **23.4X** | 76,579,239 |
| N-ary tree | 6 | 314,515 | 7,752 | 1,151 | 371 | **3.1X** | 5,631,007 | 376,824 | **14.9X** | 497,258 |
| | 7 | 2,084,503 | 43,263 | 8,682 | 1,933 | **4.5X** | 43,319,152 | 2,436,750 | **17.7X** | 3,115,667 |
| | 8 | 13,776,898 | 246,675 | 64,799 | 11,737 | **5.6X** | 326,280,532 | 15,806,535 | **20.6X** | 19,709,837 |
| | 9 | 90,939,373 | 1,430,715 | 446,970 | 66,391 | **6.7X** | 2,419,176,805 | 102,814,365 | **23.5X** | 125,611,705 |
| Sorted list | 12 | 98,227 | 13 | 491 | 140 | **3.5X** | 2,117,998 | 106,430 | **19.9X** | 131,077 |
| | 13 | 212,902 | 14 | 1,055 | 250 | **4.2X** | 49,105,46 | 229,298 | **21.4X** | 278,534 |
| | 14 | 458,648 | 15 | 2,210 | 430 | **5.1X** | 11,266,473 | 491,429 | **22.9X** | 589,831 |
| | 15 | 982,921 | 16 | 5,108 | 801 | **6.3X** | 25,617,641 | 1,048,471 | **24.4X** | 1,245,192 |
| Heap array | 7 | 14,512 | 4,147 | 90 | 72 | **1.2X** | 343,484 | 53,992 | **6.3X** | 66,952 |
| | 8 | 138,025 | 21,814 | 521 | 382 | **1.4X** | 3,319,853 | 440,829 | **7.5X** | 528,867 |
| | 9 | 2,981,757 | 231,710 | 4,083 | 1,829 | **2.2X** | 28,078,660 | 3,312,910 | **8.4X** | 3,984,900 |
| | 10 | 9,745,451 | 2,015,168 | 38,801 | 14,635 | **2.6X** | 286,898,609 | 30,059,909 | **9.5X** | 35,782,329 |
| Disjoint sets | 6 | 31,801 | 5,040 | 171 | 80 | **2.1X** | 597,707 | 35,422 | **16.8X** | 181,636 |
| | 7 | 202,832 | 40,420 | 911 | 301 | **3.1X** | 6,410,217 | 322,743 | **19.8X** | 1,814,650 |
| | 8 | 2,085,332 | 362,880 | 10,565 | 2,473 | **4.2X** | 74,576,822 | 3,266,149 | **22.8X** | 19,958,710 |
| | 9 | 23,458,671 | 3,628,800 | 126,718 | 26,391 | **4.8X** | 936,490,880 | 36,288,280 | **25.8X** | 239,501,176 |

**Table 1: Results for generating all non-isomorphic structures with up to 15 nodes. The table shows a comparison of the generation time between Korat and our approach. The tabulated times are in milliseconds. It also compares the number of field initializations performed by the two approaches. The results show up to 10 times speedup for the checkpoint-based approach over Korat.**

**Experiment 2: Generating few large structures**

We study the efficiency of the checkpoint-based backtracking approach on generating larger structures with hundreds of nodes. While enumerating all small structures is important for exhaustive bounded testing of programs, it is prudent to test the applications on larger inputs to capture bugs that are not easily detected with small tests.

We generate the first 100 non-isomorphic structures as it is infeasible to generate all of them. Table 2 shows the generation results. The items of this table are similar to those in Table 1. For structures with only 1 non-isomorphic candidate of a given size, such as the singly and doubly linked lists, we generate 100 structures within a size range. For the balanced trees, and the ordered data structures, Korat's execution does not terminate within 20 minutes for generating 2 structures with 100 nodes due to the constraints on the data elements, and thus, we drop these structures from the comparison.

The generation time results show up to two orders of magnitude in the speedup factor of the checkpoint-based approach over Korat. Note how the speedup factor increases when generating larger structures. For example the speedup factors of the binary tree example increased from 5.7X for generating trees with 12 nodes to 69.2X when generating trees with 400 nodes.

The improvement in the speedup factor is due to the improvements in the reduction factor of the number of field initializations performed. For example, for the binary tree example, the field initialization reduction ratio increases from 20X to 721X when generating trees with 400 nodes. This is an expected result as for each candidate structure, Korat re-executes `repOk` from the beginning and reinitializes all the fields accessed by `repOk`. As the size of the structure increases, the number of candidates increases, as well as the number of field initializations required for generation.

As for the undo operations, the number of operations performed when generating large structures is still approximately one order of magnitude less than the number of field initializations performed by Korat, and thus adds minimal overhead to the overall performance.

The experimental results illustrate how using the lightweight backtracking approach with the abstract undo operations optimizes Korat's generation time by reducing the number of field initializations without introducing a comparable number of undo operations. These results show the applicability of the approach to various complex structures ranging from a few nodes to hundreds of nodes.

### 4.3.2 Repair using STARC

STARC [8] is a framework for assertion-based repair of complex data structures. STARC aims at automatically repairing data structure corruptions that occur at runtime, enabling programs to recover from corruption errors and proceed with their executions. Given a corrupt structure and a `repOk` method that describes the structural constraints, STARC mutates the corrupt structure and transforms it into one that satisfies `repOk`. STARC builds on previous work on data-structure repair [7] which uses a re-execution-based backtracking search based on symbolic execution and introduces a static analysis that guides the search to efficiently repair large complex structures. Unlike the Korat search which aims at enumerating all structures of a given size, STARC aims at searching for the first structure that satisfies the given constraints starting from the corrupt structure.

| *Subject structure* | *Size* | *Candidate structures* | | *Time(ms)* | | *Speedup* | | *Field initializations* | | *Ratio* | | *Undo operations* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | *Korat* | *checkpoint-based* | | | *Korat* | *checkpoint-based* | | | |
| Singly linked list | 401-500 | 125,751 | | 10,455 | 150 | **69.7X** | | 42,043,252 | 126,752 | **331.6X** | | 254,502 |
| | 501-600 | 180,901 | | 18,076 | 210 | **86.0X** | | 72,541,902 | 182,102 | **398.3X** | | 365,402 |
| | 601-700 | 246,051 | | 29,531 | 302 | **97.7X** | | 115,070,552 | 247,452 | **465.0X** | | 496,302 |
| | 701-800 | 321,201 | | 46,210 | 410 | **112.7X** | | 171,629,202 | 322,802 | **531.6X** | | 647,202 |
| Doubly linked list | 401-500 | 377,247 | | 42,281 | 511 | **82.7X** | | 251,501,000 | 379,247 | **663.1X** | | 382,744 |
| | 501-600 | 542,697 | | 64,002 | 721 | **88.7X** | | 434,161,200 | 545,097 | **796.4X** | | 549,294 |
| | 601-700 | 738,147 | | 112,532 | 1,210 | **93.0X** | | 688,941,400 | 740,947 | **929.8X** | | 745,844 |
| | 701-800 | 963,597 | | 178,352 | 1,807 | **98.7X** | | 1,027,841,600 | 966,797 | **1,063.1X** | | 972,394 |
| Binary tree | 100 | 50,678 | | 2,103 | 93 | **22.6X** | | 9,745,867 | 51,585 | **188.9X** | | 52,222 |
| | 200 | 111,428 | | 8,662 | 201 | **43.0X** | | 41,780,717 | 112,635 | **370.9X** | | 113,072 |
| | 300 | 182,178 | | 19,528 | 350 | **55.8X** | | 100,105,567 | 183,685 | **544.9X** | | 183,922 |
| | 400 | 262,928 | | 39,457 | 570 | **69.2X** | | 188,720,417 | 26,4735 | **712.8X** | | 264,772 |
| N-ary tree | 100 | 65,516 | | 3,816 | 181 | **32.3X** | | 19,163,888 | 66,651 | **287.5X** | | 67,888 |
| | 200 | 141,066 | | 15,181 | 250 | **60.7X** | | 80,656,838 | 142,601 | **565.6X** | | 143,638 |
| | 300 | 226,616 | | 30,710 | 331 | **92.7X** | | 190,479,788 | 228,551 | **833.4X** | | 229,388 |
| | 400 | 322,166 | | 64,483 | 551 | **117.0X** | | 354,632,738 | 324,501 | **1,092.8X** | | 325,138 |
| Disjoint sets | 100 | 22,154 | | 1,902 | 130 | **14.6X** | | 4,806,571 | 22,725 | **211.5X** | | 550,052 |
| | 200 | 74,304 | | 15,393 | 320 | **48.1X** | | 29,853,071 | 75,375 | **396.0X** | | 2,829,852 |
| | 300 | 156,454 | | 57,143 | 811 | **70.4X** | | 91,139,571 | 158,025 | **576.7X** | | 7,839,652 |
| | 400 | 268,604 | | 151,181 | 1,602 | **94.3X** | | 204,666,071 | 270,675 | **756.1X** | | 16,579,452 |

**Table 2: Results for generating 100 structures with hundreds of nodes. The tabulated times are in milliseconds. The results show up to 100 times speedup for the checkpoint-based approach over Korat.**

We integrated our approach in STARC and used it to repair five subject structures. We refer to the original search as STARC. Table 3 shows the repair time taken by STARC and the checkpoint-based approach for repairing large structures with up to 100,000 nodes and with 100 faults. The repair time results show a speedup for the checkpoint-based approach over STARC that ranges from 12X when repairing trees with 10,000 nodes and 100 faults to 56X when repairing disjoint sets with 100,000 nodes.

The speedup factors increase with the size of the structure. For example, for the n-ary tree example, the speedup factor increased from 12X when repairing a corrupt structure with 10,000 nodes to 26X when repairing a structure with 100,000 nodes. This increase in the speedup factor relates to the nature of the backtracking search used in STARC. The original search in STARC is re-execution-based and thus every mutation in the structure requires traversing the structure from the root to check the class invariant. As the size (number of faults) of the structure increases, such traversals become more expensive. The checkpoint-based approach, on the other hand, incrementally checks for the class invariant and requires a single traversal of the structure to perform all the mutation.

The experiment on repair indicates that integrating the checkpoint-based approach in STARC scales its performance for repairing larger data structures more efficiently.

| *Subject structure* | *Size* | *Faults* | | *Time(ms)* | | *Speedup* |
|---|---|---|---|---|---|---|
| | | | | *STARC* | *checkpoint-based* | |
| Doubly linked list | 10,000 | 100 | | 5,083 | 166 | **30X** |
| | 100,000 | 100 | | 58,744 | 1,255 | **46X** |
| Binary tree | 10,000 | 100 | | 1,852 | 133 | **13X** |
| | 100,000 | 100 | | 36,808 | 1,615 | **22X** |
| Nary tree | 10,000 | 100 | | 1,722 | 143 | **12X** |
| | 100,000 | 100 | | 52,463 | 1,963 | **26X** |
| Avl tree | 10,000 | 100 | | 9,924 | 203 | **32X** |
| | 100,000 | 100 | | 117,092 | 2,144 | **54X** |
| Disjoint sets | 10,000 | 100 | | 7,421 | 155 | **47X** |
| | 100,000 | 100 | | 64,906 | 1,141 | **56X** |

**Table 3: Results for repairing five complex structures with up to 100,000 nodes and 100 faults.**

# 5. DISCUSSION

## 5.1 Overhead of our approach

The checkpoint-based approach removes the overhead of rebuilding the program state from scratch after each backtracking operation (as in re-execution-based backtracking). However, it introduces the overhead of maintaining the program state by saving and executing the undo commands. The experimental results in Section 4 show that the checkpoint-based approach reduces the number of field initializations performed in Korat, while introducing a set of undo commands. The number of such commands, however, is relatively an order of magnitude less than the reduction in field initializations, resulting in faster generation time.

A key reason for this improvement relates to the nature of the `repOk` methods used by Korat and STARC to build and explore the search space. Such methods are typically pure methods, i.e., they check for the structural properties without mutating the structure. Thus, we expect state changes between 2 consecutive choice points to be minimal, which results in less undo operations to retrieve the state and in turn a better performance than code re-execution.

We point out, however, that this improvement in execution time is not observed in every application of the backtracking search. For example, when generating structures with 2 or 3 nodes using Korat, where building the state requires a hand full of field initializations, the re-execution-based search performed better than the checkpoint-based search since the overhead introduced for maintaining the state exceeds that for rebuilding the program state.

In general, for applications that maintain a relatively small state, or with large number of state changes between choice points, a re-execution-based search more likely to have a better performance.

## 5.2 Soundness of our approach

The search presented in this paper is purely performed through code instrumentation of the class under analysis. This entails some modifications in the structure of the class, including adding fields to replace local variables when performing undo operations. Such modifications might affect the soundness of the approach on some Java programs. For example, consider a `repOk` method that reflectively accesses the fields of its declaring class. Such method might have a different behavior because of the changes performed on the structure of the class.

Other factors that might break the soundness of the approach are the abstract undo operations. These operations might not be equivalent to the exact inverse of the corresponding forward operations. While executing these commands reverses the effect at the abstract container level, the internal structure of the container might have changed. For example, adding and removing methods in a balanced tree might involve some reordering operations that result in a different structure. A `repOk` method that accesses the internal implementation of the container might have a different behavior after running the undo operation.

To retrieve the state of a method stack frame, our approach replaces local variable accesses with static field accesses and performs undo operations at the field level. The current implementation only supports acyclic call graphs. Extending this approach to handle recursive calls requires changing the instrumentation to allocate a stack frame for each method invocation in order to keep track of the field values at the different recursion levels.

### 5.3 Abstract analysis on containers

Although presented in the context of a Java implementation, the proposed technique is not limited to Java or its containers. Undo operations can be applied to different languages and on any (well-specified) container written in that language. For example, similar containers can be implemented for the C++ standard library.

We believe that extending current program analyses to handle libraries opens more opportunities for reasoning about programs. For example, we have previously introduced *abstract symbolic execution* in a workshop paper [15] which treats containers as symbolic objects. By treating containers symbolically, we were able to test programs that manipulate such containers, an analysis that was not feasible if the implementation of the container was to be considered.

## 6. RELATED WORK

Our work builds on the Korat search. The key difference between our approach and Korat is the backtracking algorithm. Rather than using a backtracking that (1) re-builds the structure and (2) re-checks all the integrity constraints with every search candidate, our approach uses a checkpoint-based backtracking approach that employs efficient state manipulations to minimize the redundant rebuilds and checks in Korat improving its performance.

Java PathFinder (JPF) is a general purpose model checker that has also been used as a solver for imperative predicates [14]. JPF performs stateful model checking of (multi-threaded) Java programs. It implements a custom Java Virtual Machine (JVM) that, unlike the standard JVM, enables non-deterministic re-executions of Java programs to, theoretically, cover all the possible executions of a program. JPF has been applied for testing data structure implementations both at concrete and abstract levels [28–30]. The Korat search has also been implemented in JPF using *lazy initialization* of object fields [14, 28]. While this direct implementation of Korat using JPF provides a stateful implementation of Korat, the resulting search is significantly slower than the original Korat search. A reason for the slowdown is the unnecessary overhead introduced by the generality of JPF.

Our approach in this paper is inspired by our experience of optimizing JPF, and differs from JPF in three key ways, which make our approach significantly faster than JPF (as well as Korat) for structural constraint solving. First, we implement a lightweight backtracking mechanism by performing code instrumentation (Section 3.2) rather than implementing a custom JVM, which is required by JPF. Second, we perform efficient incremental state saving. Rather than hashing the entire program state, and comparing it with the next state, we incrementally save state changes and their corresponding *undo commands* as the changes occur in the program. While storing states incrementally (as “deltas”) is a known technique in explicit-state model checking [3], we perform it at an *abstract* level. Third, to perform state backtracking, we execute the *abstract* undo commands stored when saving the program state, which retrieves the program state.

Xu et al. [31] recently proposed an approach for efficient checkpointing and replay for Java programs. Using code instrumentation, their tool generates two versions of the program, one for checkpointing and the other for replay. Given a sequence of method calls, their approach uses a static analysis to determine the choice points and the minimal amount of state to save. Our work is closely related to this work as the presented backtracking is based on code instrumentation, however, it differs in its applications, the mechanisms for maintaining the program state, as well as the implementation for changing the program counter.

Several existing applications can utilize the presented search. For example, one candidate application is symbolic execution. For symbolic execution, the search target is all the program paths, the choice points are the branch statements, the termination points are the return statements in a program, and the program variables represent the state to be maintained.

## 7. CONCLUSIONS

This paper presented a novel approach for efficient backtracking. Our approach was used to optimize the search performance of two frameworks for systematic test input generation (Korat) and automatic data structure repair (STARC).

The heart of the backtracking approach is a selective state storage as well as abstract undo operations that efficiently maintain the program state when backtracking. Experiments on generating data structures with complex structural integrity constraints show that the checkpoint-based approach achieves up to two orders of magnitude improvement in generation time over Korat and one order of magnitude in repair time over STARC.

While we have described our approach in the context of test generation and repair, it also enables efficient search in other contexts, such as symbolic execution. As the experimental results indicate, the performance benefits increase as the size of the structures being generated (or repaired) increases. We believe that integrating our approach with existing work on generation and repair of large object graphs will enable such approaches to more efficiently check strong properties of real systems.

## Acknowledgments

We would like to thank Shadi Abdul Khalek, Michele Saad, and the anonymous referees for their helpful comments on the paper. This work was partially supported by the NSF awards #CCF-0702680, #CNS-0615372, #CNS-0613665, and #IIS-0438967, and a gift from Microsoft.

## 8. REFERENCES

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[2] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th SPIN Workshop on Software Model Checking*, 2005.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[4] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Sept. 1976.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[6] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[7] B. Elkarablieh, I. García, Y. L. Suen, and S. Khurshid. Assertion-based repair of structurally complex data. In *Proc. 22th Conference on Automated Software Engineering (ASE)*, Nov. 2007.

[8] B. Elkarablieh, S. Khurshid, D. Vu, and K. McKinley. Starc: Static analysis for efficient repair of complex data. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2007.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[10] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Paris, France, Jan. 1997.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[12] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[13] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *Proc. 12th SPIN Workshop on Software Model Checking*, 2005.

[14] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[15] S. Khurshid and Y. L. Suen. Generalizing symbolic execution to library classes. In *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Lisbon, Portugal, Sept. 2005.

[16] J. L. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, Aug 1993.

[17] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[18] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[19] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[20] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[21] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.

[22] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with Korat. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, Sept. 2007.

[23] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *FSE03*, pages 267–276, 2003.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2005.

[25] K. Stobie. Advanced modeling, model based test generation, and Abstract state machine Language (AsmL). Seattle Area Software Quality Assurance Group, `http://www.sasqag.org/pastmeetings/asml.ppt`, Jan. 2003.

[26] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[27] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[28] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004.

[29] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for red-black trees using abstraction. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 414–417, New York, NY, USA, 2005. ACM.

[30] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, New York, NY, USA, 2006. ACM.

[31] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 85–94, 2007.