Efficiently Generating Structurally Complex Inputs with Thousands of Objects

Bassem Elkarablieh, Yehia Zayour, and Sarfraz Khurshid

The University of Texas at Austin
{elkarabl,zayour,khurshid}@ece.utexas.edu

Abstract. We present Shekoosh, a novel framework for constraint-based generation of structurally complex inputs of large sizes. Given a Java predicate that represents the desired structural integrity constraints, Shekoosh systematically explores the input space of the predicate and generates inputs that satisfy the given constraints. While the problem of generating an input that satisfies all the given constraints is hard, generating a structure at random, which may not satisfy the constraints but has a desired number of objects is straightforward. Indeed, a structure generated at random is highly unlikely to satisfy any of the desired constraints. However, it can be *repaired* to transform it so that it satisfies all the desired constraints.

Experiments show that Shekoosh can efficiently generate structures that are up to 100 times larger than those possible with previous algorithms, including those that are based on a dedicated search and also those that use off-the-shelf enumerating SAT solvers.

Keywords: Constraint-based Generation, Software Testing, Data Structure Repair, Integer Constraint Solving.

1 Introduction

Software systems are steadily growing in complexity and size. At the same time, reliability is becoming a more and more vital concern. Software failures already cost the US economy tens of billions of dollars annually [1]. To meet the ever-increasing demand for reliability, a great deal of progress is required in improving the current state-of-the-art to deliver higher quality software at a lower cost.

Software testing, the most commonly used technique for validating the quality of software, is a labor intensive process, and typically accounts for about half the total cost of software development and maintenance [2]. Automating testing would not only reduce the cost of producing software but also increase the reliability of modern software.

While testing is a conceptually simple process—just create a test suite, i.e., a set of test inputs, run them against the program, and check if each output is correct—the current approaches to testing remain expensive and ineffective. The key issue with the current practice of testing is the need to manually generate test suites.

For programs that take as inputs structurally complex data, which pervade modern software, test generation is particularly hard. Desired inputs must satisfy complex structural integrity constraints that characterize valid structures. Examples of structures

E. Ernst (Ed.): ECOOP 2007, LNCS4609, pp. 248–272, 2007.

include text-book data structures, such as red-black trees that characterize balanced binary search trees [3], which are widely used as library classes, as well as various other structures, such as fault-trees that characterize failures of mission-critical systems [4] and intentional names that characterize properties of services in a dynamic networked environment [5], which are implemented in stand-alone applications.

There are two fundamental approaches for generating structurally complex tests: one, representation-level generation by explicitly allocating objects and setting values of their fields such that the underlying constraints are satisfied; two, abstract-level generation by a sequence of method invocations using the API. The two approaches are complementary and have their advantages and disadvantages. For example, while concrete-level generation requires the user to a priori provide constraints, abstract-level generation requires the user to first correctly implement the methods used in a sequence.

Recent years have seen a significant progress in automating both these approaches. Constraint-based techniques are able to provide efficient test enumeration at the representation level using off-the-shelf SAT solvers [6] as well as using novel search algorithms [7, 8, 9]. Efficient state matching algorithms are able to provide test enumeration at the abstract level by pruning redundant method sequences [11, 12, 10].

Much of the prior work, however, has focused on systematic generation of small structures. The motivation—inspired by traditional model checking—for that is to enable bounded exhaustive testing, where a program is tested on all (in-equivalent) inputs within a small input size. While bounded exhaustive testing does increase a developer's confidence in their software, it is not prudent to altogether ignore testing the program on larger inputs. The existing test generation techniques do not provide an efficient way to generate large structures. Note that the ability to generate large structures even enables a systematic approach to test the performance of the software.

This paper presents a novel algorithm for constraint-based generation of large inputs that represent structurally complex data. We view structures as object graphs whose nodes represent objects and edges represent fields. A key observation behind our algorithm is that while generating an object-graph that satisfies desired structural constraints is hard, generation of a connected graph at random with a desired number of nodes is straightforward. Of course, a graph generated at random is highly unlikely to satisfy any of the desired constraints and would therefore represent an invalid structure. However, we can systematically *repair* such a graph such that it satisfies all the constraints.

Our algorithm deploys an efficient repair routine that we have developed in our previous work on error recovery [13]. Given a structure that violates desired integrity constraints, the repair routine performs *repair actions*, which mutate the structure to transform it into a valid structure. The repair routine performs a systematic state-space exploration of a neighborhood of the given structure and uses *symbolic execution* [14] as well as heuristics to perform efficient and effective repair.

We have evaluated our test generation algorithm on a variety of data structure subjects, including those from the Java Collection Framework. Experimental results using our prototype implementation, Shekoosh, show that our algorithm can generate structures that are 100 times larger than those possible with previous constraint-based generation techniques, such as Korat [7] that implements a dedicated search, or TestEra [6] that uses the Alloy Analyzer [15] and off-the-shelf SAT solvers, such as mChaff [16].

We make the following contributions:

- **Repair for generation.** We introduce the idea of using data structure repair to generate structurally complex tests.
- **Algorithm to generate large inputs.** We present an efficient algorithm for constraint-based generation of large inputs that represent structurally complex data.
- **Implementation.** We present the Shekoosh tool that implements our test generation algorithm.
- **Evaluation.** We evaluate our implementation using a variety of subjects and present experimental results that show two orders of magnitude improvement over the previous state-of-the-art.

2 Example

This section describes an example that illustrates our test generation algorithm. Consider the following class declaration that declares a binary search tree, i.e., an acyclic graph that satisfies the search constraints on the values of its nodes:

```
class BinarySearchTree {
    Node root;
    int size;

    static class Node {
        int elem;
        Node left;
        Node right;
    }
}
```

Each `BinarySearchTree` object has a `root` node and stores the number of nodes in the field `size`. Each `Node` object has an integer value, called `elem`, and has a `left` and a `right` child. The structural constraints of a binary search tree can be written as a predicate that returns true if and only if its input satisfies all the constraints. Following the literature, we term such predicates `repOk` methods and for object-oriented programs, we term structural invariants class invariants [17].

The class invariant of `BinarySearchTree` can be formulated as follows.

```
boolean repOk() {
    if (!isAcyclic()) return false;
    if (!sizeOk()) return false;
    if (!searchConstraintsOk()) return false;
    return true;
}
```

When invoked on a `BinarySearchTree` object o, the predicate `repOk` traverses the object graph rooted at o and checks all the constraints that define a binary search tree. If any constraint is violated the predicate returns false; otherwise, it returns `true`. The helper methods are implemented as standard work-list-based algorithms that keep track of visited nodes [18] (Appendix A gives an implementation of the helper methods).

To generate tests, our prototype Shekoosh takes as inputs the class declarations, the `repOk` predicate and a desired structure size. For this example, for size 100, Shekoosh takes 32 milliseconds (on average) to generate a valid binary search tree; for size 10000 (respectively 100000), Shekoosh generates a structure in less than one (respectively three) seconds (on average). In comparison, TestEra [6], which uses the Alloy Analyzer fails to generate a binary search tree with twenty nodes, due to the analyzer's inability to translate the structural invariant from Alloy to a propositional formula given twenty minutes. Korat [7], which implements a specialized search fails to generate a binary search tree with thirty nodes in twenty minutes. Section 5 presents a detailed comparison for a variety of subject structures.

3 Background: Forward Symbolic Execution

Forward symbolic execution is a technique for executing a program on symbolic values [14]. There are two fundamental aspects of symbolic execution: (1) defining semantics of operations that are originally defined for concrete values and (2) maintaining a path condition for the current program path being executed. A path condition specifies necessary constraints on input variables that must be satisfied to execute the corresponding path. As an example, consider the following program that returns the absolute value of its input:

```
int abs(int i) {
      L1. int result;
      L2. if (i < 0)
      L3. result = −1 ∗ i;
      L4. else result = i;
      L5. return result;
}
```

To symbolically execute this program, we consider its behavior on a primitive integer input, say I. We make no assumptions about the value of I (except what can be deduced from the type declaration). So, when we encounter a conditional statement, we consider both possible outcomes of the condition. To perform operations on symbols, we treat them simply as variables, e.g., the statement on `L3` updates the value of result to be $-1 * I$. Of course, a tool for symbolic execution needs to modify the type of result to note updates involving symbols and to provide support for manipulating expressions, such as $-1 * I$. Symbolic execution of the above program explores the following two paths:

```
path 1:
[I < 0] L1 -> L2 -> L3 -> L5
path 2:
[I >= 0] L1 -> L2 -> L4 -> L5
```

Note that for each path that is explored, there is a corresponding path condition (shown in square brackets). While execution on a concrete input would have followed exactly one of these two paths, symbolic execution explores both.

4 Algorithm

This section describes our test generation algorithm. Our prototype implementation, Shekoosh, has three main modules: *Egor*, a random graph generator, *Juzi++*, an optimized repair framework based on our previous work on error recovery [13], and *Dicos*, a solver for difference constraints [3].

We describe the algorithm for generating a structure that has a unique root; structures that have more than one root are handled similarly [7]. Figure 1 shows the generation framework, which takes three inputs: (1) `clazz` that represents the class of the structure's root, (2) predicate `repOk` that represents the structural integrity constraints, and (3) `size`, a set of pairs, which defines the number of objects for each class in the structure. To illustrate, consider the declaration of the class `BinarySearchTree` from Section 2. To generate tree objects with 100 nodes, we set `size = {<BinarySearchTree, 1>, <Node, 100>}`.

Shekoosh performs the following steps:

- Allocate appropriate objects using the field declarations in `clazz` and generate a random graph using these objects; indeed, this graph may not satisfy any of the desired constraints yet;
- Repair the reference fields of the random graph such that all constraints on these fields are satisfied; Juzi++ returns the constraints on the primitive variables;
- Solve the data constraints; Dicos returns a complete solution;
- Assign each data field its value; the resulting graph represents a concrete object-graph that satisfies all the desired invariants.

The rest of this section describes the details of the algorithm and its main modules.

4.1 Egor: Random Graph Generator

Egor takes an object representing the class declaration of the structure's root class, and the desired size as inputs, and generates a random graph that is allocated on the heap.

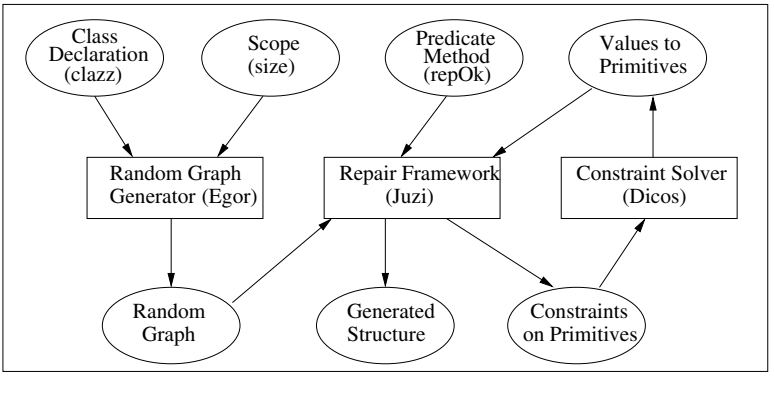

Fig. 1. Shekoosh framework for test input generation

```
Object generateRandomGraph(Class clazz, Set<Pair<Class, int>> size) {

    Random rand = new Random();
    LinkedList liveObjectWorkList = new LinkedList();
    Set deadObjectSet = new HashSet();
    LiveFieldCount liveFieldCount = new LiveFieldCount(clazz);
    CurrentSize currentSize= new CurrentSize(clazz);

    Object root = clazz.newInstance();
    liveObjectWorkList.add(root);
    liveFieldCount.update(root);

    while ( !liveObjectWorkList.isEmpty() ) {
        Object o = liveObjectWorkList.removeFirst();

        for ( Field f : fields(o) ) {
            liveFieldCount.decrement(f);
            if ( currentSize.get(f) == desiredSize(f,size) ) {
                int i = rand.nextInt(2);
                if(i == 0) f.setValue(null);
                if(i == 1) f.setValue(getRandomObject(deadObjectSet));
            }
            else {
                if ( liveFieldCount.get(f) == 0 ) {
                    Object o' = newInstance(f);
                    f.setValue(o');
                    liveObjectWorkList.add(o');
                    liveFieldCount.update(o')
                    currentSize.update(f);
                }
                else {
                    int i = rand.nextInt(3);
                    if(i == 0) f.setValue(null);
                    if(i == 1) f.setValue(getRandomObject(deadObjectSet));
                    if(i == 2) {
                        Object o' = newInstance(f);
                        f.setValue(o');
                        liveObjectWorkList.add(o');
                        liveFieldCount.update(o');
                        currentSize.update(f);
                    }
                }
            }
        }
        deadObjectSet.add(o);
    }
}
```

Fig. 2. The Egor algorithm for generating random graphs

The vertices of the graph are new objects of the given classes. The edges of the graph represent the reference fields. Figure 2 shows the pseudo-code for the Egor random graph generation algorithm.

Intuitively, the algorithm starts with an empty graph. It then allocates new objects as required to generate a graph of the desired size. For each object, the algorithm randomly assigns values to the object's reference fields, ensuring at each step that the graph can further be extended if necessary. The algorithm terminates when the graph has the desired number of objects and all the reference fields of the allocated objects have been initialized.

To explain the algorithm, we first explain the notation we use in Figure 2:

- **`clazz`** is an object representing the container class of the structure (for example the `BinarySearchTree` class).
- **`size`** is a set of pairs representing the desired size of every class in the structure. Egor provides a helper method `desiredSize` that takes a field `f` and `size`, and returns the desired size of the class that is the declared type of `f`.
- **`liveObjectWorkList`** is a list of objects whose reference fields are yet to be assigned a value.
- **`deadObjectSet`** is a set of objects whose reference fields have already been assigned a value. Egor provides a helper method `getRandomObject` that randomly returns an object from the `deadObjectSet`
- **`LiveFieldCount`** is a class that represents for each class the number of object fields, i.e., *live count*, that have not yet been assigned values in the structure. The live count of every class is initially set to zero. `LiveFieldCount` provides three helper methods: `get`, `update`, and `decrement`. The method `get` takes a field object and returns the live count of the field's declared class; `update` takes an object, and for each of its fields, increments the live count of the field's declared class; `decrement` takes a field object, and decrements the live count value of the field's declared class.
- **`CurrentSize`** is a class that represents the number of objects for each class in the structure. For each class, the current size is initially zero. The class `CurrentSize` provides two helper methods: `get` and `update`. The method `get` takes a field and returns the current size of the field's declared class; `update` takes a field and increments the size for the field's declared class.

The Egor generation algorithm first initializes its variables. Next, it creates an instance of the root class (`clazz`), adds it to the `liveObjectWorkList`, and updates the `liveFieldCount`. Next, Egor iterates until the `liveObjectWorkList` is empty. In each iteration, Egor removes the first object from the `liveObjectWorkList` and assigns values to each of the object's reference fields as follows. When assigning a field f of type t, Egor first checks the `currentSize`, and the `desiredSize` for t. If `currentSize` is equal to the `desiredSize`, Egor randomly assigns f to `null`, or to an object from the `deadObjectSet` since new objects of class t can no longer be added to the graph. If the current size is less than the desired size, Egor checks t's `liveFieldCount`. If it is zero, i.e., the graph can only be extended further by assigning a new object to f, Egor allocates a new object o' of type t, assigns o' to f, and updates the `liveFieldCount` and `currentSize` for t. If the live field count is greater than zero, Egor randomly assigns f to `null`, an object from the `deadObjectSet`, or a new object of a compatible type. After assigning all the fields of an object, Egor adds the object to the `deadObjectSet`. Figure 3 illustrates the generation of a `BinarySearchTree` with two nodes.

The generated graph satisfies two key properties: reachability, i.e., all the objects allocated are reachable from the root object, and randomness, i.e., the assignment to each field is made at random (using the Java API). Note that primitive data is left uninitialized. Determining the values for the primitive fields is performed using Dicos after the random structure is repaired by Juzi++. Figure 4 shows an example of a six node `BinarySearchTree` graph generated using Egor.

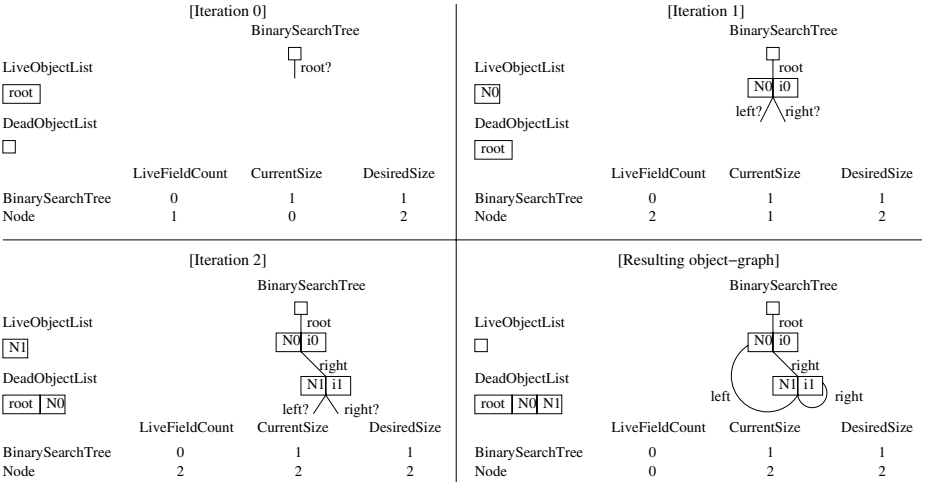

Fig. 3. Egor illustration: generating a random `BinarySearchTree` object with two nodes. The algorithm takes three iterations of the while-loop. The algorithm state at the beginning of each iteration as well as the resulting object-graph are shown. The reference fields are labeled appropriately; a `'?'` indicates the field has not yet been assigned a value by the algorithm; fields that have the value `null` are omitted for clarity. Each node is labeled with its identity (`N0` or `N1`) and a symbolic integer value (`i0` or `i1`).

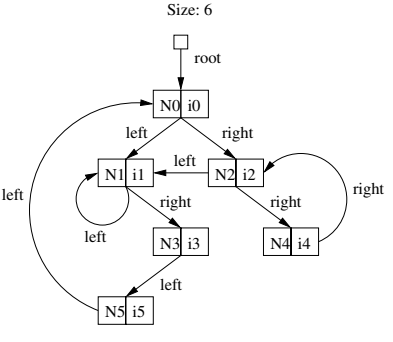

Fig. 4. Random graph with six nodes generated by Egor

4.2 Juzi++: Repair Framework

Juzi++ takes as inputs the random graph generated by Egor and `repOk`, and performs *repair actions* on the graph. Each repair action assigns a new value to an object field. Juzi++ transforms the graph so that it satisfies the desired constraints on the reference fields, and returns a set of constraints on the primitive fields of the structure. Juzi++ builds on our previous work on data structure repair and introduces new heuristics that enable repair to scale to large structures that may have a large number of corruptions. Juzi++ introduces two key heuristics: prioritizing values to use in repair as well as

prioritizing fields to repair. We first describe our basic repair algorithm Juzi [13], and then explain the heuristics that Juzi++ implements.

Juzi. This section describes the basic Juzi [13] repair algorithm. Given a structure `s` that is to be repaired and a predicate `repOk` that represents the structural constraints, the algorithm:

- invokes `s.repOk()`.
- monitors execution of `repOk` to note the order in which fields of objects in `s` are accessed[1].
- if `repOk` returns false
  - backtracks and mutates `s` by toggling the value of the last field that was accessed by `repOk` (while maintaining the values of all other fields), and re-executes `repOk`
- else
  - outputs s (which now has been repaired)

The first invocation of `repOk` (which is on the given corrupt structure) simply follows the Java semantics. When `repOk` returns false, the repair algorithm mutates the given structure, by changing the value of the last accessed field, which is non-deterministically assigned:

- `null`, if the field value was not originally null;
- an object (of a compatible type) that was encountered during the last execution of `repOk` on the corrupt structure, if the field was not originally pointing to this object;
- a new object (of a compatible type), unless the object that the field originally pointed to was different from all objects in the structure encountered during the last execution of `repOk`;

When all the choices for a field assignment have been explored, Juzi resets the value of the last field accessed to its initial value and systematically backtracks to modify the value of the second-last field accessed and so on.

Generation of large structures requires highly efficient repair. Notice that Juzi uses backtracking to perform repair. Thus, Juzi repeatedly invokes `repOk` until the predicate returns *true*. The performance of repair depends on the number of times `repOk` is executed. When repairing a graph generated at random, the number of invocations can be prohibitively expensive. For such a graph, the number of faults are likely to be proportional to its size. Therefore, the number of times `repOk` is executed, which equals the total number of *repair actions* performed, is very high and the basic Juzi approach does not scale, say to repairing large structures, say consisting of 10000 nodes, that have a large number of faults.

Juzi++. We introduce two heuristics for prioritizing repairs to enable an efficient repair framework, which we use in Shekoosh. The heuristics are aimed to optimize repair. However they do not compromise completeness (Section 6).

[1] Execution of repOk is monitored by replacing field accesses with invocations of "get" methods and adding new boolean fields that are set on the first access of the corresponding field. Details are available elsewhere [13, 19].

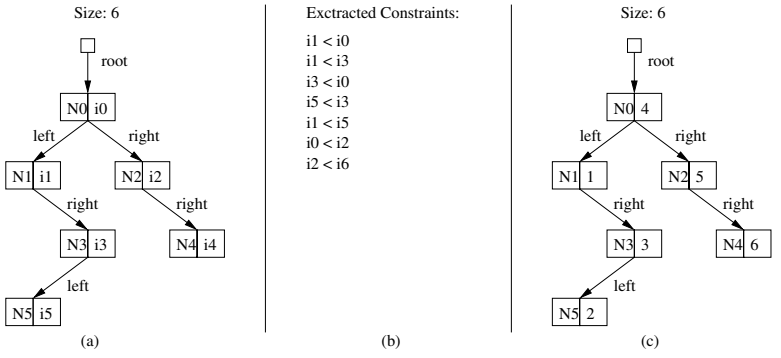

Fig. 5. (a) `BinarySearchTree` object repaired by Juzi++; primitive fields have symbolic values. (b) Difference constraints on primitive fields computed by Juzi++. (c) Solution to the constraints generated by Dicos used to assign values to the primitive fields.

The first heuristics is based on building and solving constraints on references. Juzi++ identifies equality constraint patterns of the form:

```
if ( field != value ) return false;
```

The solution of such constraints is embedded in the negation of the condition. *Juzi++* detects and directly solves such constraints without using nondeterministic assignment. This optimization enables highly efficient solving of a variety of *local* constraints. To illustrate, the parent-child relation of a binary tree takes the following form:

```
Node left = current.left;
if ( left.parent != current ) return false;
```

Juzi++ keeps track of the last equality comparison between a reference field and a value, and selects the value as the first candidate to try. Thus, for the above example, Juzi++ needs to try only one value for repairing the `parent` field of a node.

The second heuristics is based on a lightweight dynamic analysis [20] of the structure. Unlike Juzi, which for each field uses the same order of nondeterministic choices, we use a dynamic ordering. The analysis identifies a set of *core* fields—fields that are used primarily to traverse the structure. When repairing core fields, our algorithm gives higher priority to selecting a new node or `null`, over selecting an already visited node. This optimization guarantees that the reachability of the structure is preserved by repair, and reduces the number of attempts required to find the repaired structure.

These two heuristics dramatically improve the performance of repair (Section 5). The search uses the most *likely* values first, which enables generation of large structures. Note that prioritization of constraints does not compromise the completeness of the algorithm: if a solution to the constraints represented by `repOk` exists for the desired size, our algorithm will generate it. Moreover, the optimizations allow Juzi++ to fix more than one corrupt field using a single execution of `repOk`—an optimization that is essential to scale repair based on imperative constraints.

To illustrate repair, recall the structure shown in Figure 4. Figure 5(a) shows the corresponding repaired structure.

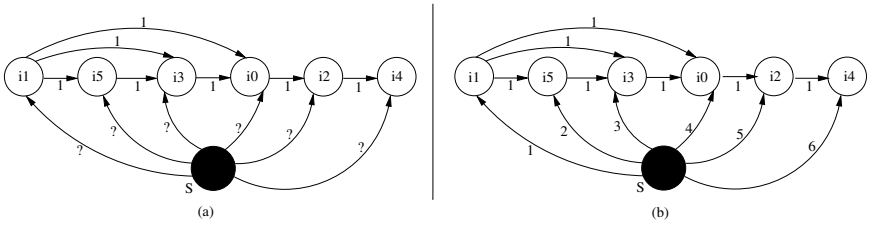

Fig. 6. (a) Data constraint graph based on the constraints from Figure 5 (b); the edges labeled with weight 1 arise from the difference constraints; `s` is the new reference node and edges labeled with `'?'` indicate the new edges. (b) Solution for the difference constrains; each `'?'` has been replaced with the value of the longest distance from `s`.

Juzi++ repairs the structural constraints and returns a set of data constraints, which constrain the primitive fields of the resulting structure. Juzi++ extracts the data constraints using symbolic execution. To illustrate, Figure 5(b) shows the data constraints extracted from the `BinarySearchTree` in Figure 5(a). The constraints returned by Juzi++ are solved by Dicos (Section 4.3), which determines an appropriate value for each primitive field.

4.3 Dicos: Data Constraint Solver

This section describes a technique for generating data values that satisfy the data constraints of the subject structure. A simple approach for generating data values is to assign the values randomly. While generating random values for the primitive fields is straightforward, such an approach is unlikely to give a valid solution in the presence of any constraints on data.

We have developed Dicos, a difference constraint solver for constraints on primitive integers. Our current implementation handles difference constraints that take the form $x < y$ and $x \leq y$ as well as equality constraints. Following a textbook algorithm [3], Dicos builds a constraint graph where the vertices are the primitive fields, and the edges are the difference constraints. Once the graph is built, the problem simplifies to finding the topological order of the nodes in the graph. The primitive values are the longest distances from a new reference node. For a directed acyclic graph with n nodes and e edges, we can compute the primitive values in $O(n + e)$ using a topological sort.

To illustrate, consider the constraints shown in Figure 5(b). Figure 6(a) shows the corresponding constraint graph. The solution of this constraint graph is shown in Figure 6(b). The values for the nodes are the longest distances from the reference node added during the graph construction.

We use the solution returned by Dicos to assign values to the data fields of the structure (Figure 5(c)).

5 Experiments

In this section we evaluate the performance of Shekoosh in generating large data structures. We first describe our experimental methodology. We present different subject

structures with various complex structural and data integrity constraints. We use Shekoosh to generate the subject structures, and we compare our results with related techniques in automatic test input generation. Finally, we discuss the scalability of Shekoosh in terms of the size and the complexity of the generated structures.

5.1 Methodology

We evaluate Shekoosh by applying it to generate ten subject structures. For each subject, we evaluate the time it takes to generate one valid structure for sizes: 10, 100, 1000, 10000, and 100000. We repeat the generation using 50 different randomization seeds and report the average generation time.

Our subjects are divided into three categories: (1) subjects with simple constraints on the structure only, (2) subjects with simple constraints on both the structure as well as the primitive data (3) and subjects with highly complex structural and data constraints. Subjects in category (1) can be generated without using our constraint solver Dicos, i.e., without the need for symbolic execution for primitives, while subjects in category (2) and (3) require its use.

For solving purely structural constraints, two of the previous tools that have been shown to provide efficient solving are TestEra [6], which uses the Alloy Analyzer [21] and off-the-shelf SAT technology, and Korat [7], which implements an imperative constraint solver. For the benchmarks in category (1), we present a comparison of Shekoosh with these two tools. For the benchmarks in categories (2) and (3), TestEra and Korat are unable to compete with Shekoosh because they require explicit enumeration of primitive values and checking of their constraints—the two tools do not use any dedicated solver of constraints on primitives. We have conducted experiments to generate the structures in categories (2) and (3) using Korat and TestEra. We gave Korat and TestEra 20 minutes to generate one structure. Korat and TestEra failed to generate structures with 25 nodes within the given time. The comparison with TestEra and Korat shows that Shekoosh can generate structures of sizes that are 100 times larger.

5.2 Experimental Results

We next describe the data structure subjects and the generation results. All experiments used a 1.7 GHZ Pentium M processor with 512 MB RAM.

Category (1). Subjects in this category only have simple constraints on the structure, and no constraints on primitive data. The structural constraints are limited to reachability, acyclicity, and transposition. We use these structures to evaluate the performance of Shekoosh in comparison with TestEra and Korat. The subjects in this category are as follows:

- **Singly-linked acyclic list.** A list object has a `header` node; each list node has a `next` field. Structural integrity is acyclicity along `next`.
- **Doubly-linked circular list.** A list object has a `header` node; each list node has a `next` and a `previous` field. Structural integrity is circularity along `next` and the transpose relation between `next` and `previous`. This subject is based on the library class `java.util.LinkedList`.

Table 1. Results on solving constraints on the structure. Shekoosh is able to generate structures that are 100 times larger than those feasible with TestEra and Korat. A time of '-' indicates failure to generate in 20 minutes. All tabulated times are in milliseconds.

	Shekoosh			*Korat*	*TestEra*
Singly Linked List	Generation Time(msec)	Repair Time(msec)	Total Time(msec)	Total Time(msec)	Total Time(msec)
10 Nodes	1	2	3	37	3000
100 Nodes	4	4	8	334	-
1000 Nodes	8	14	22	-	-
10000 Nodes	30	33	63	-	-
100000 Nodes	199	483	682	-	-
Doubly Linked List	Generation Time(msec)	Repair Time(msec)	Total Time(msec)	Total Time(msec)	Total Time(msec)
10 Nodes	1	16	17	82	8000
100 Nodes	3	44	43	3204	-
1000 Nodes	14	271	285	-	-
10000 Nodes	50	3718	3768	-	-
100000 Nodes	396	43174	43570	-	-
Binary Tree	Generation Time(msec)	Repair Time(msec)	Total Time(msec)	Total Time(msec)	Total Time(msec)
10 Nodes	1	14	15	21	5000
100 Nodes	2	125	127	512	-
1000 Nodes	14	372	386	-	-
10000 Nodes	85	3672	3777	-	-
100000 Nodes	397	45267	45664	-	-

– **Binary tree.** A binary tree object has a `root` node; each node has a `left` and a `right` child node. Structural integrity is acyclicity along `left` and `right`.

Solving Constraints on Structure. We used Shekoosh to generate the subject structures of this category with sizes ranging from 10 to 100000 nodes. Table 1 shows the results for the subjects in category (1). For test generation, Shekoosh's performance scales essentially linearly. Singly-linked list has the simplest of the constraints and its generation is therefore the fastest. Even though doubly-linked list and binary tree have two fields each, the constraints for doubly-linked list are more complex since they involve two properties (circularity and transpose relation between `next` and `previous`) as opposed to one (acyclicity).

Notice that the generation time is essentially proportional to the number of fields in the structure. The repair time dominates the generation time as expected. The actual time to repair depends on the complexity of the underlying structural constraints.

We gave TestEra and Korat 20 minutes to generate one structure. Overall, Korat performs better than TestEra. However, Korat is unable to generate any subject structure with more than 800 nodes within the given time. Thus, Shekoosh is able to generate structures that are up to 100 times larger than those feasible with Korat and TestEra.

Table 2. Results on solving constraints on the structure as well as on data. All times are in milliseconds.

	Shekoosh			
Sorted Linked List	Structure Generation Time(msec)	Structure Repair Time(msec)	Data Generation Time(msec)	Total Time(msec)
100 Nodes	16	21	5	42
1000 Nodes	22	51	27	100
10000 Nodes	46	210	26	282
100000 Nodes	338	1423	178	1939
Binary Search Tree	Structure Generation Time(msec)	Structure Repair Time(msec)	Data Generation Time(msec)	Total Time(msec)
100 Nodes	10	142	8	160
1000 Nodes	27	422	14	463
10000 Nodes	65	4008	30	4103
100000 Nodes	446	48401	201	49048
Heap Array	Structure Generation Time(msec)	Structure Repair Time(msec)	Data Generation Time(msec)	Total Time(msec)
100 Nodes	10	11	8	29
1000 Nodes	14	37	15	66
10000 Nodes	15	124	29	168
100000 Nodes	55	1084	184	1323

Category (2). Subjects in this category are similar in complexity to those of category (1) yet they have constraints on the order of primitive data. We use these structures to evaluate our constraint solver and measure its efficiency in completing the solution. The subjects in this category are as follows:

- **Sorted linked list.** A sorted linked list is an acyclic linked list whose nodes have integer elements. Integrity constraints include acyclicity as well as ordering of elements: all elements appear in sorted order.
- **Binary search tree.** A binary search tree is a binary tree whose nodes have integer keys. Integrity constraints include acyclicity as well as ordering on keys: for each node, its key is larger than any of the keys in the left sub-tree and smaller than any of the keys in the right-sub tree.
- **Heap arrays.** Heap arrays provide an array-based implementation of the binary heap data structure that is also commonly known as a priority queue. A heap has a `capacity` that is the length of the underlying array and a `size` that is the number of elements currently in the heap. For a heap element at index i, its left child is at index $2 * i + 1$ and the right child is at index $2 * i + 2$. Integrity constraints require that `size <= capacity` and the heap satisfies the max-heap property: an element is larger than both its children.

Solving Constraints on Structure as well as Data. Structures in this category have constraints on the order of the data. For a sorted list, the elements are ordered in a strictly increasing/decreasing order along the `next` field. For a binary search tree the element in the root of a tree is larger than all the elements in the left sub-tree, and less

than all the elements in the right sub-tree. For a heap array, an element at a node is larger than both its children. We used TestEra and Korat to generate these structures, and both failed to generate the first structure with 30 nodes within 20min. TestEra and Korat use a search algorithm to solve the reference constraints as well as data constraints where as in our approach we try to solve the two problems separately if possible (section 6) which allows us to use a dedicated solver for data constraints.

Table 2 tabulates the results for the subjects in category (2). We point out the efficiency of our constraint solver Dicos. The performance of Dicos scales essentially linearly with the size of the generated structures. For test generation, Shekoosh's performance still scales. Note that the structure repair time includes the time to build the constraints on the primitives. (Recall, Juzi++ returns these constraints as its result).

Category (3). Subjects in this category have more complex structural and data constraints than those in categories (1) and (2). These constraints include height balance, path coloring, sentinel reachability (all nodes should have a pointer to a sentinel node) and more. These structures are used to measure the scalability of our approach in generating large data structures, and to discover which phase has the most contribution in the generation time. Again we tried to use TestEra and Korat to generate these structures, and both tools couldn't generate the first structure of 25 nodes within the given 20min threshold. The subjects in this category are as follows:

- **Disjoint set.** The Disjoint set data structure is a linked-based implementation of the fast union-find data structure [3]; this implementation uses both path compression and rank estimation heuristics to improve efficiency. A Disjoint set object has a `header` and a `tail` node as well as a `size` field that represents the size of the set; each set node has a `next` and a `parent` field. Structural integrity constraints are acyclicity and reachability to the sentinel header node (the `parent` field of each node should point to the header node).
- **TreeMap.** TreeMap implements the Map interface using red-black trees. A TreeMap object has a `root` node; and stores the number of entries in the `size` field. A TreeMap node stores a data element in the field `key`, has a `left` and a `right` child, and also has a `parent` pointer. Furthermore, a node has a `color`, which is either RED (false) or BLACK (true). Structural integrity is acyclicity along `left` and `right`, the transpose relation between `left, right` and `parent`, and the natural order on the keys. Furthermore a TreeMap structure should satisfy the following constraints on the colors of its nodes:
  - red entries have black children;
  - the number of black entries on any path from the root to a leaf is the same.
- **AVL tree.** implements the intentional name trees that describe properties of services in the Intentional Naming System (INS) [22], an architecture for service location in dynamic networks. An AVL tree is a balanced binary search tree. The integrity constraints are the same as those of the binary search tree as well as the balance property where the height of the left and the right sub-trees does not differ by more than one.
- **Fibonacci heap.** A Fibonacci-Heap is a dynamic data structure that also implements a heap. A Fibonacci heap object has a `min` field that points to the minimum

Table 3. Results on large structures with very complex structural integrity constraints. All times are in milliseconds.

	Shekoosh			
Disjoint Set	Structure Generation Time(msec)	Structure Repair Time(msec)	Data Generation Time(msec)	Total Time(msec)
10 Nodes	16	4	5	25
100 Nodes	15	63	10	88
1000 Nodes	31	4255	37	4323
5000 Nodes	62	33455	87	33604
10000 Nodes	132	250485	178	250795
AVL Tree	Structure Generation Time(msec)	Structure Repair Time(msec)	Data Generation Time(msec)	Total Time(msec)
10 Nodes	16	16	10	42
100 Nodes	33	94	21	148
1000 Nodes	47	3322	48	3397
5000 Nodes	78	23313	121	23424
7000 Nodes	115	455813	201	456297
Fibonacci Heap	Structure Generation Time(msec)	Structure Repair Time(msec)	Data Generation Time(msec)	Total Time(msec)
10 Nodes	16	32	7	55
100 Nodes	31	64	9	124
1000 Nodes	47	2140	29	2204
4000 Nodes	62	97952	52	98066
7000 Nodes	94	248828	184	249106
Red Black Tree	Structure Generation Time(msec)	Structure Repair Time(msec)	Data Generation Time(msec)	Total Time(msec)
10 Nodes	16	16	10	42
100 Nodes	43	268	22	333
1000 Nodes	62	6546	53	6661
5000 Nodes	267	315671	149	316087

element in the heap, and a `size` field that stores the number of nodes. Each Fibonacci heap node has a `parent` pointer, a `child` pointer, and a `sibling` pointer. A node stores the key element in a `key` field. Detailed description of the structural and data integrity constraints of a fibonacci heap are found in [3].

Generating Structurally Complex Subjects. The results for generating the structures in categories (1) and (2) show that all the components of Shekoosh (Egor, Juzi++, and Dicos) scale linearly. Yet as expected Juzi++ dominated the time required for generation. We further test the scalability of Shekoosh when generating structures with very complex constraints on the structure (complex structures truly determine the efficiency of the repair algorithm and thus the generation approach).

Table 3 shows the results for generating the structures in category (3). Note that as the structure size increases, both Egor and Dicos scale linearly , yet the repair algorithm grows faster. This is due to the complex nature of the structures being generated. The domination of the repair algorithm over the overall generation time is clearly obvious in

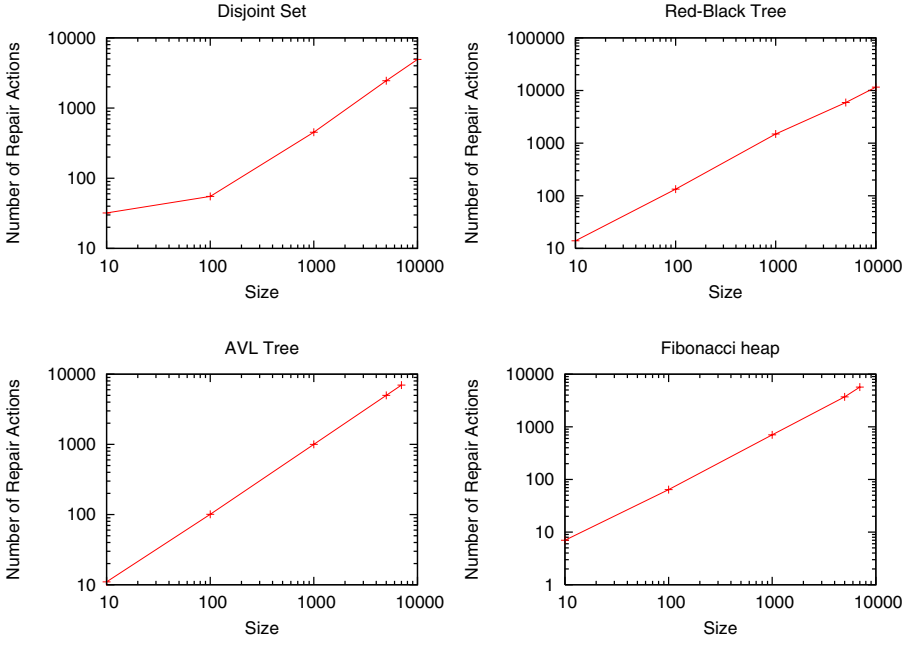

Fig. 7. Variation in the number of repair actions with the size of the structures. For all the studied subject structures, the number of repair actions grows essentially linearly with the size of the structure.

these structures. Even with the complexities, Shekoosh is still able to generate structure with thousands of nodes.

We point out that although the repair algorithm uses a search based approach (similar to Korat), it was still able to repair large structures with very complex properties (the balancing property of the AVL, the path color property of the red-black tree, the sentinel property of the disjoint sets, and the heap property of the fibonacci heap) within the threshold value that we used before terminating the generation. The next section will further investigate the scalability of Shekoosh and the performance of the repair algorithm.

5.3 Scalability of Shekoosh

The experimental results show that the scalability of Shekoosh is highly dependent on the scalability of the repair algorithm. Recall that the performance of Juzi++ is directly proportional to the number of repair actions performed while repairing the structure. We use the number of repair actions as a metric to measure the scalability of Juzi++, and thus Shekoosh. Figure 7 displays the graphs of the variation of the number of repair actions with the size of the structures. Unlike what we expected, although the performance of Juzi++ is not linear (Section 5.2), the number of repair actions grows linearly with the size of the structure. These results are justified as follows:

- First, the number of faults in the structures grows linearly with the size of the structure due to the random creation of the graph generated by Egor. The random graph acts as a partial solution that in the worst case satisfies the reachability and the size constraints of the structure.
- Second, the optimizations that were added to Juzi++ directs the search algorithm to the most likely value to repair a field, and thus the number of repair actions are closer to the number of faults as most of the fault are highly likely to be repaired with only one action.
- Third, and most important, recall that the backtracking algorithm in Juzi++ uses a stateless search space approach. Basically, with every repair action `repOk` is called, the original structure is re-initialized, and the state is built up to the faulty field. Using this approach the structure is reconstructed with every repair action. This causes a quadratic effect on the runtime of Juzi++ even though the number of repair actions grows linearly.

This observation about the number of repair actions shows that for the subject structures the performance of Juzi++ is more dependent on the backtracking algorithm rather than the search algorithm. An alternative approach for backtracking is to use a stateful approach similar to that used in software model checkers like the Java Path Finder (JPF) [10]. Future versions of Shekoosh will implement a stateful approach which can allow Shekoosh to generate even larger structures with complex integrity constraints.

6 Discussion

We next present some characteristics and limitations of Shekoosh and discuss some promising future directions.

Completeness of Generation. Shekoosh repairs a randomly generated structure in two steps: repair of structural constraints (using Juzi++) and repair of data constraints (using Dicos). Doing so enables efficient solving of structural constraints without the overhead of building data constraints on every execution of `repOk`. However, due to possible dependencies among different kinds of constraints, it is possible that a structure that satisfies its structural constraints cannot be repaired further to satisfy the data constraints without modifying the structure. In such cases, Shekoosh systematically backtracks and uses Juzi++ to generate a different structure. We point out that the algorithm is complete for difference constraints: the given structure size makes the underlying problem decidable. Even though the initial structure is generated at random, the repair by Juzi++ and Dicos is systematic. If any valid structure of the desired size exists, such a structure will be returned by Shekoosh; if no such structure exists, Shekoosh notifies the user.

Data Constraint Solving. The current implementation of Shekoosh supports solving two types of data constraints: difference constraints and equality constraints. Solving a problem that comprises additional types of constraints requires extending our solver Dicos or the use of a more sophisticated solver, such as the CVC-lite theorem prover [23].

Test Case Enumeration. We have illustrated Shekoosh for generating one structure of a desired size. Shekoosh can also be used to systematically enumerate a given number

of structures. We expect a typical usage of Shekoosh to be to generate a small set of large test inputs; for inputs of large size, exhaustive generation is infeasible in principle due to the enormous number of valid structures.

Sensitivity to `repOk`. Repair actions performed by our algorithm depend on the formulation of `repOk`. Recall that Juzi++ backtracks on the last field accessed by repOk and modifies that field. Therefore, `repOk`s that return false as soon as they determine a constraint violation without accessing remaining parts of the structure induce faster generation. Naturally written `repOk`s have been shown to enable efficient generation [18].

Constraint-Based Generation Versus Construction Sequences. As we discussed in the introduction, an alternative to constraint-based test generation is the complementary approach of using method sequences to construct structures of a desired size. While this alternative requires an a priori implementation of the methods used in the construction sequence as well as their correct functionality, it presents a viable alternative for generating certain large structures. We have conducted experiments to compare the performance of Shekoosh with construction sequences using our subjects (Section 5). For the subjects in category (1) the performance of both approaches was similar in magnitude. Subjects in category (1) have no data elements, thus both approaches scaled linearly. The comparison for subjects in category (2) is more informative. For sorted linked list, Shekoosh outperforms construction sequences, e.g., for 100000 nodes, construction sequence takes 19800 milliseconds, which is 10 times the time Shekoosh takes. For binary search tree, the times are of a similar magnitude: for 100000 nodes, Shekoosh takes 2372 milliseconds while construction sequences take 1858 milliseconds. For heap array, construction sequences outperform Shekoosh, e.g., for 100000 nodes, construction sequences take 251 milliseconds, which is one-sixth of the time Shekoosh takes. For the structures in category (3) construction sequences outperforms shekoosh due to the complexity of the constraints being solved.

The two approaches are complementary and have their advantages and disadvantages. While construction sequences outperform Shekoosh when generating structurally complex structure, it still requires an a priori correct implementation of the methods used in the sequence. Shekoosh generically generates the structures from the given specification, but its performance is sensitive to the complexity of the structures being generated.

7 Related Work

We discuss how Shekoosh is related to other approaches in test input generation. We first survey some related work on specification-based testing. We then examine different approaches for assertion-based data structure repair. Finally we look at different methodologies for solving constraints on data.

7.1 Specification-Based Linked Structure Generation

Specification-based testing has been present in the testing literature since decades [24]. Many approaches automate test generation from specification languages such as Z

specification [25], JML annotations [26], ASML specifications [27], or UML statecharts [28]. Originally these specifications did not handle complex data structures like the ones we present in this paper, yet some were extended to handle such structures.

The AsmL Test Generator (ASMLT) [27] was recently extended to handle structurally complex data using a search based approach. ASMLT transforms Asml specifications into finite state machines (FSM), and a search algorithm is performed on the FSM to generate the test inputs.

Korat [7] is a search-based test generation tool that exhaustively enumerates all non-isomorphic instances of a complex data structure up to a bound on the size. Unlike AsmlT, Korat takes the constraint written as a *Java* predicate. Korat's approach is highly favorable by programmers since there is no need to learn a new language (which is declarative in most of the cases) to write the specification. Yet the performance of Korat is highly sensitive to the way the Java predicate is written. Korat uses a search based approach that systematically explores the input space to find all the structures to satisfy the given predicate.

TestEra [29] is a test generation tool that uses the Alloy Analyzer [15] which in turn uses off-the-shelf enumerating SAT solvers [16] to generate all the structures that satisfy the integrity constraints. TestEra translates the class declarations of a structure into an Alloy model, and the Java predicate into an Alloy formula which is then fed into the Alloy Analyzer. TestEra is insensitive to the way the specifications are written since the Alloy Analyzer translates the model into a CNF formula before using a SAT solver to find the solution. The performance of TestEra is limited to the efficiency of the Alloy Analyzer when translating the Alloy model, and the performance of the SAT solver to find the solution.

Handling data constraints is a challenge for both search and SAT based approaches. Korat treats data members the same way it treats references. TestEra does not provide an efficient way to handle data elements due to the way integer types are modeled in Alloy [30].

Our generation tool, Shekoosh, differs from the above approaches as it targets generating the first structure that satisfies a given Java predicate rather than enumerating all the test cases. Although Shekoosh uses an approach similar to Korat when repairing the structures, the algorithm scales due to three factors: (1) the partial solution provided by the random graph generator (Although the graph generated by egor is random, yet it still acts as a partial solution which at least satisfies the reachability constraint of the structure), (2) the optimizations added to the repair algorithm which direct the search to the most-likely value to repair a structure, and (3) the dedicated solver (Dicos) for solving the constraints on primitives (unlike Korat).

7.2 Error Recovery

Dynamic error recovery has been part of software systems for a long time [31]. For example, state-full techniques used checkpointing [32] to recover the program state in distributed systems, and stateless techniques implemented dedicated repair routines for special faults [33]. The problem with these approaches is that they require dedicated special routines to be implemented with each system, and thus it is hard to build a robust

generic repair framework using such approaches, since the developer must envision all possible bugs.

Recent work on repair proposes assertion-based techniques to repair a data structure. Assertions have long been used to describe the properties of code. Many tools and techniques make use of assertions to check the program correctness statically and dynamically [2]. The success of assertion checks in hardware verification [34] motivated the use of the same approach for software validation [35]. Most of the recent programming languages have special support for assertions, for example the `assert` keyword in `Java 1.5`. Assertions can be used to describe the structural integrity constraints of a data structure. Such a description can be written in a declarative language like first order logic, or in an imperative language as a predicate method. Although declarative languages provide a more succinct method for describing constraints, there is a large gap between the syntax and grammar of such languages and those of imperative programming languages which are commonly used by software developers and testers.

Recent techniques for repair use the structural constraints to dynamically repair data structures. Demsky and Rinard have recently proposed an constraint-based generic framework for data structure repair [36]. Given the consistency constraints in a new declarative language, their algorithm generates repair routines for each of the integrity constraints.

Juzi [13] is assertion-based framework for data structure repair [13]. However in Juzi, assertions are written in the host language (Similar to Korat). Given the structural constraints written as a `Java` predicate, Juzi systematically searches the neighborhood of the fault and mutates the structure to satisfy the integrity constraints. *Juzi* uses a dedicated search algorithm [7] to find the correct candidate that repairs an erroneous data structure. The performance of *Juzi* depends on the number of faults and the efficiency of the search algorithm in finding the correct candidate.

Shekoosh builds on the Juzi algorithm; it optimizes the search algorithm in order to scale for repairing larger structures with more faults.

7.3 Other Test Generation Techniques

Automatic test case generation is a very old idea, and there are large number of techniques on the topic. We select a couple of techniques and compare them with our approach. A straightforward method for test generation is to build structures using already existing construction sequences. Godefroid et al. proposed DART [37], an automated random testing tool that uses both static and dynamic analysis to generate test cases. Many techniques for test generation use randomized algorithms [38, 39] to generate input tests. Pargas [40] used genetic algorithms to generate a sequence of construction calls that builds up a test input. Claessen et al. [41] use specifications written in Haskell [42] to randomly generate a large number of inputs to test programs written in Haskell. All these techniques can scale to generate large structures with complex structural and data integrity constraints.

Shekoosh differs from the above techniques in that it does not require any implementation of the generated structure. Shekoosh only uses the class declaration and the predicate method describing the constraints.

8 Conclusions

We have presented Shekoosh, a novel framework for generating large data structures. Given a Java predicate that represents the desired structural and data integrity constraints, and the size of the structure to be generated, the Shekoosh test generation algorithm generates a structure that has the given size and satisfies all the constraints. Generation has three phases. The first phase generates a random graph. The second phase repairs the graph to satisfy the structural constraints using an optimized framework based on our previous work on error recovery. The third phase solves the constraints on primitive data and assigns appropriate values to the primitive fields.

Experiments on generating large data structures using subjects with complex structural and data constraints show that Shekoosh can efficiently generate structures with thousands of nodes. In comparison with two existing constraint-based generation frameworks, Shekoosh is able to generate structures that are up to 100 times larger.

We believe Shekoosh presents an exciting new approach to test generation and represents an important step towards a wider application of constraint-based approaches for automated testing and error recovery.

Acknowledgments. We would like to thank Christine Kehyayan, Feras Karablieh, Darko Marinov, and the anonymous referees for their helpful comments on the paper. This work was funded in part by the EDGE scholar program and the NSFITR-SoD award #0438967.

References

1. National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning report 02-3 (May 2002)
2. Beizer, B.: Software Testing Techniques. International Thomson Computer Press (1990)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. The MIT Press, Cambridge, MA (1990)
4. United States Nuclear Regulatory Commission. Fault Tree Handbook, NUREG-0492 (1981)
5. Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., Lilley, J.: The design and implementation of an intentional naming system. In: Proc. 17th ACM Symposium on Operating Systems Principles (SOSP), Kiawah Island (December 1999)
6. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: Proc. 16th Conference on Automated Software Engineering (ASE), San Diego, CA (November 2001)
7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Proc. International Symposium on Software Testing and Analysis (ISSTA) (July 2002)
8. Khurshid, S., Pasareanu, C., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, Springer, Heidelberg (2003)
9. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proc. 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) (2005)
10. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (2004)

11. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: Proc. 19th IEEE International Conference on Automated Software Engineering (September 2004)
12. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
13. Khurshid, S., García, I., Suen, Y.L.: Repairing structurally complex data. In: Proc. 12th SPIN Workshop on Software Model Checking (2005)
14. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7) (1976)
15. Jackson, D.: Micromodels of software: Modelling and analysis with Alloy (2001), `http://sdg.lcs.mit.edu/alloy/book.pdf`
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 39th Design Automation Conference (DAC) (June 2001)
17. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley, Reading (2000)
18. Marinov, D.: Automatic Testing of Software with Structurally Complex Inputs. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology (2004)
19. Khurshid, S., Suen, Y.L.: Generalizing symbolic execution to library classes. In: 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Lisbon, Portugal (September 2005)
20. Malik, M.Z., Pervaiz, A., Khurshid, S.: Generating representation invariants of structurally complex data. In: Proc. 11th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS, vol. 4424, Springer, Heidelberg (March 2007)
21. Jackson, D.: Software Abstractions: Logic, Language and Analysis. The MIT Press, Cambridge, MA (2006)
22. Khurshid, S.: Exploring the design of an intentional naming scheme with an automatic constraint analyzer. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (2000)
23. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, Springer, Heidelberg (2004)
24. Goodenough, J., Gerhart, S.: Toward a theory of test data selection. IEEE Transactions on Software Engineering (June 1975)
25. Horcher, H.-M.: Improving software tests using Z specifications. In: Proc. 9th International Conference of Z Users, The Z Formal Specification Notation (1995)
26. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, Springer, Heidelberg (2002)
27. Stobie, K.: Advanced modeling, model based test generation, and Abstract state machine Language (AsmL). Seattle Area Software Quality Assurance Group (January 2003), `http://www.sasqag.org/pastmeetings/asml.ppt`
28. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: France, R.B., Rumpe, B. (eds.) UML '99 - The Unified Modeling Language. Beyond the Standard. LNCS, vol. 1723, Springer, Heidelberg (1999)
29. Khurshid, S., Marinov, D.: Checking Java implementation of a naming architecture using TestEra. In: Stoller, S.D., Visser, W. (eds.) Electronic Notes in Theoretical Computer Science. ENTCS, vol. 55. Elsevier Science Publishers, Amsterdam (2001)
30. Jackson, D.: Alloy: A lightweight object modeling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(2) (2002)

31. Smirnov, A., Chiueh, T.-c.: DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In: The 12th Annual Network and Distributed System Security Symposium, February 2005, San Diego, CA (2005)
32. Kim, J.L., Park, T.: An efficient protocol for checkpointing recovery in distributed systems. IEEE Transactions on Parallel and Distributed Systems (August 1993)
33. Karablieh, F., Bazzi, R.A., Hicks, M.: Compiler-assisted heterogeneous checkpointing. In: SRDS, p. 56 (2001)
34. Wile, B., Goss, J., Roesner, W.: Comprehensive Functional Verification: The Complete Industry Cycle. Morgan Kaufmann, San Francisco (2005)
35. Synopsis. Assertion-based verification (March 2003), `http://www.synopsys.com/products/simulation/assertion_based_wp.pdf`
36. Demsky, B., Rinard, M.: Automatic detection and repair of errors in data structures. In: Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2003)
37. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (2005)
38. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7) (1997)
39. Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Software Engineering Journal (1996)
40. Pargas, R.P., Harrold, M.J., Peck, R.: Test-data generation using genetic algorithms. Journal of Software Testing, Verification, and Reliability 9(4) (1999)
41. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM SIGPLAN Notices 35(9), 268–279 (2000)
42. Jones, S.P., Hughes, J.: Report on the Programming Language Haskell 98. A Non-strict Purely Functional Language (February 1999)

A Structural Invariants for BinarySearchTree

The helper methods used in class invariant of `BinarySearchTree` can be formulated as follows.

```
// checks the acyclicity property of the tree
boolean isAcyclic() {
    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);
```

```
        }
    }
    return true;
}
// checks if the reachability constraint is satisfied
boolean sizeOk() {
     return size == numNodes(root);
}
// returns the number of nodes reachable from the root node
int numNodes(Node n) {
    if (n == null) return 0;
    return (1) + (numNodes(n.left))+ (numNodes(n.right));
}
// checks if the tree elements satisfy the order constraints
boolean searchConstraintsOk() {
     return  isOrdered(root, MINUSINFINITY, PLUSINFINITY);
}

boolean isOrdered(Node n, Object min, Object max) {
    if ((min != null && compare(n.element,min) <=0 ) ||
    (max != null && compare(n.element,max) >=0)
        return false;
    if (n.left != null)
        if (!isOrdered(n.left, min, n.element)) return false;
    if (n.right != null)
        if (!isOrdered(n.right, n.element, max)) return false;
    return true;
}
```