Repairing Structurally Complex Data

Sarfraz Khurshid and Iván García and Yuk Lai Suen

Department of Electrical and Computer Engineering
The University of Texas at Austin
1 University Station C5000
Austin, TX 78712
{khurshid,igarcia,suen}@ece.utexas.edu

Abstract. We present a novel algorithm for repairing structurally complex data. Given an assertion that represents desired structural integrity constraints and a structure that violates them, the algorithm performs repair actions that mutate the given structure to generate a new structure that satisfies the constraints. Assertions are written as imperative predicates, which can express rich structural properties. Since these properties can be arbitrarily complex, our algorithm is sound but not complete, and it may not terminate in certain cases. Experimental results with our prototype implementation, Juzi, show that it is feasible to efficiently repair a variety of complex data structures that are routinely used in library code. Juzi can often repair structures comprising of over a hundred objects (even when majority of the objects have some corrupted field) in less than one second. Our algorithm is based on systematic backtracking but does not require storing states and can easily be implemented in a variety of software model checkers, such as the Java PathFinder, SPIN, and VeriSoft.

1 Introduction

Assertions have long been used to state crucial properties of code. A variety of tools and techniques make use of assertions to check program correctness statically at compile-time or dynamically at run-time [3–5,7,10,12,14,16,20,23,29]. If an assertion violation is detected at run-time, the program is deemed to have reached an inconsistent state. The usual process then is to terminate the execution, debug the program (if necessary and possible), and re-execute it.

In some cases, however, termination and re-execution is not feasible. And at times, it is simply impossible to re-execute a program on a desired input since the input may now represent (persistent) data that has been corrupted. Even correct programs can have corrupt data due to errors in transmission, hardware, etc. In such cases, it may be desirable to have a routine that can repair the corrupted data and bring the data in a state that is consistent with the integrity constraints expressed in the assertion, which would enable the program (to continue) to execute. A goal of such a routine is not to bring the data in a state that a correct execution/environment would have resulted in, but to bring it in a state that is acceptable to the user for continuing program execution [25].

We present a novel algorithm [11,27] for repairing *structurally complex data*, which pervade modern software, in particular object-oriented programs. A defining characteristic of such data is their *structural integrity constraints*, e.g., in a binary tree, there

To appear in 12th International SPIN Workshop on Model Checking of Software (SPIN 2005).

are no cycles. Examples of complex structures include textbook data structures, such as circular linked lists and red-black trees, which are routinely used in library code to implement a variety of abstract data types. Complex structures arise in various other contexts, e.g., intentional naming systems [1] and fault-tree analyzers [28].

The integrity constraints of a structure can be written as a formula that evaluates to true if and only if the input satisfies the desired constraints. Such formulas can be written declaratively [15,17,23], e.g., using fist-order logic, or imperatively [4,22], e.g., using a Java or C++ predicate (i.e., a method that returns a boolean). Declarative notations often provide a more natural and succinct way of expressing constraints. However, these notations are usually syntactically and semantically different from common programming languages, which can impede their wide-spread adoption among practitioners.

Our repair algorithm uses imperative descriptions of constraints. In object-oriented programs such constraints are often already present as class invariants (which are usually called `repOk` methods) [21]. Given a predicate that represents desired structural integrity constraints and a structure that violates them, the algorithm performs repair actions that mutate the given structure to generate a new structure that satisfies the constraints. Each repair action assigns some value to a field of an object in the structure.

The repair actions are governed by the (1) exploration of the set of field assignments to reference variables and (2) evaluation of constraints on values of primitive data fields. Due to the enormous number of combinations of field assignments, it is not possible to simply enumerate all possible assignments (even for small structures) and check whether any assignment represents a repaired structure. For efficient repair, our algorithm employs (1) pruning techniques that are based on our previous work on the Korat framework for specification-based testing of Java programs [4], and (2) decision procedures for primitive data, similar to our previous work on test input generation using symbolic execution [18].

The algorithm executes the predicate on the corrupted structure and monitors the execution to record the order in which fields are accessed before the execution returns false. The algorithm then backtracks on the last field that was accessed and either assigns that field a different reference or assigns it a symbolic primitive value (which is different from the original value), and re-executes the predicate using (forward) symbolic execution [19] where needed. To determine the feasibility of path conditions, our prototype implementation, Juzi, uses CVC Lite [2].

At its core, our algorithm performs a systematic search using backtracking based on field accesses and on results of decision procedure invocations. Our algorithm does not require storing states. These characteristics make it very easy to implement our algorithm to work in conjunction with a variety of software model checkers, such as the Java PathFinder [29], SPIN [14], and VeriSoft [12].

Imperative predicates enable formulation of rich structural properties. Since these properties can be arbitrarily complex, our algorithm is sound but not complete: the repaired structures that the algorithm returns satisfy the constraints, but the algorithm may not terminate in certain cases. Experimental results with our prototype implementation show that it is feasible to efficiently repair a variety of complex data structures, which are used routinely in library code. Juzi can repair structures with a hundred nodes—half of which have some field that needs repair—in less than one second.

1.1 Background

Fault-tolerance and error recovery have been a part of software systems for a long time. File system utilities, such as `fsck`, routinely check and correct the underlying file structure. Some commercially developed systems, such as IBM MVS operating system [24] and the Lucent 5ESS telephone switch [13], have provided routines for monitoring and maintaining data structure properties. These routines, however, typically focus on repairing particular structures by performing specific repair actions that work only in the contexts for which they are designed.

Checkpointing and roll-back are standard mechanisms in databases to recover data to the last known good state. DIRA [26] adapts these mechanisms to detect buffer overflow attacks and repair the structures damaged by the attack.

Demsky and Rinard have recently proposed a generic model-based framework for data structure repair [9]. Given consistency constraints in a declarative language, their repair algorithm translates these constraints into a repair routine, which corrects the given corrupt structure. A distinguishing feature of our work from previous work on repair is that we provide a generic repair algorithm that does not require any input from the user beyond a description of the desired constraints and does not require learning a language different from the underlying programming language.

1.2 Contributions

This paper makes the following contributions:

- **Imperative constraints in data structure repair**. Our use of imperative constraints in the context of generic data structure repair is novel. It enables users to write constraints in a familiar notation and eliminates the need for requiring mappings between abstract models of data and concrete values; such mappings are often required when the constraint language differs from the implementation language [9].
- **Forward symbolic execution in data structure repair**. Forward symbolic execution has traditionally been used to check correctness of programs (via static or dynamic analyses) and to debug the programs. We have developed an unconventional application of symbolic execution: we use it to repair the *data* on which the programs operate on.
- **Data structure repair algorithm**. We build on algorithms from our previous work on specification-based test generation to develop a novel technique for performing generic data structure repair.
- **Repair as an application of a model checker**. We have designed our algorithm to work with off-the-shelf model checkers. To our knowledge this is the first instance that shows how a standard model-checking tool can efficiently perform data structure repair.
- **Repair studies from library code**. We perform repairs on a suite of complex structures used routinely in library code and evaluate the feasibility of structure repair. Experiments show that moderately large structures, e.g., red-black trees with a few hundred nodes, can often be repaired within a few seconds.

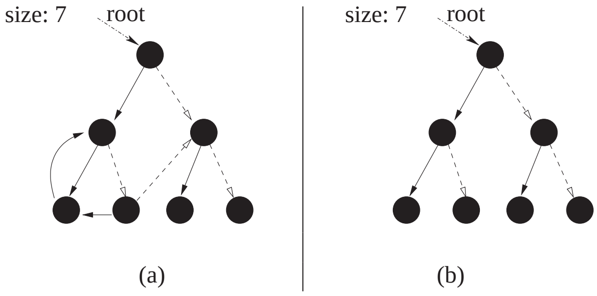

Fig. 1. Repairing a binary tree. (Solid arrows represent `left` fields; dashed arrows represent `right` fields; and `root` field is labeled appropriately.) (a) A corrupted binary tree structure: values of left and right fields of some nodes introduce directed cycles in the structure. (b) Tree resulting after repair has been performed.

2 Examples

We present two examples of repairing linked structures to illustrate the use of our algorithm and prototype implementation. The first example illustrates an acyclic data structure that has been corrupted. The second example illustrates a structure that has cycles and also shows how repair can sometimes even correct program behavior on-the-fly.

2.1 Binary tree

Consider the following declaration of a binary tree:

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

    static class Node {
        Node left;
        Node right;
    }

    boolean repOk() { ... }
}
```

Each tree has a `root` node and caches the number of nodes in the `size` field. Each node has a `left` child and a `right` child. The method `repOk` checks the structural integrity constraints for `BinaryTree`. It can be implemented as a simple graph traversal algorithm that checks for acyclicity by keeping track of the set of visited nodes and checking that it never encounters the same node twice. The method also checks for consistency of the `size` field. (Appendix A gives an implementation of `BinaryTree.repOk`.)

As an illustration of the repair algorithm, consider the corrupted structure shown in Figure 1(a). Incorrect values of fields `left` and `right` in some of the nodes result in the structure having directed cycles, which violates acyclicity. Given a description of this structure and the `repOk` method, our repair algorithm produces the repaired structure shown in Figure 1(b). Note that the field assignments now satisfy the desired constraints. To repair the corrupted structure shown in this example, Juzi takes a tenth of a second.

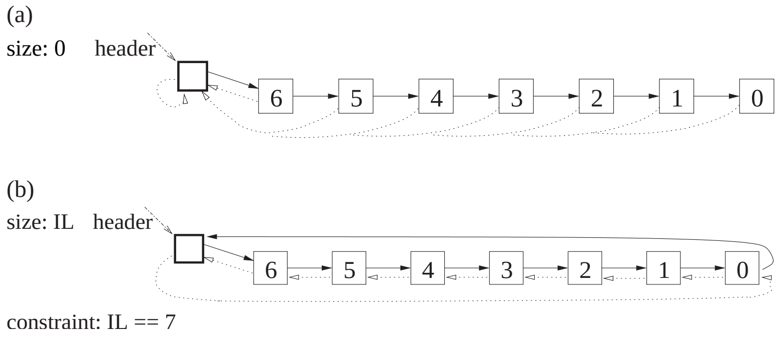

Fig. 2. Repairing a doubly-linked list. (Solid arrows represent `next` fields; dotted arrows represent `previous` fields; and `header` field that points to the sentinel node is appropriately labeled.) (a) List generated as result of erroneous `addFirst`: all `previous` pointers incorrectly point to the `header` entry, and `size` is set to 0. (b) List resulting after repair has been performed: all reference fields have correct values and `size` field is correctly constrained to 7.

2.2 Doubly-linked list

We illustrate how repair can potentially even correct program behavior on-the-fly. The class `LinkedList` declares doubly-linked circular lists similar to those implemented in `java.util.LinkedList`:

```
class LinkedList {
    Entry header; // sentinel header entry
    int size;     // number of non-sentinel entries

    static class Entry {
        Object element;
        Entry next;
        Entry previous;
    }
}
```

The inner class `Entry` models the entries in a list. Each list has a `header` entry, which is treated as a *sentinel*. An empty list consists of just the header entry, whose next and previous fields point to itself. The `size` field stores the number of non-sentinel entries in the list.

Consider a method `LinkedList.addFirst` that given an object `o`, adds a new entry with element `o` at the head of `this` list (i.e., it makes the new entry the first non-sentinel entry in the list while preserving the original entries of the list). The following code gives an erroneous implementation of the `addFirst` method:

```
    void addFirst(Object o) {
        Entry t = header.next;
        Entry e = new Entry();
        e.element = o;
        header.next = e;
        e.previous = header;
        e.next = t;
        t.previous = header;
    }
```

The above code contains two bugs:

- it does not maintain the correspondence between `next` and `previous` fields of an entry as it erroneously sets `t.previous` to `header` instead of setting it to `e`
- it does not update the value of the `size` field.

Figure 2(a) illustrates the list that is generated by inserting integer objects with values $[0, \ldots, 6]$ in that order into an empty list using the erroneous `addFirst`. Notice the incorrect values for `previous` pointers (dotted arrows) and `size` field. All `previous` pointers incorrectly point to the `header` entry, and `size` is 0 even though there are 7 non-sentinel entries in the list.

Given this corrupted list and the `LinkedList.repOk` method, which we have not given here due to brevity, Juzi generates the structure illustrated in Figure 2(b). Notice how the `previous` pointers have been set to correct values and how the size field is constrained to have the correct value 7. For this example, Juzi took a tenth of a second to complete the repair.

3 Background: Symbolic execution

Forward symbolic execution is a technique for executing a program on symbolic values [19]. 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 line 3 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 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;
- if `repOk` returns false
  - backtracks and mutates `s` by toggling the value of the last field[1] that was accessed by `repOk` (while maintaining the values of all other fields), and re-executes `repOk`
- else
  - if (pathCondition is feasible)
    - outputs `s` (which now has been repaired)

The first execution of the algorithm is on the corrupted structure. Notice that all fields of this structure have concrete values. Therefore, the first invocation of `repOk` simply follows Java semantics. But when `repOk` returns false, the algorithm mutates the given structure, and may introduce fields that have symbolic values for primitive data; value updates to these field then follow standard forward symbolic execution [19].

To modify a field value when backtracking, the algorithm considers two primary cases[2]:

- primitive field access: the field is assigned a symbolic value `I` and the current path-condition is updated to reflect that `I != v`, where `v` is the original value of this field in the corrupt structure;
- reference field access: the field is nondeterministically assigned
  - `null`, if the original field value was not `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`.

It is tempting to think that a reference field of an object in a structure can potentially point to any other object that has a compatible type in that structure and to explore all such assignments. However, our algorithm does not explore them all. It turns out that it suffices to select the possible assignments from the part of the structure that has so far been accessed, and only one object that is distinct from those previously encountered during the last execution of `repOk`. In fact, trying more than one such object amounts to making equivalent assignments since they result in isomorphic structures [4]. Indeed,

[1] If all values for the last field accessed have already been explored, reset the value of that field to its initial value and backtrack further to modify the value of the second-last field accessed and so on.

[2] Our current prototype does not handle arrays.

there is little reason to explore more than one structure from a set of isomorphic structures since they are either all valid or all invalid[3].

Our repair algorithm builds on our previous work on test input generation using Korat [4] and generalized symbolic execution [18], and adapts those algorithms to perform efficient data structure repair.

5 Implementation

Our prototype, Juzi, is written in Java and works for repairing structures that a Java program manipulates. There are three key inputs to Juzi: (1) the name of the class to which the structure belongs; (2) the name of the method that represents the desired structural constraints; and (3) the name of a method which represents the corrupted structure. To systematically modify field values and to perform symbolic execution, Juzi performs instrumentation of Java bytecode and implements a simple backtracking algorithm. Juzi uses CVC Lite [2] to determine feasibility of path conditions that it builds during symbolic execution.

5.1 Bytecode instrumentation

There are two basic functions that Juzi performs using bytecode instrumentation: (1) systematic assignment of values to fields; (2) symbolic execution.

Recall that our repair algorithm uses a systematic assignment of values to fields, where some values may be symbolic. How these assignments are made depends crucially on the order in which fields are accessed. To record this order, Juzi transforms the original code and replaces field accesses by invocations of methods that Juzi adds to the given code[4]. The method invocations allow Juzi to record field accesses. For example, for the doubly-linked list example (Section 2.2), the Java bytecode statement

```
6:   getfield        #18; //Field header:Ljuzi/examples/LinkedList$Entry;
```

which accesses the `header` field, transforms to

```
6:   invokevirtual   #252; //Method _get_header:()Ljuzi/examples/LinkedList$Entry;
```

which invokes the method `_get_header`—a method that Juzi adds to allow monitoring `repOk`'s executions.

To enable symbolic execution, Juzi replaces primitive types by library types that it provides to represent expressions on symbolic (and concrete) values. Juzi performs a conservative reachability analysis to see what types need to be transformed and generates appropriate bytecodes. Juzi also transforms operations on primitive values into appropriate method invocations. For example, when a primitive integer constant forms a sub-expression in an expression on symbols and concrete values, Juzi wraps the integer constant in an object. As an illustration, consider the following sequence of Java bytecodes:

[3] We assume that `repOk` uses actual object identities only in comparison operations [4, 22].

[4] We used a similar approach in previous work that used source-code instrumentation to perform test generation [4, 18, 22].

```
   ...
   16:  iconst_3
   17:  iadd
   ...
```

is transformed to

```
   ...
   16:  new     #280; //class IntConstant
   19:  dup
   20:  iconst_3
   21:  invokespecial   #283; //Method juzi/expr/literal/IntConstant."<init>":(I)V
   24:  invokevirtual   #296; //Method juzi/expr/Expression.iadd:
                              //        (Ljuzi/expr/Expression;)Ljuzi/expr/Expression;
   ...
```

which shows the wrapping of the integer constant 3 into an object of the library class `juzi.expr.literal.IntConstant`, followed by an invocation of the library method `iadd`, which is one of the methods Juzi implements to build expressions containing symbols and concrete values.

To allow symbolic execution to explore different program paths, Juzi uses a non-deterministic boolean choice whenever there's a branch in bytecode that cannot be deterministically evaluated on-the-fly.

Juzi uses the Java Programming Assistant (Javaassist) [6] to perform bytecode instrumentation.

5.2 Backtracking

Juzi implements a simple backtracking algorithm to provide non-deterministic choice. The class `Explorer` provides method `choose` which takes an integer input and represents a non-deterministic choice, for example the assignment

```
x = Explorer.choose(3);
```

non-deterministically assigns the values $0, 1, 2, 3$ to `x`. Such non-deterministic choice operators are an essential feature of software model checkers [12, 14, 29].

The Juzi backtracking algorithm performs stateless depth-first search (i.e., stores no states but remembers the values it uses when making non-deterministic assignments with `choose`). Non-deterministic code is thus re-executed from the beginning, and during each execution one of the non-deterministic assignments is made differently from that in the previous execution.

5.3 Satisfiability of path conditions

Juzi checks satisfiability of path conditions using the CVC Lite [2] automated theorem prover. CVC Lite provides a C++ API for checking validity of formulas over several interpreted theories including linear arithmetic on integers and reals, arrays and uninterpreted functions. Since CVC Lite is implemented in C++, it can be expensive to make calls to it from a Java program. Juzi, therefore, implements some on-the-fly simplifications of path conditions as it builds them. The simplifications not only allow Juzi to generate smaller path conditions but also, in some cases, let it decide satisfiability without having to call CVC Lite routines. Juzi's simplifications include transforming constraints in a path condition to a canonical form, performing subsumption checking for simple cases, and propagating constants.

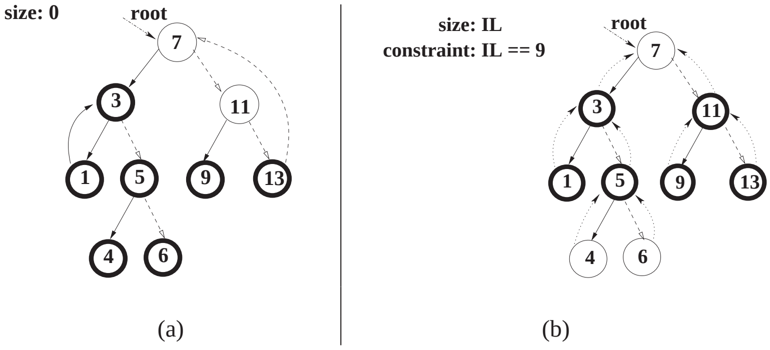

Fig. 3. Repairing a red-black tree. (Solid arrows represent `left` fields; dashed arrows represent `right` fields; dotted arrows represent `parent` pointers; `root` field is labeled; `key` is written inside the entry; entries drawn with thick circles are black and the others are red.) (a) A corrupted red-black tree structure: has cycles; has variable number of black entries along different paths from root; has a red entry with a red child; has incorrect size. (b) Tree resulting after repair has been performed: field values have been modified to satisfy the structural constraints; the `size` field is correctly constrained to be exactly 9.

6 Case study: red-black trees

To illustrate the variety of constraints that our repair algorithm can handle, we next present a case study on repairing red-black trees [8], which implement balanced binary search trees. Red-black trees are structurally among the most complex of the commonly used data structures and therefore present a challenging study for repair. The experiments show that Juzi can efficiently repair red-black trees of moderately large sizes, e.g., repairing a tree with over a hundred nodes—almost all of which had at least one field with a corrupted value[5]—in less than a second. It is worth pointing out that prior work [9] on repair has not addressed repair of structures as complex as red-black trees.

All experiments reported in this paper were performed on a 1.6 GHz Pentium M processor with 1 GB of RAM.

The following code declares a red-black tree in a fashion similar to the implementation in `java.util.TreeMap`:

```
class TreeMap {
    Entry root;
    int size;

    static class Entry {
        int key;
        Entry left;
        Entry right;;
        Entry parent;
        boolean color;
    }

    boolean repOk() { ... }
}
```

[5] Less corrupt structures are repaired even more efficiently.

A tree has a `root` entry and stores the number of entries in the `size` field. An entry stores a data element in the field `key`, has a `left` and a `right` child, and also has a `parent` pointer. Furthermore, an entry has a color, which is either `RED` (false) or `BLACK` (true).

Red-black trees are binary search trees. In addition to acyclicity and correct search order on keys, there are two fundamental constraints that define red-black trees:

- red entries have black children;
- the number of black entries on any path from the root to a leaf is the same.

Of course, the value of the `size` field needs to correctly reflect the number of entries.

Consider the corrupted structure shown in Figure 3 (a). Not only is it not acyclic, but it also violates both the constraints on the coloring of entries, has an incorrect value for the `size` field, and has all the `parent` pointers set incorrectly to `null`.

Given this structure and `repOk` for `TreeMap` (which we do not present here due to brevity), Juzi produces the structure shown in Figure 3 (b). Notice that all fields now have correct values; the value of `size` field is correctly constrained to equal 9. Juzi completed the repair in a tenth of a second.

7 Discussion

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

7.1 Structure repair versus structure generation

We view structure repair as being closely related to structure generation and therefore closely related to test input generation from input constraints [17, 22]. Generation addresses the problem of generating structures that satisfy given structural constraints, while repair addresses the problem of generating a structure that not only satisfies the constraints but is also heuristically close to a given corrupt structure. Interestingly, on the one hand, generation can aid repair since any structure that satisfies the constraints is indeed a candidate for being the repaired structure, while on the other hand, repair can aid generation since an arbitrarily selected corrupt structure may be repaired to generate a desired structure.

7.2 On-demand symbolic execution

Our use of symbolic execution is non-conventional not only in the sense of our application to data structure repair, but also in the sense of how we perform it. Symbolic execution is usually performed either by treating all program inputs as symbolic [19] or by a priori determining which inputs to treat as symbolic and which to treat as concrete (e.g., symbolic primitives and concrete references [18]).

Our repair algorithm takes a different approach. It starts by executing `repOk` (class invariant) on a structure, all of whose fields have concrete values. During subsequent executions of `repOk`, the algorithm makes values of certain fields symbolic. However, the values of these fields do not have to stay symbolic during all subsequent executions—a field may regain a concrete value (since backtracking re-initializes field values). The algorithm, thus, performs symbolic execution on an as-needed basis and whether it treats a field as symbolic or concrete depends on the particular execution being considered. This hybrid approach enables the algorithm to explore structures in a "neighborhood" of the original structure and generate a new structure that is heuristically similar to the original one.

7.3 Sensitivity of repair to `repOk`

Repair actions performed by our algorithm intrinsically depend on how `repOk` is formulated. Recall that the algorithm backtracks on the *last* field accessed by `repOk` and modifies this field. This means that for the same corrupted structure, two different `repOk` implementations that access fields in different orders may cause our repair algorithm to produce different structures. Even though this sensitivity to the way constraints are written may be considered an inherent limitation of the algorithm, in fact, it allows the user to control how the structure may be repaired. By ordering constraints appropriately the user can ensure that the algorithm will not perturb the values of certain fields (that the user's deems unlikely to be corrupted) unless absolutely necessary.

7.4 Repairing primitive data values in a structure

The question of how to repair primitive data values in a structure is rather important for any repair algorithm. For example, consider repairing a binary search tree whose elements are not in the correct search order. One way to repair this structure is to replace the elements with new elements that appear in the correct search order. However, this is unlikely to be a good repair choice, e.g., consider the case when the tree is implementing a set—it is the elements that define the set and are therefore of crucial significance.

Our approach is to allow the user to specify ranges of data values for primitive fields and to use these ranges to constrain the repaired values of these fields. Juzi reads these ranges from a configuration file. The user can choose not to provide any range, in which case Juzi (by default) tries to preserve as many of the original values as possible. We plan to allow the user to state specific relations between (values of) a corrupted structure and a repaired structure akin to specifying post-conditions that relate pre-state with post-state. A more sophisticated approach could define metrics of similarity between corrupted and repaired structures; these metrics could then be used as a basis of new algorithms that produce repaired structures that are maximally similar to the given corrupted structures.

7.5 Converting symbolic values to concrete values

Our repair algorithm uses a decision procedure for evaluating feasibility of path conditions. A caveat of using an automated theorem prover for this purpose is that such tools typically report only the feasibility of constraints and not actual valuations of the variables in feasible constraints. This implies that when we repair the value of a primitive data field, we need to perform an additional step of selecting a concrete value. In the benchmarks we have shown, selection of such values has just been a trivial step. However, in other cases when constraints are more complex, it is non-trivial to select these values.

7.6 Multi-threaded programs

Corrupted data in multi-threaded programs can be repaired by suspending processes that manipulate this data, repairing the data using the repair routine, and resuming the processes. This requires, however, control over thread scheduling, which cannot easily be achieved for arbitrary programs running under a standard virtual machine. Programs where it is crucial to maintain essential structural integrity constraints can, however, be run under environments that provide such suspend/resume mechanisms.

7.7 Structure repair and program debugging

Repair of a structure can lend useful information about error localization in a faulty program to aid its debugging. For example, if repair actions only set values of the fields `previous` and `size` (as in Section 2.2), the user can start debugging by first looking at those parts of the code that modify these fields.

7.8 Repairing large structures

Our repair examples so far have involved small structures. Repair can, however, be performed feasibly for modestly large structures. For example, for doubly-linked list and binary tree (Section 2), Juzi can repair structures with over 1000 nodes—of which over a 100 have some field that needs repair—in under a minute. Structures with over 100 nodes—of which over 50 have some field that needs repair—are repaired in about a second, even in the case of red-black trees (Section 6). These results are encouraging as they point out that repair routines can efficiently be included in code where it is essential to enforce structural integrity constraints at key control points. Whether a generic repair approach can scale to repairing structures with millions of nodes—of which thousands have some field that needs repair—well, that remains to be seen.

8 Conclusions

We have presented a novel algorithm for repairing structurally complex data. Given an assertion that represents desired structural integrity constraints and a structure that violates them, the algorithm performs repair actions that mutate the given structure to generate a new structure that satisfies the constraints. Assertions are written as imperative predicates that can express rich structural properties. Since these properties can be arbitrarily complex, our algorithm is sound but not complete.

Experimental results with our prototype, Juzi, show that it is feasible to efficiently repair a variety of complex data structures that are used routinely in library code. Juzi can often repair structures with over a hundred objects (where majority of the objects have at least one field that has been corrupted) in less than one second.

Our algorithm is based on systematic backtracking but does not require storing states and can easily be implemented in a variety of software model checkers, such as the Java PathFinder, SPIN, and VeriSoft.

Acknowledgments

We would like to thank the anonymous reviewers and Darko Marinov for detailed comments on a previous draft. This work was funded in part by NSF ITR-SoD award #0438967 and in part by the GEM fellowship.

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, December 1999.
2. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference On Computer Aided Verification*, Boston, MA, July 2004.
3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36thConference on Design Automation (DAC)*, New Orleans, LA, 1999.
4. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.
5. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, June 2002.
6. Shigeru Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
7. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.
8. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
9. Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proc. ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 78–95, 2003.
10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.
11. Iván García. Enabling symbolic execution of Java programs using bytecode instrumentation. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2005.
12. Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, January 1997.
13. G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, 1985.
14. Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
15. Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.
16. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, August 2000.
17. Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, December 2003.
18. Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

19. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
20. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.
21. Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.
22. Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.
23. Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.
24. Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, 13(10):1135–1139, 1987.
25. Martin Rinard. Resilient computing. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, 2003. (Research Abstract).
26. Alexey Smirnov and Tzi cker Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *The 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2005.
27. Yuk Lai Suen. Automatically repairing structurally complex data. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, May 2005.
28. United States Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492.
29. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

A Structural invariants for binary tree

The following code gives the `repOk` method for `BinaryTree` (Section 2.1) [4, 22]:

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