Juzi: A Tool for Repairing Complex Data Structures

Bassem Elkarablieh
The University of Texas at Austin
elkarabl@ece.utexas.edu

Sarfraz Khurshid
The University of Texas at Austin
khurshid@ece.utexas.edu

ABSTRACT

This paper describes *Juzi*, a tool for automatic repair of complex data structures. Juzi takes a Java class representing the data structure as well as a predicate method that specifies the structural integrity constraints as inputs. Juzi instruments its inputs and generates a new Java class which behaves similarly to the original class, yet automatically repairs itself when the structural integrity constraints are violated.

Juzi implements a novel *repair* algorithm. Given a structure that violates its integrity constraints, Juzi performs a systematic search based on symbolic execution to repair the structure, i.e., mutate it such that the resulting structure satisfies the given constraints. Experiments on structures ranging from library classes to stand-alone applications, show that Juzi repairs complex structures while enabling programs to recover from erroneous executions caused by data structure corruptions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*

General Terms

Reliability

Keywords

Data structure repair, Assertions, Systematic search, Symbolic execution, Error recovery

1. INTRODUCTION

Programs assume the consistency of their internal data structures. Software bugs can corrupt these structures, violate the consistency assumption, and result in un-predictable program behavior. Assertions have long been used to check for program properties. Data structure integrity constraints can also be represented as assertions; for example, using a Java predicate method that takes the structure as input, traverses it, checks for the structural integrity constraints, and returns whether it satisfies the constraints or not.

The standard approach when an error is detected at runtime, say due to an assertion violation, is to terminate the program, debug it if possible, and re-execute it. An alternative to program termination is *repair*. Instead of terminating the program, use the violated assertion as a basis to repair its state and let it continue.

We have developed the Juzi tool for automatic repair of complex data structures [2, 3, 5]. Given a Java predicate method that describes the structural integrity constraints and a structure that violates the given constraints, Juzi performs *repair actions* that mutate the given structure so that it satisfies the constraints. Juzi repairs corruption in both the reference as well as the data fields of the structure. To repair data fields, Juzi uses forward symbolic execution. To solve the path conditions that arise during symbolic execution, Juzi implements a solver for difference constraints [4], and uses a general integer constraint solver for complex ones. To repair reference fields, Juzi employs an efficient search algorithm that is based on the Korat [1, 9] test generation framework.

To enable the user to understand the mutations performed during repair, Juzi provides an abstraction of the repairs performed. The abstraction specifies the set of fields that are mutated. Such information can help the user debug their program (if the corrupt structure was a result of a bug in the program). The level of details of such abstraction is configured by the user. A key feature of Juzi is enabling the user to control the repair algorithm. Section 3.3 describes the user configurations.

We evaluated Juzi on repairing a variety of structurally complex subjects, including library classes from the Java Collection Framework as well as stand-alone applications. Experimental results show that Juzi efficiently and effectively repairs corrupt structures with a small number of errors and with a few thousands of objects. In addition to repairing a corrupt structure, Juzi enables the program manipulating the structure to continue its execution.

2. EXAMPLE: BINARY TREE

We present an example of repairing a binary tree to illustrate Juzi and the repair algorithm. We illustrate how Juzi can on-the-fly repair faults in the structure as well as the primitive fields of the tree. Consider the following declaration of the `BinaryTree` class.

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

  static class Node {
    Node left;
    Node right;
    Node parent;
  }
}
```

Copyright is held by the author/owner(s).
ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

```
boolean repOk() {
  // An empty tree must have zero in size
  if (root == null) return size == 0;

  // The parent of the root node must be null
  if(root.parent != null) return false;

  Set<Node> visited = new HashSet<Node>();
  visited.add(root);
  LinkedList<Node> workList = new LinkedList<Node>();
  workList.add(root);

  while (!workList.isEmpty()) {
      Node current = workList.removeFirst();
      if (current.left != null) {
        // The tree must have no cycles along left
        if (!visited.add(current.left)) return false;

        // Left's parent must be current
        if (current.left.parent != current) return false;
        workList.add(current.left);
      }

      if (current.right != null) {
        // The tree must have no cycles along right
        if (!visited.add(current.right)) return false;

        // Right's parent must be current
        if (current.right.parent != current) return false;
        workList.add(current.right);
      }
  }
  // Size must be equal to the number of visited nodes
  return visited.size() == size;
}
```

Figure 1: Class invariant of the `BinaryTree` class.

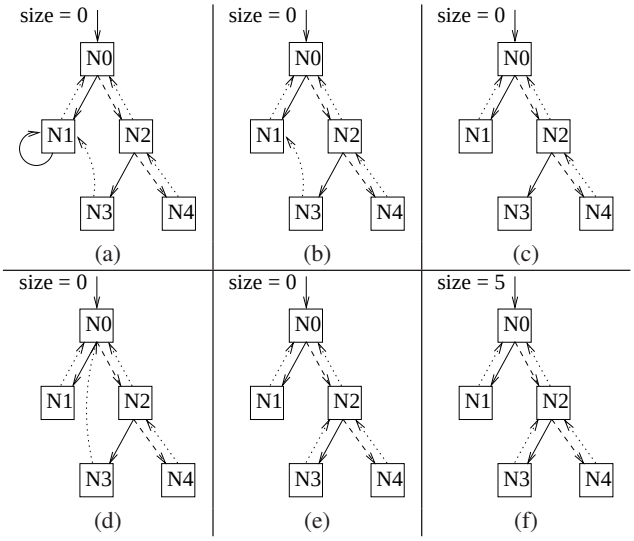

Figure 2: Repairing a binary tree. Solid lines represent `left` fields; dashed lines represent `right` fields; and dotted lines represent `parent` fields. (a) An erroneous tree with the `left` field of node N1 creating a cycle, the `parent` field of node N3 does not point to N2, and the `size` field contains the value 0 rather than 5. (b–f) The steps performed by Juzi to repair the corrupt binary tree.

The internal `Node` class models the nodes in the tree. Each binary tree has a root node and stores the number of nodes reachable from the root in the `size` field. Each `Node` instance holds three `Node` pointers: a `left` pointer which references the root of the left subtree, a `right` pointer which references the root of the right subtree, and a `parent` pointer which references the node's parent.

The structural integrity constraints are: (1) acyclicity along the `left` and `right` pointers, (2) transpose relation between the `parent` and each child pointer (`left` and `right`), and (3) the number of nodes reachable from the root node along the `left` and `right` fields is stored in the `size` field.

Such constraints can be represented as a Java predicate method which we term `repOk` [8] and which traverses the structure and checks for each of the constraints. To illustrate, the `repOk` method for the `BinaryTree` class is displayed in Figure 1.

An assertion can invoke `repOk` to check the structural constraints. For example, the following Java `assert` statement checks them at the beginning of the `traverse` method in `BinaryTree`:

```
String traverse() {
  assert repOk();
  ...
}
```

Instead of terminating the program upon a violation of the `assert` statement, Juzi uses the violated assertion to repair the structure of the tree and continue the program execution.

To illustrate repair, consider the structure shown in Figure 2 (a). The tree has 3 corruptions: (1) the `left` pointer of node N1 creates a cycle in the tree, (2) the `parent` pointer of node N3 does not point to node N2, and (3) the value of `size` is not equal to the number of nodes reachable from the root of the tree.

Given the corrupt structure in Figure 2 (a), and the `repOk` predicate, Juzi first invokes `repOk` on the structure and monitors the field accesses during the execution of `repOk`. When `repOk` returns `false` due to a constraint violation, Juzi systematically mutates the last field accessed by `repOk` by setting it to : (1) `null`, (2) nodes that have already been visited during `repOk`'s execution, and (3) one node that has not yet been visited.

To illustrate, Figures 2 (b–f) show the sequence of mutations that Juzi performs on the corrupt structure. Following the first execution of `repOk`, Juzi first detects the cycle at node N1. The last field accessed is the `left` field of node N1. Thus, Juzi mutates the field's value and sets it to `null`—the first candidate choice—which breaks the cycle (Figure 2(b)) and enables `repOk`'s execution to proceed further. Juzi re-executes `repOk` on the data structure and encounters the fault in the `parent` field of node N3. Juzi first mutates the field to `null` (Figure 2(c)) which does not repair the field. It then mutates the field to tree nodes already encountered during `repOk`'s execution. It first tries N0 (Figure 2(d)) and then N2 (Figure 2(e)) which in this case repairs the `parent` field. Note that Juzi does not try node `N1`, since this choice is the original value of the `parent` field, and it is already tried in the first execution of `repOk`.

Finally, Juzi detects the corruption in the `size` field of the tree. To repair faults in the primitive fields of the structure, Juzi uses on-demand symbolic execution [2,6,7] where a corrupt field is treated symbolically, and a path condition is computed for that field during the execution of `repOk`. Once the path condition is computed, the algorithm uses an integer constraint solver to solve the path condition, and determine the correct value to repair the field. In this example, Juzi sets the value of the `size` field to the number of nodes visited during the execution of `repOk` and in turn generates the repaired structure in Figure 2 (f).

This section illustrated the repair algorithm using an example of repairing a corrupt binary tree. Further details are found elsewhere [2,3,5].

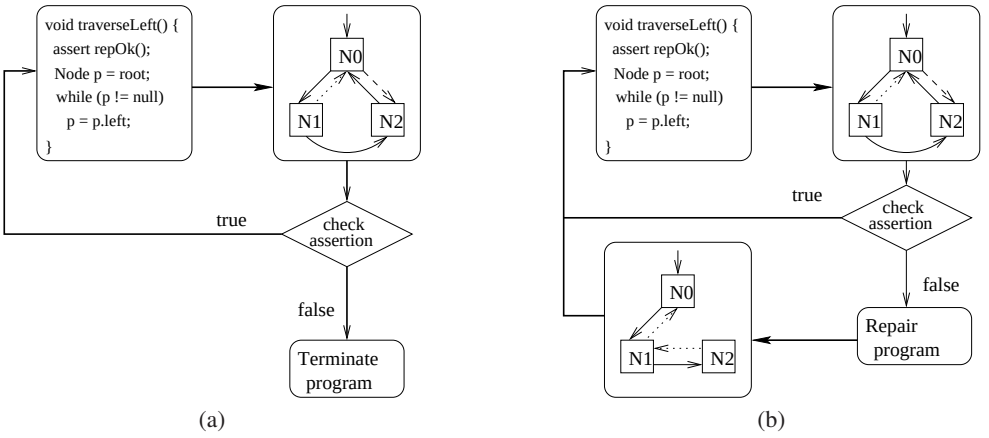

Figure 3: (a) Behavior of the original data structure class upon invoking the `traverseLeft` method; the execution terminates due to an assertion violation caused by the cycle following the `left` field of the tree. (b) Behavior of the Juzi generated class. Upon the assertion violation, the repair algorithm is triggered, and the method execution proceeds after the corrupt structure is repaired.

3. TOOL OVERVIEW

This section describes the main components of Juzi. The heart of Juzi is a code instrumentation module that instruments the given data structure class and predicate method, and generates a new class that enables automatic data structure repair when a corruption occurs at runtime. The generated class represents a repairable data structure where its properties are checked and any corruptions are repaired when its `repOk` predicate method is asserted.

To illustrate, consider the code example in Figure 3. Figure 3(a) describes the result of invoking the `traverseLeft` method on the given structure using an instance of the original class. Due to the cycle in the structure, the original code terminates because of an assertion violation. Figure 3(b) describes the result of invoking the `traverseLeft` method using an instance of the instrumented class. Note how instead of terminating the method execution, the result of asserting `repOk` is a repaired tree which enables the `traverseLeft`'s execution to proceed and safely terminate.

Note that any client code using the original class need not be changed when using the Juzi generated class since Juzi preserves the public interface and the method functionality of the original class. Thus, in case of no corruption the instrumented class behaves as the original class, and the repair algorithm is only triggered when a corruption is detected.

Two key components comprise the instrumentation module of Juzi: instrumentation to support reference field repair, i.e., the search algorithm, and instrumentation to support data repair, i.e., symbolic execution. These components can be configured using an input file provided by the user which enables further user control to the repair algorithm as well as the output of Juzi. We next give an overview of the instrumentation modules, and describe the various ways the user can configure Juzi.

3.1 Search Algorithm

Juzi implements a systematic search algorithm based on the Korat [1, 9] test generation tool. The heart of the search is a backtracking approach that is based on non-deterministic choice. Recall that Juzi monitors the field accesses during `repOk`'s execution and backtracks mutating the last accessed field to its next candidate choice (Section 2).

To monitor the field accesses and the choices made for each field, Juzi implements the `Explorer` [2] class that provides an abstract representation for state space exploration as well as non-deterministic choice. Each field in the structure is associated with a `counter` that is maintained by the `Explorer` class which counts from 0 up to the number of possible candidates to repair a field. Upon backtracking, the counter for the last field is incremented to simulate the last field mutation.

To support non-deterministic choices, the `Explorer` class provides a `choose` method that takes an integer which represents the number of non-deterministic choices and returns an integer which represents one of these choices. For example, the assignment

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

non-deterministically assigns the values $0, 1, 2, 3$ to `x`.

Juzi uses the `Explorer` class to perform non-deterministic reference field assignments. To enable non-deterministic field assignments, Juzi instruments the Java bytecode of both the structure classes and the predicate method.

For each reference field, Juzi adds custom *get* and *set* accessor methods. Each field access in the `repOk` predicate is replaced by a call to the added `get` method. On the first execution of `repOk`, at each field access, the `get` method adds a `counter` in the `Explorer` that specifies the number of possible candidates for a field. This number is equivalent to 2 (null and one non-visited field) + the number of nodes encountered up to that field access. In further executions of `repOk`, the `get` method increments the counter corresponding to the last accessed field and assigns the field to the next choice. All the other fields remain the same. When all the choices of the last accessed field are tried, the field's counter is reset to 0, and the counter of the previous to last field is incremented. The search terminates when the structure is repaired, or when all the candidate choices for all the fields are explored.

3.2 Symbolic Execution

To enable symbolic execution, Juzi replaces type declarations of primitive integers with the library class `SymbolicInt`. The `SymbolicInt` class defines the semantics of the operations on symbolic integers. All the operations over primitive integers are replaced with invocations of library methods that are members of the `SymbolicInt` class. A key requirement of symbolic execution is the ability to cover different program paths. To allow symbolic execution to explore different program paths, Juzi performs a non-deterministic boolean choice whenever there's a branch that

cannot be deterministically resolved on-the-fly, e.g., a conditional branch that involves a symbolic variable. To keep track of the path conditions generated using symbolic execution, Juzi implements a `PathCondition` class that is updated with every non-deterministic choice. At the end of each program path, Juzi checks the satisfiability of the path condition using an integer constraint solver. If the path condition is satisfiable, Juzi solves the path condition and assigns values to the symbolic variables.

3.3 User Control

We next describe the different ways the user can configure the repair algorithm.

3.3.1 Controlling the Fields to Repair

Juzi provides a configuration file for the user to specify what classes to instrument and what fields to repair. This feature allows the user to add more constraints on the repair algorithm, which might be needed in some cases. For example, when the structure needs to have a certain number of nodes, the user can specify not to repair the `size` field and keep it concrete rather than symbolic. In this case, Juzi cannot modify the size field to satisfy other constraints, and if the size property is not satisfied, Juzi reports the structure as non-repairable.

3.3.2 Controlling Data Repair

Data repair is a challenging problem for repair. To illustrate, 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 choice is unlikely to be a good one, since it might end up corrupting all the information in the tree. Juzi gives the user some control on how to repair the data. (1) The user can specify ranges of data values for primitive fields and use these ranges to constrain the repair algorithm. (2) The user can select which type of constraint solver to use for solving the path conditions. Constraints on the order of data can be solved using a difference constraint solver which reorders the data elements without mutating the values. Other complex constraints require more complex solvers to repair the data values.

3.3.3 Controlling the Search Algorithm

The search algorithm monitors the fields as they are accessed by `repOk`. Repair actions performed by Juzi depend on how `repOk` is formulated. Thus, two different `repOk` implementations that access fields in different orders may cause Juzi to produce different structures. This 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 deems unlikely to get corrupted) unless necessary.

4. EVALUATION

We evaluated Juzi by applying it on seven subjects, including two from standard Java libraries, as well as two stand-alone applications. For brevity, we select the binary tree described in Section 2 as a representative structure and we study the time required to repair a tree with 2000 nodes and with up to 20 faults. Further evaluation details are found in [2, 3].

Figure 4 shows the variation in repair time for a binary tree with 2000 nodes as the number of faults in the structure increases. The algorithm requires less than 20 seconds to repair 20 faults in the structure. The results show the feasibility of repairing modestly large structures with few number of corruptions.

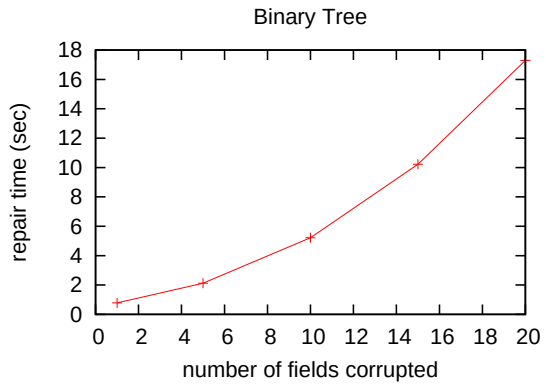

Figure 4: Time to repair a binary tree with 2000 nodes as the number of corrupt fields increases.

Acknowledgments

This work was funded in part by grants #CCF-0702680 and #IIS-0438967 awarded by the National Science Foundation.

5. REFERENCES

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

[2] Bassem Elkarablieh, Iván García, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of structurally complex data. In *Proc. 22th Conference on Automated Software Engineering (ASE)*, November 2007.

[3] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn McKinley. Starc: Static analysis for efficient repair of complex data. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2007.

[4] Bassem Elkarablieh, Yahya Zayour, and Sarfraz Khurshid. Efficiently generating structurally complex inputs with thousands of objects. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, July 2007.

[5] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *Proc. 12th SPIN Workshop on Software Model Checking*, 2005.

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

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

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

[9] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 771–774, Washington, DC, USA, 2007.