DSDSR: A Tool that Uses Dynamic Symbolic Execution for Data Structure Repair

Ishtiaque Hussain, Christoph Csallner
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, TX 76019, USA
ishtiaque.hussain@mavs.uta.edu,csallner@uta.edu

ABSTRACT

We present *DSDSR*, a generic repair tool for complex data structures. Generic, automatic data structure repair algorithms have applications in many areas. Reducing repair time can may therefore have a significant impact on software robustness. Current state of the art tools try to address the problem exhaustively and their performance depend primarily on the style of the correctness condition. We propose a new approach and implement a prototype that suffers less from style limitations and utilizes recent improvements in automatic theorem proving to reduce the time required in repairing a corrupt data structure. We also present experimental results to demonstrate the promise of our approach for generic repair and discuss our prototype implementation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery, symbolic execution*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, class invariants, reliability*

General Terms

Algorithms, Reliability, Verification

Keywords

Data structure repair, data structure invariants, dynamic symbolic execution

1. INTRODUCTION

Generic repair of complex data structures is a new approach to software robustness [5, 4, 6, 7, 10]. It promises to mutate the state of a running program in such a way that the resulting state satisfies a given assertion or correctness condition. It is generic in the sense that a single repair algorithm can repair many kinds of data structures. It thereby differs from traditional repair, as in traditional repair for each kind of data structure we need a separate repair algorithm. This makes generic repair potentially very powerful. Indeed, initial generic repair techniques [5, 6] and implementations such as Juzi [7] are very promising.

Besides runtime data structures and their respective correctness conditions, a generic repair algorithm does not need any additional inputs. This makes the correctness conditions the focal point of generic repair. However, it is well-known that writing correctness conditions is hard [1, pages 371–373]. I.e., a correctness condition has to satisfy several criteria. First, it has to check the right properties. Second, it has to be implemented correctly. Third, it should be implemented in a way that is easy to read and understand. In addition to these criteria, current approaches to generic repair make writing correctness conditions even harder, by adding another requirement. I.e., in addition to the above three criteria, a correctness condition also has to be written in a repair-specific style. If a correctness condition does not satisfy these style requirements, it may render current generic repair approaches inefficient or ineffective—even if the correctness condition checked the correct properties, checked them correctly, and was readable.

In [9] we motivated how dynamic symbolic techniques enable generic repair to support a wider range of correctness conditions, presented a prototype that implements the proposed algorithm and initial empirical results. In this paper, we make the following contributions:

- We describe the algorithm and an implementation in further detail.
- We present more experimental data, demonstrating the promise of our approach for repairing data structures independent of the style of the data structure correctness condition.

We describe our implementation in terms of object-oriented software and especially Java programs, but the algorithm equally applies to related languages (C++, C#, etc.) and related programming paradigms (i.e., functional and procedural languages).

2. MOTIVATING EXAMPLE

To illustrate the approach of our algorithm and how it improves upon the state of the art, we now consider the binary tree data structure given in Figure 1. We took the simplified correctness condition from [7] and the constraints for the binary trees are: (1) acyclic along the left and right pointers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WODA 2010 Trento, Italy
Copyright 2010 ACM 978-1-4503-0137-4/10/07 ...$10.00.

```
public class Node {
 Node left;
 Node right;
 // ..
}

public class BinaryTree {
 Node root;
 int size;
 // ..
 public boolean repOk() {
  // An empty tree must have zero in size
if (root == null)
 return (size == 0);

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;
  } else
   workList.add(current.left);
 }

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

// Size must be equal to #visited nodes
return (visited.size() == size);
 }
}
```

Figure 1: Example binary tree data structure, abbreviated, consisting of a Node class and a BinaryTree class. Method repOk is a contrived correctness condition for the binary tree, which may be invoked by assertions throughout the program.

and (2) the number of nodes reachable from the root node along the left and right fields is stored in the size field. To emphasize the fact that repair actions heavily depend on the writing style of the correctness condition, we slightly modified the correctness condition of Figure 1 creating another version as in Figure 5. This modified version of repOk produces the same result as the original one but only differs in the style that whenever it discovers a corruption, stores the result in a temporary boolean variable named *result* and returns the desired answer only at the end of the method.

Figure 2 (a) shows an example binary that consists of five nodes. The first node has a corrupt value in its left field, namely it points to the root node creating a cycle. Note that, in the modified repOk of Figure 5, the *size* field of the binary tree is always the last accessed field. To repair the corruption, Juzi first (b) tries to mutate this field- the last accessed field of the repOk. Failing to repair the data structure, subsequently (c - g and omitted from the figure), Juzi backtracks in the list of fields accessed by the repOk method and continues repair actions in an exhaustive fashion trying all possible mutations for each field. Finally, Juzi reaches

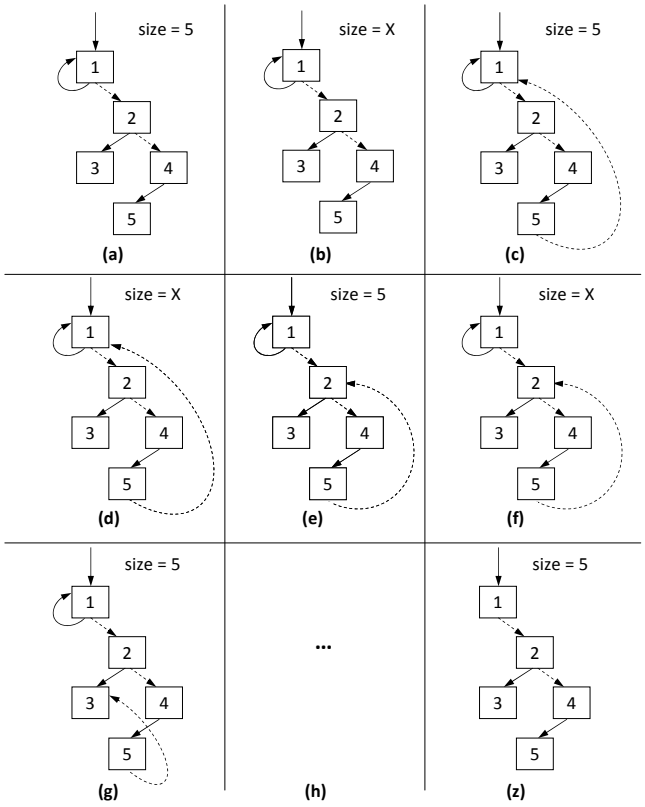

Figure 2: Exhaustive approach in Juzi. Initially (a), the binary tree is corrupt. Dotted lines and X in *size* field show repair attempts (b - g). Omitted are several subsequent repair attempts (h). Ultimately repair culminates in the binary tree (z). Note that there are no fresh objects in Juzi.

a field that the repOk method had accessed very early, the corrupt left field of the first node, and now Juzi repairs the binary tree successfully (z). For each repair attempt Juzi executes the repOk method, to check if the resulting list satisfies the repOk correctness condition.

In situations like this, an exhaustive approach works well for repairing small data structure instances, containing few nodes. But when repairing larger structures, at some point exhaustive search becomes inefficient. The number of possible mutations grows exponentially and most mutations do not result in a correct state.

Our key insight is that we can guide data structure repair by mutating the data structure in such a way that the repaired data structure takes a predetermined execution path. In our example, we want to invert the outcome of the if-condition just after which the temporary variable *result* got assigned such that, instead of returning false, repOk returns true. For multiple corruptions, this *result* variable will be assigned false multiple times. In such cases, we take the last if-condition after which *result* is finally assigned false. In our example, this does not occur as there is only one corruption in the data structure. Indeed, if we take the path condition of the original path, which returned false, invert that particular conjunct and ignore the following conjuncts, and solve the resulting path condition, we can obtain the

correct repair action directly.

3. PROPOSED APPROACH, ALGORITHM AND IMPLEMENTATION

Our tool, DSDSR consists of two parts. At the lower level is a dynamic symbolic execution engine that has a broad interface to allow modification of path conditions, etc. At the top layer sits our generic repair algorithm. In this section we briefly describe both components and the implementation.

3.1 Dynamic Symbolic Execution Engine

Our dynamic symbolic execution engine automatically inserts instrumentation code into a given repOk method, which yields an instrumented version of repOk. The execution of the instrumented version behaves just like the original, except that it also creates a symbolic representation of the program execution state. In that our engine is similar to previous ones such as Dart, jCute, and Pex [8, 12, 13]. When we apply dynamic symbolic execution to repOk, we obtain a complete symbolic representation of the path taken by the repOk correctness condition. For example, we now have a symbolic representation of the if-condition, whose concrete value resulted triggered repOk to return false.

3.2 Algorithm for Data Structure Repair

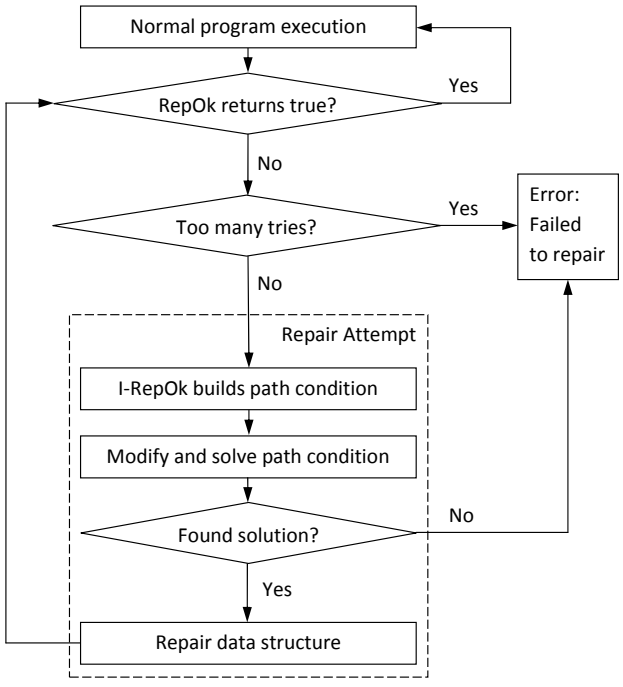

Figure 3: Overview of our dynamic symbolic data structure repair algorithm (DSDSR). RepOk is a method that implements a given correctness condition. I-RepOk is the instrumented version of repOk.

Figure 3 gives a high-level overview of our algorithm. As part of its normal execution, a program invokes assertions or other methods that implement a correctness condition. In our description we follow previous work and name such a method repOk [6, 7]. When the correctness condition fails, i.e., repOk returns false, execution is handed over to our extended dynamic symbolic engine, which in turn invokes the instrumented version of repOk. Executing the instrumented repOk builds the path condition of the execution path that leads to the point at which repOk failed.

With the full symbolic path condition in hand, we can now modify the path condition to obtain a different path. I.e., if we invert or negate the last if-condition, we obtain a path that does not return false at the point at which the original execution failed. Also, for our example, if we invert the last if-condition before *result*'s assignment and ignore the following conditions, we also obtain a path that does not return false. At the same time, solving such a new path condition can give us an input state that will trigger the new path. If the new state satisfies the repOk correctness condition, we can mutate the existing state to resemble the new one, which completes the repair.

The algorithm relies on a faithful encoding of the path condition and other program constraints in a format suitable for automated reasoning. It further relies on a powerful automated constraint solver that can simplify such constraints and, if a solution exists, can produce a concrete solution. Finally, the solution of the constraint solver needs to be mapped back into the program state, to repair the existing data structures.

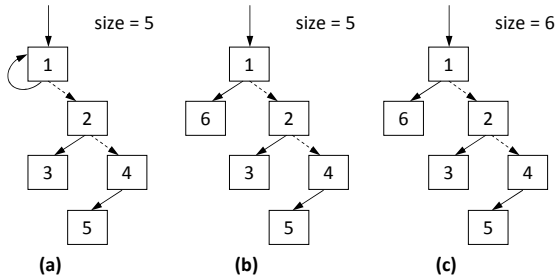

Figure 4: Directed approach in DSDSR. Solid lines represent *left* field and dashed lines represent *right* field. Initially (a), the binary tree is corrupt, as the first node's left field points to the root, creating a cycle, which is incorrect according to repOk correctness condition. In first attempt (b), DSDSR creates a new node for left field. Finally, in (c), updates the *size* field to reflect the repair.

We repair the data structure according to the solution of the constraint solver and invoke repOk to check if the resulting structure satisfies the repOk correctness condition. If repOk again returns false, we may make another iteration and attempt another repair. In our example, the first solution was a new node for the left field of the root (Figure 4(b)). But resulting structure still did not satisfy repOk due to mismatch in total number of nodes in size field. A second iteration finally repaired the data structure (Figure 4(c)). At this point note that, Juzi only works with existing objects whereas our tool can suggest new objects as solution.

To prevent an infinite loop of repair attempts, the algorithm terminates after reaching a user-defined number of futile attempts. If the repOk method returns true, we consider the repair attempt to be successful and resume normal program execution.

The main advantage of our approach is that, unlike Juzi, in the search for a data structure that satisfies a repOk correctness condition, we do not need to exhaustively generate all possible candidate data structures. Instead, DSDSR derives conditions directly from the repOk implementation to generate a single data structure that satisfies the correctness condition.

3.3 Implementation

We implement our generic repair algorithm on top of our new dynamic symbolic execution engine for Java, called Dsc. Dsc works on top of any standard Java virtual machine. It does not require modifications of the virtual machine or the user code. This means we can repair existing Java code when it is executed on a standard JVM.

Dsc analyzes user code at the bytecode level. It uses the instrumentation facilities provided by Java 5 to instrument user code at load-time, using the ASM bytecode instrumentation framework [3, 2].

$fieldAccesses \leftarrow emptyList$;
$instanceAccesses \leftarrow emptyList$;
$constraints \leftarrow emptyList$;
$lastFieldAccessed \leftarrow -1$;
$lastInstanceAccessed \leftarrow -1$;
$corruptConstraint \leftarrow -1$;

$repOkExec1(repOk)$;
$updateMaps(root)$;
$repOkExec2(repOk)$;

negate $constraints[corruptConstraint]$;
solve $constraints$;
update data structure with the solution;

Algorithm 1: Main Algorithm

We explain the implementation details with the help of our motivating example, the binary tree data structure given in Figure 4 and the correctness condition of Figure 5. Recall that when the correctness condition fails, i.e., repOk returns false, execution is handed over to the extended dynamic symbolic engine, which in turn invokes the instrumented version of repOk. It calls the instrumented version of repOk twice, as shown in Agorithm 1, and the purpose of these two executions are twofold:

- Explore the state space to collect meta information of the data structure and identify the corrupt instance and field.
- Build an appropriate path condition, by negating the constraint that represents the corruption in the data structure.

Meta information includes the subtype and supertype relation, the dynamic type of objects, types of referenced objects, visibility, etc. Algorithm 2 describes the handler that interprets the first call of the instrumented repOk method. It mainly detects the corrupt instance and corrupt field of the data structure. Ideally, there should not be any assignment statements in correctness conditions, so for any such assignment statement at a program point P, as in line 19 in Figure 5, we wrap the value with two indices: the last field and instance accessed prior to program point P and push it to the operand stack. At the end of this call, when repOk finally returns false, either by unwrapping the operand stack top value e or simply getting the maximum index of *fieldAccesses* and *instanceAccesses*, *lastFieldAccessed* and *lastInstanceAccessed* hold the indices to the field and instance accessed last. In our example, as shown in Figure 5 in line 37, when repOk returns false using temporary variable *result*, we deduce that the first node (node 1) is the corrupt instance and its *left* field is the corrupt field.

while *more statements* **do**
  match *statement* **do**
    case *field read:*
      resolve accessed instance and field;
      add to $fieldAccesses$;
      add to $instanceAccesses$;
    case *local variable read:*
      if *local variable maps to field access* **then**
        resolve accessed instance and field;
        add to $fieldAccesses$;
        add to $instanceAccesses$;
    case *($x \leftarrow e$):*
      wrap e with pointers to maximum index of $fieldAccesses$, $instanceAccesses$;
    case *return e:*
      if *operand stack top e is wrapped* **then**
        unwrap the e and get pointers:
        $fieldAccessedPointer$
        $instanceAccessedPointer$;
        $lastFieldAccessed \leftarrow$
        $fieldAccessedPointer$;
        $lastInstanceAccessed \leftarrow$
        $instanceAccessedPointer$;
      else
        $lastFieldAccessed \leftarrow$
        maximum index of $fieldAccesses$;
        $lastInstanceAccessed \leftarrow$
        maximum index of $instanceAccesses$;

Algorithm 2: repOkExec1(Method repOk)

Between the two repOk calls, Algorithm 3 uses Java's reflection mechanism to traverse the data structure and map the value of each reference field to our symbolic representation of object fields. But to allow the constraint solver to compute a solution, we do not assert such constraints for the corrupt instance and field.

Algorithm 4 describes the interpreter for the second call of the instrumented repOk method, which builds the path condition, incorporating all the meta information. For similar reasons as in Algorithm 2, for any assignment statement we wrap the value e with the maximum index of *constraints* at that point, which basically points to the constraint that represents the corruption. At the end, before returning, if the stack top value e is wrapped, we unwrap it to get the index to the corrupt constraint or uses the maximum index of the *constraints*. For each branch condition, we create a corresponding constraint to represent the outcome of the branch condition. At the end of the execution, the conjunction of all such constraints yields the path condition. For example, in Figure 5 with root node (node 1), line 3 creates a constraint : *notNull (node 1)*, line 16 creates *notNull((node 1).left)* and from Set's add method in line 18 (in our example, the last if-condition before the *result* is assigned false) Dsc creates

```
classes ← emptySet;
objects ← emptySet;
objects.add(root);
worklist ← emptyQueue;
worklist.enqueue(root);
corruptField ← fieldAccesses[lastFieldAccessed];
corruptInstance ←
instanceAccesses[lastInstanceAccessed];
while worklist not empty do
    obj ← worklist.dequeue();
    classes.add(class(obj));
    foreach field f in refFields(obj) do
        if obj!=corruptInstance AND f!=corruptField
        then
            refObj ← get referenced object in f of obj;
            update map for f at index obj with refObj;
            if objects.add(refObj) then
                worklist.enqueue(refObj);
foreach class in classes do
    foreach field in refFields(class) do
        assert map for field;
```

Algorithm 3: updateMaps(Object root)

yet another constraint *(node1) == (node 1).left*. At the very end of this algorithm, Dsc uses *corruptConstraint* to negate this constraint, resulting in *(node 1) != (node 1).left* and in conjunction with the previously formed constraints, we finally have the path condition to solve.

```
while more statements do
    match statement do
        case branch:
            create constraint of outcome;
            add constraint to constraints;
        case (x ← e):
            wrap e with pointer to maximum index of
            constraints;
        case return e:
            if operand stack top e is wrapped then
                unwrap the e and get pointer:
                ptConstraint;
                corruptConstraint ← ptConstraint;
            else
                corruptConstraint ←
                maximum index of constraints;
```

Algorithm 4: repOkExec2(Method repOk)

4. PRELIMINARY RESULTS

We conducted experiments with binary trees of different sizes. Each run constructed a correct binary tree of a given size, corrupted one of the leaf node's left or right field by pointing the root as its child, invoked one of the repair tools, and measured the time the tool takes to repair.[1] To emphasize the fact that writing good correctness condition is very

[1]The current version of our prototype makes only one repair attempt. For cases where we needed multiple repair actions to finally correct the data structure, we ran the tool multiple times—adding the time taken each time to simulate the final repair.

```
public boolean repOk() {
 boolean result = true;
 if (root == null) {
  if (size != 0)
   result = false;
  return result;
 }

 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)) {
    result = false;
   } else
    workList.add(current.left);
  }

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

 // Size must be equal to #visited nodes
 if (visited.size() != size)
  result = false;

 return result;
}
```

Figure 5: Modified correctness condition. It produces the same output as the corresponding correctness condition of Figure 1. However, it stores the status of the check temporarily in a local variable and returns it at the end.

hard and repair actions heavily depend on the programmer defined correctness condition, we considered slightly different versions of the same correctness condition and applied both the repair tools. We considered three versions of the same correctness condition. They vary in places of return statements, e.g., the first one returns false immediately when it detects a corruption (Figure 1) , second one waits until the end of the method (Figure 5) and the last one returns false immediately if the corruption is in the right pointer but returns late if it is in the left pointer and vice versa.

As before [9], We conducted experiments with the latest version of Juzi (0.0.0.1) which we obtained from the Juzi website[2] and took all measurements on a Sun HotSpot JVM 1.6.0_17 running on Windows on an intel laptop 2.26GHz Core2 Duo processor.

Figures 6, 7, and 8 show the result of our experiment. For a carefully designed correctness condition, Juzi repairs more efficiently than our prototype implementation, as shown in Figure 6. But with a modified correctness condition, as shown in Figure 7, starting with 5 nodes, our approach is more efficient.

Figure 8 shows the result of our experiment with third version of repOk that returns false immediately if the corruption is in the right pointer but returns late if the corruption

[2]http://users.ece.utexas.edu/~elkarabl/Juzi/index.html

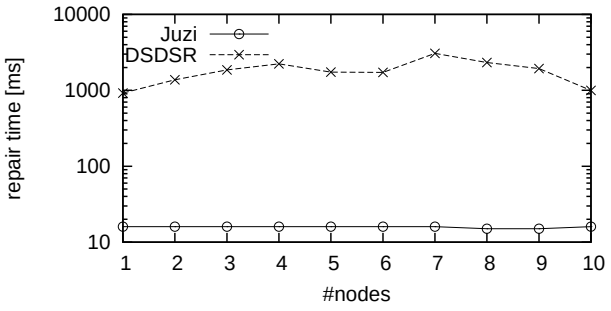

Figure 6: Repair time for binary trees of different sizes and immediately returning correctness condition (Figure 1). #nodes is the number of nodes in the binary tree. Repair time is the time a tool took to produce the correct repair action. Smaller repair times are better.

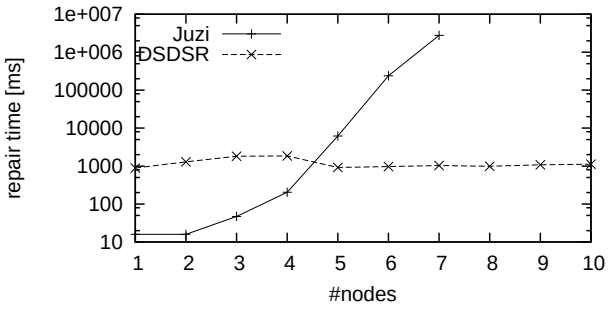

Figure 7: Repair time for binary trees of different sizes and late returning correctness condition (Figure 5).

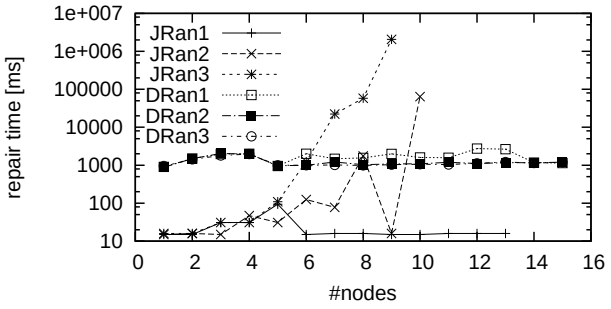

Figure 8: Repair time for three sets of random binary trees and yet another variation of the correctness condition. We had to terminate Juzi for 10, 11, or 14 nodes.

is in the left pointer. We found similar results for the opposite case of this repOk, which returns immediately for left field corruption but returns late if corruption is in the right field. We applied both the tools for three sets of binary trees, where each set had randomly built binary trees containing from 1 to 15 nodes. In each case, we had to terminate Juzi prematurely because it was taking too long to repair. This is intuitively expected as an exhaustive approach such as Juzi is bound to be inefficient for larger data structures. This motivates our more directed approach, which takes approximately the same amount of time to repair—irrespective of the size of the data structure and variation in the correctness conditions.

5. RELATED WORK

Generic data structure repair, pioneered by Demsky and Rinard [5], is a relatively new area of research. Non-generic data structure repair is not new, classic examples include the IBM MVS/XA operating system [11].

Acknowledgments

We thank Bassem Elkarablieh and Sarfraz Khurshid for helping us with Juzi.

6. REFERENCES

[1] B. Bruegge and A. H. Dutoit. *Object-oriented software engineering: Using UML, patterns and Java*. Third edition, Aug. 2009.

[2] É. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proc. ACM SIGOPS France Journées Composants 2002: Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Nov. 2002.

[3] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with Joie. In *Proc. USENIX Annual Technical Symposium*, pages 167–178. USENIX, June 1998.

[4] B. Demsky. *Data Structure Repair Using Goal-Directed Reasoning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[5] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proc. 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 78–95. ACM, Oct. 2003.

[6] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 64–73. ACM, Nov. 2007.

[7] B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 855–858. ACM, May 2008.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[9] I. Hussain and C. Csallner. Dynamic symbolic data structure repair. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Emerging Results Track*. ACM, May 2010. To appear.

[10] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2009.

[11] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering (TSE)*, 13(10):1135–1139, Oct. 1987.

[12] K. Sen and G. Agha. Cute and jCute: Concolic unit testing and explicit path model-checking tools. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 419–423. Springer, Aug. 2006.

[13] N. Tillmann and J. de Halleux. Pex - white box test generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*, pages 134–153. Springer, Apr. 2008.