STARC: Static Analysis for Efficient Repair of Complex Data

Bassem Elkarablieh Sarfraz Khurshid
University of Texas at Austin
{elkarabl,khurshid}@ece.utexas.edu

Duy Vu Kathryn S. McKinley
University of Texas at Austin
{duyvu,mckinley}@cs.utexas.edu

Abstract

Data structure corruptions are insidious bugs that reduce the reliability of software systems. Constraint-based data structure repair promises to help programs recover from potentially crippling corruption errors. Prior work repairs a variety of relatively small data structures, usually with hundreds of nodes.

We present STARC which uses static analysis to repair data structures with tens of thousands of nodes. Given a Java predicate method that describes the integrity constraints of a structure, STARC statically analyzes the method to identify: (1) the *recurrent fields*, i.e., fields that the predicate method uses to traverse the structure; and (2) *local field* constraints, i.e., how the value of an object field is related to the value of a neighboring object field. STARC executes the predicate method on the structure and monitors its execution to identify corrupt object fields, which STARC then repairs using a systematic search of a neighborhood of the given structure. Each repair action is guided by the result of the static analysis, which enables more efficient and effective repair compared to prior work. Experimental results show that STARC can repair structures with tens of thousands of nodes, up to 100 times larger than prior work.

STARC efficiency is probably not practical for very large data structures in deployed systems, but opens a promising direction for future work.

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 Static analysis, data structure repair, symbolic execution

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.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010…$5.00

1. Introduction

As software systems are growing in complexity, reliability is becoming harder to achieve. Efforts to improve reliability are directed toward design, testing, and validation [2]. All these activities are performed before the software is deployed. Such approaches are fundamental for providing a certain level of confidence in program correctness and robustness, yet they do not prevent errors and anomalies from occurring dynamically in deployed software. For example, a single bit flip, say due to a cosmic ray, can compromise the safety of a Java Virtual Machine with high probability and allow an intruder to run arbitrary code [19].

Usually, when an error occurs in a running application, programmers terminate the application, debug, test, and redeploy it. While this halt-on-error approach is sometimes necessary, e.g., during the execution of a security protocol, there are situations where alternative approaches are more desirable. For example, with corruption of persistent data, such as a file system, a simple reboot is unlikely to help. As another example, consider an intentional naming server for service location in a dynamic network [1]. If the server fails due to a malformed query, a continual subjection to the query will force perpetual failures. This problem compounds for deployed software, which cannot be promptly debugged and re-installed.

An alternative to halt-on-error is to repair the state of the program and let it continue. In several cases, this alternative allows systems to resume their correct behavior. For example, a server that does not crash on a malformed query but repairs it, can continue to correctly resolve well-formed ones. Similarly, repairing a file system or a database can recover valuable data. For deployed software, repair, even if it degrades performance, is an attractive alternative.

Traditionally, special routines perform repairs [20, 36]. Recently, generic repair techniques have been introduced [10, 11, 12, 13, 28]. These techniques use constraints (given as first-order logic formulas, imperative predicates, or assertions), which describe desired properties of program states, as a basis for repairing corrupt states that violate the properties. While generic techniques repair a variety of complex data structures, their execution time limits them to relatively small structures with up to a few hundred nodes.

This paper presents STARC, an *assertion-based* framework for efficient and effective repair of data structures, that may consist of tens of thousands of objects. STARC systematically explores a neighborhood of the given corrupt structure using a backtracking search [8, 18, 21, 22] and performs *repair actions*, i.e., mutations of object fields, to transform the structure into one that satisfies the desired assertion. STARC draws its key strength from a static analysis that enables it to perform repair actions that are more likely to correct the corruption.

Given a Java predicate method that represents the structural integrity constraints, the static analysis identifies two key characteristics. One, it identifies a set of *recurrent fields*, i.e., fields that the Java predicate primarily uses to traverse its input structure, using a forward data-flow analysis [6]. Two, it identifies a set of *local field constraints*, i.e., how the value of an object field is related to the value of a neighboring object field, using symbolic execution [17]. STARC uses the result of the static analysis to (1) prioritize the order of repair actions based on the role of the corrupt field (recurrent or not), which makes efficient local repairs of corrupt fields and (2) monitor field accesses based on their relationship with their neighboring fields, which enables effective pruning of the search space.

We evaluate our repair algorithm by using STARC to repair a set of complex data structures that violate their structural integrity constraints. Experiments show that STARC feasibly repairs faulty structures with tens of thousands of nodes. These results show that constraint-based data structure repair is promising and may be able to reach the performance required for on-the-fly repair of in deployed systems.

We make the following contributions:

- **Static analysis for repair**. We use static analysis for efficient search-based repair.
- **Algorithm to repair data structures**. We present an algorithm that builds on previous work, and combines static analysis with systematic search to enable efficient and effective repair of structures using imperative predicates.
- **Implementation**. We present the STARC tool that implements our repair algorithm.
- **Evaluation**. We evaluate our implementation using a variety of subjects and present experimental results that show two orders of magnitude improvement over the previous work.

2. Examples

In this section, we present two examples to describe the use of our repair algorithm. The first example is a circular doubly linked list. This example illustrates how STARC can repair faults in the structure of the list, and how static analysis improves the performance of repair. The second example is an acyclic binary search tree. This example illustrates how

```
boolean repOk() {
       // if the list is empty, size must be 0
L1.    if (header == null)
L2.        return size == 0 ;
L3.    Set visited = new HashSet();
L4.    visited.add(header);
L5.    Node current = header;

L6.    while (true) {
L7.        Node n = current.next;
           // next fields must not be null
L8.        if (n == null)
L9.            return false;
           // prev must be the transpose of next
L10.       if (n.prev != current)
L11.           return false;
L12.       current = n;
L13.       if (!visited.add(n))
L14.            break;
        }
        // reachability constraint
L15.    if (visited.size() != size)
L16.        return false;
L17.    return true;
    }
```

Figure 1. Class invariant for the `DoublyLinkedList`.

STARC repairs errors in the structure, as well as errors in the values of the primitive fields of the tree to satisfy its data constraints.

2.1 Doubly Linked List

Consider the following class declaration of a circular doubly linked list:

```
class DoublyLinkedList {
    Node header;
    int size;

    static class Node {
        int element;
        Node next;
        Node prev;
    }
}
```

The inner class `Node` models the entries in a list. Each list has a `header` and a `size` field, and each node has an integer element and two node pointers (`next` and `prev`). The `size` field represents the number of unique node objects reachable from the header node by following the `next` (or the `prev`) fields of the nodes.

The structural integrity constraints (class invariant) of `DoublyLinkedList` are: (1) circular structure along `next`; (2) transpose relation between the `next` and `prev` fields; and (3) number of nodes reachable from the `header` field following `next` is cached in `size`.

The structural constraints of the `DoublyLinkedList` can be written as a predicate that returns `true` if and only if its input satisfies all the constraints. Following the literature, we term such a Java predicate `repOk` and for object-oriented programs, we term structural invariants, class invariants [33]. The class invariant for the `DoublyLinkedList` class is displayed in Figure 1.

To repair a doubly linked list, STARC analyzes the `repOk` predicate and detects that `next` is the recurrent field, and that the `prev` field is always equal to the transpose of the `next`

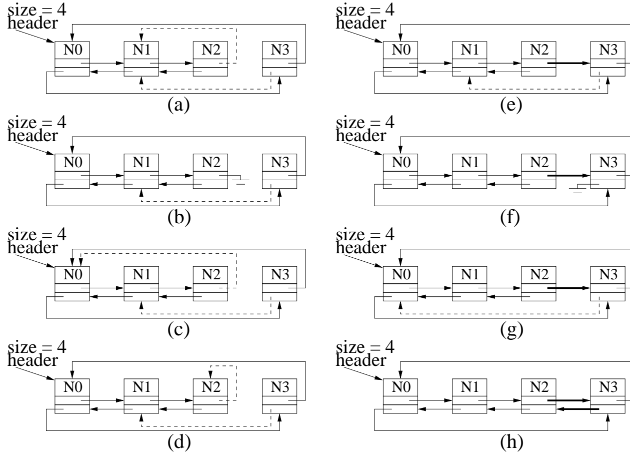

Figure 2. Repairing a circular doubly linked list. The dashed lines represent violations of the structural constraints of a doubly linked list. (a) An erroneous structure with two corruptions in the `next` field of node `N2` and the `prev` field of node `N3`. (b - g) Mutations that an assertion based repair algorithm performs to repair the structure.

field. Then, given an erroneous structure STARC mutates the structure to satisfy the constraints described in `repOk`. STARC uses static analysis to scale the performance of previous assertion based repair algorithms [13, 17, 28, 39].

To illustrate, consider the `DoublyLinkedList` instance in Figure 2(a). This list violates both the transpose constraint between `next` and `prev` fields of nodes `N2` and `N3`, as well as the reachability constraint where the `next` field of node `N2` points to the node `N1` rather than node `N3`. Figure 2(b-g) shows the steps and mutations that an assertion-based repair tool, Juzi [28], performs to repair the structure. To repair the corruption in the `next` field of node `N2`, Juzi first sets the field to `null` (Figure 2(b)). If `null` does not repair the corruption, Juzi tries all the previously visited node during traversal, (Figure 2(c,d)). Finally, Juzi tries one new non-visited node (Figure 2(e)) which in this example repairs the corruption in the `next` field. Juzi applies the same procedure to repair the `prev` field, and thus, performs a total of 7 repair actions in order to repair the corrupt list.

STARC, on the other hand, only performs 2 mutations, Figures 2(e) and 2(h), to repair the corrupt structure. STARC uses the static analysis result to prioritize the order of the mutations on the corrupt field. For a recurrent field, STARC gives a higher priority for selecting a new non-visited node rather than a visited one or `null`, since recurrent fields are used for traversal, and are highly likely to point to a new node. For this example, STARC first sets the `next` field of node `N2` to the new non-visited node `N3`, and thus, repairs the `next` field in one try. STARC also detects the transpose relation between the `prev` and the `next` field using static analysis. Using this information, STARC directly sets the `prev` field of node `N3` to node `N2`, and repairs the structure.

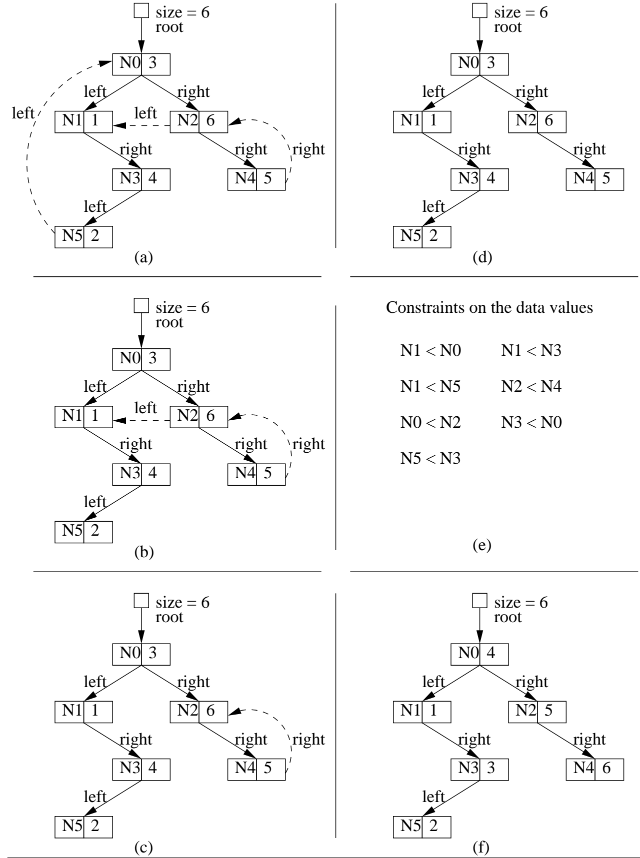

Figure 3. Repairing a binary search tree. The dashed lines represent violations of the structural constraints of a binary search tree. (a) An erroneous tree with four faults in the structure, and faults in the data. (b-d) The steps that STARC takes to repair the structure of the tree, i.e., break all the cycles. (e) The constraints on the order of the data computed using symbolic execution. (f) Resulting tree after applying our repair algorithm.

2.2 Binary Search Tree

The `DoublyLinkedList` example illustrates the use of STARC to repair faults that violate the structural constraints of a list. We now present an example that illustrates the use of STARC in generating primitive values that satisfy the data constraints of a data structure.

Consider the following class declaration of a binary search tree, i.e., an acyclic graph that satisfies the search constraints on the values of its nodes:

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

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

Each `BinarySearchTree` object has a `root` node and stores the number of nodes in the field `size`. Each `Node`

object has an integer value called `elem` and has a `left` and a `right` child. The class invariant of `BinarySearchTree` can be formulated as follows.

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

When invoked on a `BinarySearchTree` object *o*, the predicate `repOk` traverses the object graph rooted at *o* and checks all the constraints that characterize a binary search tree. If any constraint is violated the predicate returns `false`; otherwise, it returns `true`. The implementation of the helper methods is presented elsewhere [14].

To collect constraints on the order of the data in a binary search tree, STARC uses symbolic execution [31]. After repairing the faults in the structure of the tree, STARC solves the data constraints and generates values that complete the repair of the binary search tree.

To illustrate, consider the binary search tree in Figure 3(a). The dashed lines represent fields that violate the acyclicity constraints. Figures 3(b-d) show the steps that STARC takes to break the cycles in the structure. Following a depth first traversal which accesses the `left` field before the `right` field, STARC breaks a cycle each time it encounters an already visited node. Figure 3(e) shows the path condition after symbolically executing `repOk` on the repaired structure. The path condition contains the constraints on the order of the data values. Figure 3(f) shows the repaired structure after solving the path condition and reordering the values in the tree.

This example illustrated how STARC repairs faults in the structure as well as data. However, STARC could be configured not to alter the data values, if the developers preferred to only repair the structure (Section 6.5).

3. Background

This section gives a brief description of forward symbolic execution and search-based repair.

3.1 Forward Symbolic Execution

Forward symbolic execution is a technique for executing a program on symbolic values [31]. 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; }
```

```
boolean repair(Object s, Pred repOk) throws Exception {
    Search.initialize(s);
    PathCondition.initialize();
    boolean done = false;
    do {
        try {
            if (repOk.invoke(s)) {
                if (!PathCondition.isFeasible())
                    continue;
                done = true;
                break;
            }
        } catch (Exception e) {
            if (e.getClass() != BacktrackException.class)
                throw e;
        }
    } while (Search.incrementCounter());
    return done;
}
```

Figure 4. Search-based repair algorithm.

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

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

Note that for each explored path, 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.

3.2 Search Based Repair

This section describes Juzi [13, 17, 28, 39], a search based algorithm for assertion-based repair. Given a structure `s` and a `repOk` method that represents desired structural integrity constraints such that `!s.repOk()`, Juzi performs mutations on `s` to transform it into a structure `s'` such that `s'.repOk()`.

Figure 4 gives an overview of the repair algorithm, which performs a systematic search and uses symbolic execution. The class `Search` represents the backtracking engine; `PathCondition` represents integer constraints that arise during symbolic execution. The algorithm repeatedly invokes `repOk`. Each invocation results in a repair on `s`. The algorithm terminates when `s` is repaired, i.e., `s.repOk()` returns `true` and the corresponding path condition is satisfiable, or when the search is exhausted.

The `do-while` loop performs a systematic search by implicitly enabling *non-deterministic field assignments* within a standard Java Virtual Machine: the repeated invocations of `repOk` inside the loop body systematically tries different

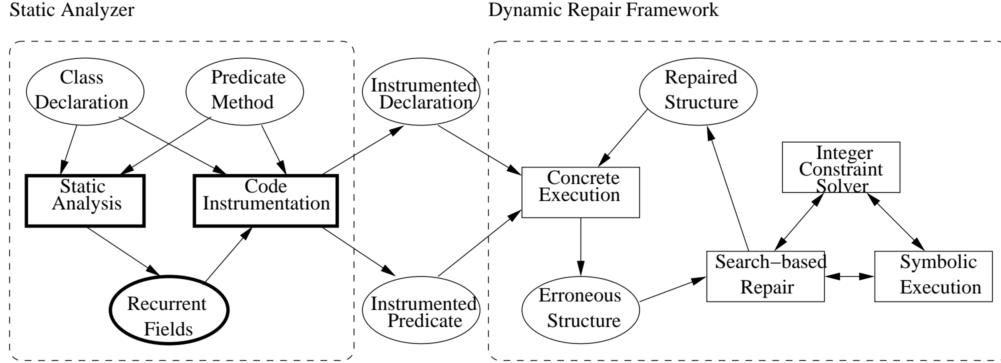

Figure 5. STARC framework for automatic data structure repair. The bold rectangles are the components that STARC adds on previous work on assertion-based repair. The static analysis guides the search of the dynamic repair framework.

values for fields that `repOk` accesses. The first invocation of `repOk` uses the original values for all fields. Each subsequent invocation modifies the value of the field that was accessed last by `repOk` according to the ordering of fields by their first access. If all values for the current field have been exhausted, the algorithm backtracks by resetting the original value for the current field, and modifying the value of the field accessed before the current one in the ordering.

Let `repOk` read field f of object o such that $o.f == v$ for some value v. There are two cases to consider: reference and primitive field accesses.

References. Let f be a reference field of type T. The algorithm non-deterministically assigns $o.f$:

- v, i.e., its current value;
- `null`, if $v \neq$ `null`;
- value w of a type T, such that $v \neq w$ and w has already been encountered during `repOk`'s invocation;
- a new value of type T, if v is not different from all values of type T already encountered during `repOk`'s invocation.

Primitives. Let f be a primitive field of type T. The algorithm non-deterministically assigns $o.f$:

- v, i.e., its current value;
- if $T ==$ `int`, a new symbolic value I, and adds the constraint $I \neq v$ to the current path condition;
- if $T \neq$ `int`, a primitive value w $(\neq v)$ non-deterministically chosen from $domain(t)$

Notice that a primitive field access may introduce symbolic integer values. For these values, invocation of `repOk` follows forward symbolic execution [29, 31], and satisfiability of path conditions is checked using CVC Lite [3]. Since reference as well as non-integer primitive fields initially have concrete values, the first execution of `repOk` follows standard Java semantics for these values.

4. STARC

This section describes STARC, an efficient engine for performing data structure repair. STARC incorporates both static and dynamic analysis to provide an efficient framework for automatic data structure repair. Figure 5 shows the repair framework. STARC has two main modules, a static analyzer and a dynamic search-based repair framework. STARC takes two inputs: (1) the class declaration of the desired structure, and (2) a predicate method `repOk` that describes the structural integrity constraints. The repair process is performed in three phases. First, STARC performs static analysis on the structure declaration and `repOk` to detect the recurrent fields of the structure (Section 4.1) and to extract constraints on references that can be solved statically (Section 4.2). Then, STARC uses the results of the static analyzer to instrument the structure declaration. Structure instrumentation includes replacing field access with invocations of ”get” methods, adding boolean variables to monitor the initialization of the fields, and inserting calls into the repair routines. Details are available elsewhere [29, 39]. Finally, STARC monitors the execution of the predicate and, if necessary, triggers the repair framework which uses (1) symbolic execution [31] to populate the constraints of primitive values, (2) systematic search to repair the faults in the structure [5], and (3) integer constraint solver to solve the constraints on primitives (Section 4.3).

We next describe each of the components of STARC in detail, and illustrate the performance advantage of STARC over Juzi with an example (Section 4.4).

4.1 Detecting recurrent fields of a structure

The performance of Juzi depends on the number of *repair actions*[1] that are required to repair a structure. To scale the performance of the search algorithm, STARC first implements a static analyzer that detects the recurrent fields of the data structure. A key observation behind finding the re-

[1] We use the term *repair action* to indicate a mutation that the algorithm tries when exploring the space for repairing a field

current fields is that such fields satisfy the reachability constraint of the structure. A recurrent field is more likely to point a new un-visited object rather than an already visited one. STARC uses this information to prioritize the repair actions, as well as prioritizing fields to repair when exploring the neighborhood of the faults.

Cahoon and McKinley [6, 7] proposed a data flow analysis framework for detecting the recurrent fields for prefetching of linked structures. We use this analysis to prioritize the repair actions. The problem is modeled as a forward data flow analysis problem. We first define some terms that we use to describe the components of the framework, then we describe the data-flow framework and illustrate how STARC uses the recurrent field information to prioritize repair actions.

4.1.1 Terminology

We start by defining the following terms:

- *Information unit (IU)*: An information unit is the left hand side of an assignment operation on objects or object fields. An information unit is used to save and propagate information in the data flow framework.
- *Object field (F)*: An object field is a reference field in the data structure.
- *Recurrent status (RS)*: The recurrent status of an object field can have one of three values: non_recurrent (nr), possibly_recurrent (pr), and recurrent (r), where the elements are ordered such that $nr \leq pr \leq r$.
- *Information site (IS)*: An information site is a program statement of interest.

For example, consider the `repOk` method for the `doubly LinkedList` class (Section 2.1). The information units are the local variables `visited`, `current`, `n`, and the implicit variable `this`. The object fields are the `header` field of the variable `this`, and the `next` and `prev` fields of the variables `current` and `n` respectively. The information sites are lines `L3, L5, L7, L12,` and `L0`, the entry of the method.

4.1.2 Data-flow framework

The basic data unit in the data flow framework is the *information tuple* T:

$$T \subset (IU \times F \times IS \times RS)$$

The data flow framework includes:

Initialization: When initializing an information tuple, all the components of the tuple are initialized to the bottom element of their lattices. For an information unit, iu, the bottom element is iu, for an object field, the bottom element is `null`, for an information site, the bottom element is $L0$, and for the recurrent status, the bottom element is *nr*.

The data-flow functions: The propagation of information in the framework occurs at the information sites. We consider two types of information patterns: equality patterns and access patterns. Given an input set of information tuples, *Rin*, we compute:

Equality patterns: Equality patterns take the form: ⟨*information unit*⟩ = ⟨*information unit*⟩′

$$\begin{array}{lcl} GEN(iu = iu', Rin) & = & \{(iu, f, is, rs) | (iu', f, is, rs) \in Rin\} \\ KILL(iu = iu', Rin) & = & \{(iu, f, is, rs)\} \end{array}$$

Access patterns: Access patterns take the form: ⟨*information unit*⟩ = ⟨*information unit*⟩′.⟨*object field*⟩

$$\begin{array}{lcl} GEN(iu = iu'.f, Rin) & = & \left\{ \begin{array}{c} \text{if } \{(iu', null, L0, nr)\} \in Rin \\ (iu, f, is, pr) \\ \text{if } \{(o, f, is, pr)\} \in Rin \\ (iu, f, is, r) \end{array} \right\} \\ KILL(iu = iu'.f, Rin) & = & \{(iu, f, is, pr), (iu, null, L0, nr)\} \end{array}$$

The meet operation: The meet operation ($\sqcup$) is defined on sets of tuples. Given two sets T1 and T2, the meet operation is defined as follows:

$$\begin{array}{lcl} T1 \sqcup T2 & = & \{t | t \in T1 \land t \notin T2\} \cup \{t | t \notin T1 \land t \in T2\} \\ & & \cup \{(iu, f, is, rs_1 \sqcup rs_2) | (iu, f, is, rs_1) \in T1 \\ & & \land (iu, f, is, rs_2) \in T2\} \end{array}$$

The transfer functions: The transfer functions are:

$$\begin{array}{lcl} A_{in}(ip) & = & \bigsqcup_{p \in pred(ip)} A_{out}(p) \\ A_{out}(ip) & = & (A_{in}(ip) / KILL(ip, A_{in}(ip))) \sqcup GEN(ip, A_{in}(ip)) \end{array}$$

Starting at the entry of the analyzed method (`repOk`), all the tuples are initialized. The algorithm proceeds by propagating information and iterating until a fixed point is reached. To illustrate, the first three iterations of the data flow analysis for the `DoublyLinkedList`'s `repOk` are displayed in Table 1. The framework converges in the fourth iteration.

At the end of the analysis, the field objects in the tuples that have a recurrent status r are considered the recurrent fields of the class as used by `repOk`. For example, as expected, in Table 1 all the tuples that have r as a recurrent status have `next` as an object field. Thus, `next` is the field used for traversing a list in the `repOk` method of the `DoublyLinkedList` class. The `prev` field is not reported by the analysis as a recurrent field since it is not used by `repOk` to traverse the structure.

Note that interprocedural analysis is performed similarly. At the call site, information is propagated to the entry of the called method by following a set of *equality patterns* for each argument in the method signature. At the return site, information is propagated from called method to the caller by following an equality pattern at the caller side.

4.1.3 Prioritizing repair actions

STARC uses the information about the reference fields to prioritize the candidates for repairing the structure fields. Recall that Juzi follows the same search pattern (Section 3.2)

stmt	RA	Iteration 1	Iteration 2	Iteration 3
L5	in	(current, null, L0, nr) (n, null, L0, nr)	(current, null, L0, nr) (n, null, L0, nr)	(current, null, L0, nr) (n, null, L0, nr)
L5	out	**(current, header, L5, pr)** (n, null, L0, nr)	(current, header, L5, pr) (n, null, L0, nr)	(current, header, L5, pr) (n, null, L0, nr)
L7	in	(current, header, L5, pr) (current, null, L0, nr) (n, null, L0, nr) **(n, next, L7, pr)**	(current, header, L5, pr) **(current, next, L12, pr)** (n, null, L0, nr) **(n, next, L7, r)**	(current, header, L5, pr) **(current, next, L12, r)** (n, null, L0, nr) (n, next, L7, r)
L7	out	(current, header, L5, pr) (current, null, L0, nr) **(n, next, L7, pr)**	(current, header, L5, pr) (current, next, L12, pr) **(n, next, L7, r)**	(current, header, L5, pr) (current, next, L12, r) (n, next, L7, r)
L9	in	(current, header, L5, pr) (current, null, L0, nr) (n, next, L7, pr)	(current, header, L5, pr) (current, next, L12, pr) (n, next, L7, r)	(current, header, L5, pr) (current, next, L12, r) (n, next, L7, r)
L9	out	**(current, next, L12, pr)** (n, next, L7, pr)	**(current, next, L12, r)** (n, next, L7, r)	(current, next, L12, r) (n, next, L7, r)

Table 1. The first three iterations of the data flow framework. The bold tuples indicate an update in the tuple information during successive iterations. The tuples for the information unit `this` and `visited` are never updated and thus they are omitted for brevity. The fields of the tuples that have a recurrent status r are recurrent fields of the structure.

when taking repair actions to fix an error in a reference field. The recurrent fields of a linked data structure are used to traverse the structure starting from a given root node. For traversing a structure, recurrent fields are more likely to point to new (non visited) nodes or `null` rather than pointing to previously visited nodes. STARC orders its repair candidates based on the type of the faulty field (recurrent or not). For recurrent fields, STARC gives higher priority for choosing a new (non-visited) candidate over choosing a visited one or `null`. For the non-recurrent fields, STARC chooses the same order presented in Section 3.2; STARC gives higher priority to choosing a visited node over a new node. This optimization not only improves performance (Section 4.4) but also guarantees that the reachability of the structure is preserved by repair.

4.2 Detecting constraints on references

Structural properties often constrain aliasing possibilities, e.g., $o.f == p \Leftrightarrow p.g == o$ for objects o and p, and fields f and g. Solving such constraints can be efficiently performed symbolically without enumerating the search space.

STARC implements a static constraint solver that repairs particular fields instantaneously; without triggering the search algorithm. Some of the imperative constraints on reference fields take the following pattern:

```
if (iu != iu') {
    ...
    return false; }
```

For example, the transpose relation between the `next` and `prev` fields of the `DoublyLinkedList` class takes the following form:

```
Node n = current.next;
if (n.prev != current)
    return false;
```

The solution of such constraints is embedded in the negation of the condition. STARC performs static analysis on the control flow graph of the `repOk` method to detect these patterns. Once these patterns are detected, the solver injects the solution of the constraint into the `repOk` method.

All the analysis that STARC performs is at the Java bytecode level. To detect patterns in a method, STARC builds the control flow graph (CFG) and searches for basic blocks where the entry instruction is a conditional branch and the exit instruction is an integer return. To detect the items being compared in the conditional statement, STARC uses the JVM specification [32] to trace the last two items produced on the stack. For example, consider the bytecode example of the transpose constraint of the `DoublyLinkedList` as described in `repOk` lines `L10` and `L11` in Section 2.1:

```
// compare prev to current
42: aload_3
43: getfield #32; //Field DoublyLinkedList\$Node.prev;
46: aload_2
47: if_acmpeq 52
// return false
50: iconst_0
51: ireturn
```

STARC detects the parameters of the conditional statement by following the consumer/producer chain of the previous instruction until two items are produced in the stack. In the above example, the instructions used to produce the comparison objects are:

```
42: aload_3       // consume: 0   produce: 1
43: getfield #32; // consume: 1   produce: 1
and
46: aload_2       // consume: 0   produce: 1
```

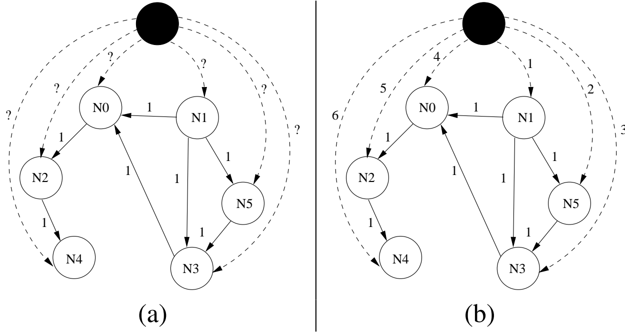

Figure 6. Data constraint graph for the `BinarySearch Tree` in Figure 3(e). The dotted lines are the edges from the newly added root. (a) Before solving the constraints, all the dotted edges are labeled with `'?'`. (b) Solution for the difference constraints; each `'?'` has been replaced with a value that satisfies the constraints.

These instructions are then used to produce the solution for the constraint and add the solution to the byte code as follows:

```
42: aload_3
43: getfield #32; //Field DoublyLinkedList\$Node.prev;
46: aload_2
47: if_acmpeq 57
// set the field of prev to current
51: aload_3
52: aload_2
53: putfield #32; //Field DoublyLinkedList\$Node.prev;
```

Using this solver, STARC identifies equality constraints and directly solves such constraints without using any non-deterministic search. This optimization enables highly efficient solving of a variety of *local* constraints. To illustrate, STARC automatically detects the transpose relation at lines `L10` and `L11` in Figure 1 of `DoublyLinkedList` and fixes any violation in the `prev` field by setting it to the transpose of its predecessor's `next` field.

4.3 Solving data constraints

Previous work on assertion based repair proposed different approaches for checking the satisfiability of the data constraints of a data structure. Demsky [10] proposed an approach similar to the one taken in Alloy [23] and the Alloy Analyzer. Their framework translates the constraints written in their language into a disjunctive normal form formula and solves the formula for satisfiability. Juzi uses symbolic execution on `repOk` to extracts the path condition that the data variables should satisfy and checks the satisfiability of the path condition using a theorem prover (CVClite [3]).

We have previously developed Dicos [14], a difference constraint solver for primitive integers. Dicos handled integer constraints that take the form $x < y$ and $x \leq y$. We extended the implementation of Dicos to handle equality constraints of the form $x \neq y$ and $x == y$. Following a textbook algorithm [9], the current implementation builds a constraint graph where the vertices are the primitive fields, and the edges are the constraints. Dicos adds a *root* node in the graph that is a predecessor of all the nodes. Once the graph is built, the problem simplifies to finding the single source shortest path from the added *root* node. To check the satisfiability of the constraints, Dicos checks for negative cycles in the graph. A negative cycle indicates a contradiction in the constraints. Dicos implements the Bellman-Ford [15] algorithm to find the shortest path in time $O(v.e)$. Since the complexity of the data constraints varies between structures, Dicos uses faster algorithms for handling simple constraints. For example, the data integrity constraints of the binary search tree example (Section 2.2) are translated into a directed acyclic graph (DAG) rather than a cyclic one. For a directed acyclic graph with v nodes and e edges, Dicos can compute the primitive values in $O(v + e)$ using a topological traversal. To illustrate, Figure 6 shows the data constraint graph for the path condition in Figure 3(e). The topological distance from the added root node to each node determines the order of the data and solves the path condition. Dicos keeps track of the nature of the graph being constructed and then decides on which algorithm to use. Dicos even performs some simplifications on the path condition that might solve satisfiability without the need of a solver. These simplifications include transforming constraints in a path condition to a canonical form, performing subsumption checking for simple cases, and propagating constants.

Dicos also handles more complex constraints like linear programs, and quadratic programs, yet using this capability requires user intervention to pre-select the desired algorithm based on the target problem. Note that Dicos still uses CV-Clite to check satisfiability if the constraints do not fit a category that it can handle.

4.4 Illustration

We next illustrate the performance gain that STARC achieves due to the aforementioned optimizations. We compare the performance of Juzi and STARC in repairing faults in the doubly linked list example from Section 2.1. We first consider the example of the erroneous list in Figure 7(a). All the `prev` fields in the list point to the header node. We repair this list using Juzi and STARC. Juzi made 25 repair actions and required 15ms to repair the structure whereas STARC took 5 repair actions in less than 1ms. We then consider the erroneous list in Figure 7(b). In this list, the faults occur in the `next` field of node `N3` and in the `prev` field of nodes `N0` and `N2`. For repairing this list Juzi took 47 repair actions in 37ms whereas STARC took three repair actions again in less than 1ms. This improvement in performance is due to the optimizations applied in STARC which direct the repair algorithm to select the most-likely option first. This reduces the number of attempts needed to repair a field.

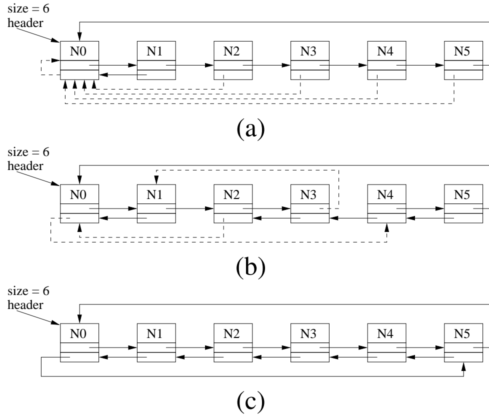

Figure 7. (a) An erroneous list with all the `prev` fields pointing to the header node. (b) An erroneous list with a fault in the `next` field of the fourth node and the `prev` field of the third node. (c) The resulting list after applying our repair algorithm; all the constraints are satisfied.

5. Evaluation

This section evaluates the efficiency of STARC in repairing large data structures. First, we present the methodology for evaluating STARC. We then use STARC to repair a set of standalone subject data structures. Finally, we demonstrate STARC on an application that implements a software cache.

5.1 Methodology

We evaluate STARC by applying it to faulty implementations of ten subject structures. For each subject, we evaluate the time it takes to repair a faulty structure for sizes: 100, 1,000, 10,000, and 100,000. We repeat the repair procedure using 50 different randomization seeds and report the average repair time. We also compare the number of repair actions required to fix the errors. This metric measures the growth of the search algorithm and the efficiency of the added optimizations. We consider one example in detail to illustrate how STARC generates structures that can automatically repair themselves when an error occurs.

We compare the results of STARC with those of Juzi. We set a threshold time of 10 minutes to repair a faulty structure, and stop the execution after that period.

To study the efficiency of both the static analyzer and the constraint solver, we consider subjects that vary in the types and complexity of their constraints. We first consider structures with constraints on the structure only, that do not require the integer constraint solver. We then consider structures with constraints on both the structure and data that require the use of the integer constraint solver to determine the satisfiability of the constraints on primitive data. In order to clearly expose the advantage of the static analyzer, we consider structures with complex structural and data constraint and use our constraint solver in both Juzi as well as in STARC. Any performance advantage is thus due to the improvement in the performance of the search algorithm.

We next describe the data structure subjects and the repair results. All experiments used a 1.7 GHZ Pentium M processor with 1 GB of RAM.

5.2 Subjects

Table 2 lists the subject structures and the integrity constraints. The binary search tree and the doubly linked list are both presented in Section 2. Disjoint set is a linked-based implementation of the fast union-find data structure [9]; this implementation uses both path compression and rank estimation heuristics to improve efficiency. Fibonacci heap is a dynamic data structure that implements a heap, but differ from a binary heap in complexity for certain operations [9]. Singly linked list is the simplest structure with constraints only on the acyclicity of the structure. Sorted list is structurally identical to a singly linked list but the elements are sorted. Red-black and AVL tree implement a balanced search tree, with red-black trees having complex constraints on the colors of the nodes along the paths from the root [9]. N-ary trees are used in the implementations of XML documents and file systems. Software cache is a more complex structure that comprises both a hash table and a doubly linked list.

For each of the subjects, we constructed a set of faulty structures by generating valid structures using available API's calls, and randomly traversing and mutating their reference and value fields.

5.3 Results

Table 3 displays the time and the number of repair actions taken by Juzi and STARC to repair the described subject structures. Singly linked list has the simplest of the constraints and the least number of faults and its repair is therefore the fastest. The n-ary tree data structure is similar in complexity to the singly linked list, yet the number of faults is larger. Note that the performance of Juzi and STARC is indistinguishable for simple constraints like acyclicity. Breaking cycles is achieved by setting the value of the corrupt field to `null`. The performance of the integer constraint solver is tested in the sorted list and the binary search tree examples. Recall that Juzi uses a theorem prover (CVClite [3]) to check for the satisfiability of the path condition. The results show that using a dedicated integer constraint solver provides up to one order of magnitude improvement for STARC especially in large structures. For the rest of the subjects, we use the same constraint solver for Juzi and STARC in order to study the improvement due to the static analysis.

The doubly linked list, and the disjoint set data structure are structurally more complex than the singly list and the n-ary trees. Results show that for solving constraints like transpose, sentinel (all pointers point to a sentinel node) and reachabilitiy, STARC outperforms Juzi by more than two orders of magnitude. Juzi did not finish the execution within

subject	structural constraints	data constraints
Singly linked list	acyclicity, reachability	N/A
Sorted linked list	acyclicity, reachability	sorted elements
n-ary tree	acyclicity, reachability, one parent	N/A
Binary Search tree	acyclicity, reachability, one parent	natural order on elements
Doubly linked list	reachability, transpose, circularity	N/A
Disjoint sets	acyclicity, reachability, sentinel	disjoint members
AVL tree	acyclicity, reachability, one parent, balance	natural order on elements
Red-black tree	acyclicity, reachability, one parent, transpose, path, color	natural order on elements
Fibonacci heap	acyclicity, reachability, transpose, circularity	min, heap property
Software cache	reachability, transpose, circularity, hash	correct binning of items

Table 2. Subject structures and their integrity constraints. Subjects vary in the complexity of their structural and data constraints. We generated a valid set of structures for each subject, and randomly injected faults that violate the integrity constraints.

the given time when repairing a doubly linked list with 1,000 nodes and 100 faults, whereas STARC was able to repair a doubly linked list with 100,000 nodes in less than a minute.

Note that although the algorithm is complete for both Juzi and STARC, the former took 21,656 repair actions to repair 10 faults in a doubly linked list of size 1,000 whereas the latter only took 9 actions. The static analyzer in STARC bias the repair algorithm toward solving the reachability constraint while repairing a recurrent field. Juzi on the other hand repairs the faults in the structure, yet the repaired structure might not satisfy the reachability constraint, thus it keeps searching for a structure that satisfies all the constraints. This explains the large performance gain of STARC over Juzi.

We test the efficiency of STARC on repairing more complex constraints like balance (AVL), color and path (red-black). Again for these structures, STARC is able to repair structures with one hundred thousand nodes within the given time.

Recall that the performance of STARC is directly proportional to the number of repair actions taken while repairing a structure. In our experiments, the number of repair actions grows linearly with the number of corruptions in the structure. We plot the repair time versus the number of faults in the structure (Figure 8) for a doubly linked list with ten thousand nodes. The repair time grows essentially linearly with the number of faults in the structure. We plot the repair time versus the size of the structure (Figure 9) for a doubly linked list with 10 corruptions. The repair time grows quadratically with the size of the structure for a fixed number of corruptions. This result is justified as follows: the static analysis in STARC direct the search algorithm to the most likely value to repair a fault, and thus most of the faults are repaired from the first attempt. The backtracking algorithm that STARC uses is stateless. Each execution of `repOk` re-initializes the state of the structure. Thus, the structure is constructed with every repair action, which adds a quadratic effect on the runtime of STARC.

An alternative approach is to implement a stateful search, which allows real backtracking similar to that in the Java PathFinder (JPF) model checker [40] and obviates the need of repeated invocations of `repOk` from the beginning. We applied both approaches to repair, and preliminary experimental results showed that due to the high overhead of saving the state, a stateful approach, as in JPF, is less efficient when repairing small structure (less than five thousand nodes). However, as the size of the structure increases, JPF outperforms stateless backtracking. The results open a direction for future work. We believe that, since `repOk` is a pure function, i.e., does not change the state of the structure, using an incremental approach, i.e., saving only sections of the state that are of interest for repair, reduces JPF's overhead and leads to more efficient repair.

5.4 Overhead

In this section, we study the runtime overhead of STARC. Overhead occurs in two forms. (1) Delays due to the extra method calls performed when running the instrumented code. This overhead is minimal since it only includes calling simple accessor methods rather than direct field accesses. (2) Runtime overhead that arises from calling `repOk` to check the validity of the structure. Each call to `repOk` performs a linear traversal of all the fields in the structure when the structure is valid.

Similar to other techniques on automatic data structure repair [11, 10, 12, 28], our repair algorithm does not state when to check for the validity of the structure, but it leaves it to the user to decide when to do so. In this section, we study the overhead due to a *conservative checking* which calls `repOk` at the boundaries of the public methods that modify the structure, and an *optimistic checking* which only checks for the validity of the structure when an exception is thrown in the program. To study the runtime overhead,

Subject Structure	*Size*	*# of faults*	*Time(ms)*		*# of repair actions*	
			Juzi	*STARC*	*Juzi*	*STARC*
Singly linked list	1,000	1	43	41	1	1
	10,000	1	138	118	1	1
	100,000	1	1,698	1,581	1	1
Sorted list	1,000	1	6178	53	1	1
	10,000	1	13,142	1,411	1	1
	100,000	1	89,656	8,733	1	1
Doubly linked list	100	10	516	31	1,525	9
	1,000	10	34,046	63	21,656	9
		100	$\geq \tau$	234	$\geq \delta$	99
	10,000	10	$\geq \tau$	656	$\geq \delta$	9
		100	$\geq \tau$	4,078	$\geq \delta$	99
	100,000	10	$\geq \tau$	4,594	$\geq \delta$	9
		100	$\geq \tau$	53,828	$\geq \delta$	99
N-ary tree	100	10	17	16	11	10
	1,000	10	96	88	11	10
		100	374	328	101	100
	10,000	10	715	656	11	10
		100	3,781	3,672	101	100
	100,000	10	7,268	6,656	11	10
		100	57,468	55,327	101	100
Binary search tree	100	10	128	24	11	10
	1,000	10	7,424	137	11	10
		100	8,173	428	101	100
	10,000	10	182,817	5,211	11	10
		100	210,577	10,755	101	100
Disjoint set	100	10	2,781	31	2,376	12
	1,000	10	45,103	63	32,941	12
		100	$\geq \tau$	438	$\geq \delta$	112
	10,000	10	$\geq \tau$	734	$\geq \delta$	12
		100	$\geq \tau$	4,672	$\geq \delta$	112
	100,000	10	$\geq \tau$	5,149	$\geq \delta$	12
		100	$\geq \tau$	61,751	$\geq \delta$	112
AVL tree	100	10	1,006	78	1,675	11
	1,000	10	12,521	738	9,648	11
		100	$\geq \tau$	1,597	$\geq \delta$	101
	10,000	10	$\geq \tau$	4,816	$\geq \delta$	11
		100	$\geq \tau$	9,483	$\geq \delta$	101
	100,000	10	$\geq \tau$	7,469	$\geq \delta$	11
		100	$\geq \tau$	58,422	$\geq \delta$	101
Red-Black tree	100	10	3,107	169	1,811	29
	1,000	10	16,437	1,382	24,399	49
		100	$\geq \tau$	2,810	$\geq \delta$	139
	10,000	10	$\geq \tau$	6,942	$\geq \delta$	109
		100	$\geq \tau$	14,385	$\geq \delta$	299
	100,000	10	$\geq \tau$	10,191	$\geq \delta$	153
		100	$\geq \tau$	117,491	$\geq \delta$	244
Fibonacci heap	100	10	799	26	1,833	19
	1,000	10	17,903	622	10,899	24
		100	$\geq \tau$	6,953	$\geq \delta$	99
	10,000	10	$\geq \tau$	18,615	$\geq \delta$	9
		100	$\geq \tau$	45,703	$\geq \delta$	99
	100,000	10	$\geq \tau$	37,197	$\geq \delta$	9
		100	$\geq \tau$	117,366	$\geq \delta$	99

Table 3. Results for repairing large structures with up to 100,000 nodes. The tabulated times are in milliseconds. τ represents a time threshold of 10 minutes. δ represents a threshold one million repair actions. STARC is able to repair structures with 100 times more faults than Juzi.

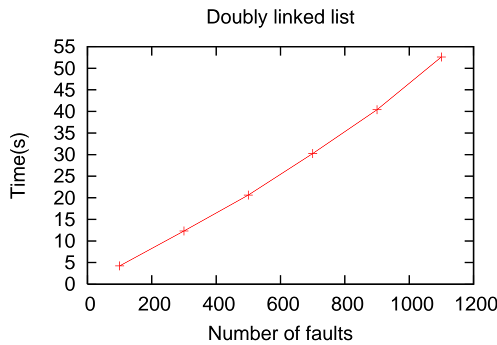

Figure 8. Variation in the repair time of a doubly linked list with 10000 nodes with the number of corruptions.

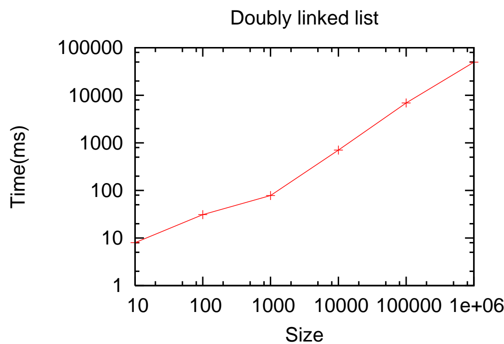

Figure 9. Variation in the repair time of a doubly linked list with 10 corruptions with the size of the structure.

we created a simple program that, for each of the subjects, performs the following:

1. builds a structure with 5,000 nodes
2. injects up to 5 faults in the structure
3. adds another 5,000 nodes
4. traverses the structure

We first measure the overhead when there are no faults in the structures. We run the original code, and the instrumented code when the structure has no faults, i.e., without performing *step 2*, and measure the overhead due to calling `repOk` in the optimistic and the conservative checking methods. Table 4 shows the runtime in milliseconds for the experiment.

Since the structures have no faults, no exceptions are thrown, and thus, the runtime of the original code and the instrumented code when performing an optimistic check is very similar. Conservative checking adds one order of magnitude for complex structures, such as AVL trees, and almost two orders of magnitude for simple structures, such as a singly linked lists. This overhead is expected as the program traverses the structure every time a node is added.

We next measure the overhead when there are faults in the structure, i.e., with *step 2*. Table 5 shows the results for the original code, the optimistic checking, and the conservative checking.

The original code crashes while adding nodes after the faults are injected for some structures, and while traversing the nodes for other structures. For optimistic checking, the structure is repaired whenever an exception is thrown during addition or traversal, and the program safely terminates. For the conservative checking, the structures are repaired on the first node addition after the fault injection process since `repOk` is checked with every addition.

As expected, conservative checking takes more time than optimistic checking in almost all the subject structures. However, for the optimistic checking, the structure implementation should throw exceptions when performing the traverse or the add operations. If no exceptions are thrown, the program might end up crashing, say because of an infinite loop. For the conservative checking this problem might

subject	original code	optimistic checking	conservative checking
Singly linked list	36	47	3,314
Sorted linked list	250	301	3,847
n-ary tree	206	241	5,203
Binary Search tree	351	388	5,612
Doubly linked list	42	56	3,609
Disjoint sets	148	162	4,601
AVL tree	409	472	6,215
Red-black tree	573	627	7,243
Fibonacci heap	366	402	6,682

Table 4. Overhead imposed by the repair framework when running an application which manipulates structures with up to 10,000 nodes. Column 2 shows the runtime for the original code. Column 3 shows the runtime for the instrumented code when performing optimistic checking. Column 4 shows the runtime for the instrumented code when performing conservative checking. All the times are in milliseconds.

subject	original code	optimistic checking	conservative checking
Singly linked list	26	118	3,550
Sorted linked list	150	634	4,188
n-ary tree	116	470	5,597
Binary Search tree	224	2,622	6,271
Doubly linked list	39	462	3,916
Disjoint sets	87	566	4,809
AVL tree	294	3,814	7,083
Red-black tree	503	5,048	9,730
Fibonacci heap	263	10,411	10,169

Table 5. Results for performing optimistic versus conservative checking when repairing a corrupt structure.

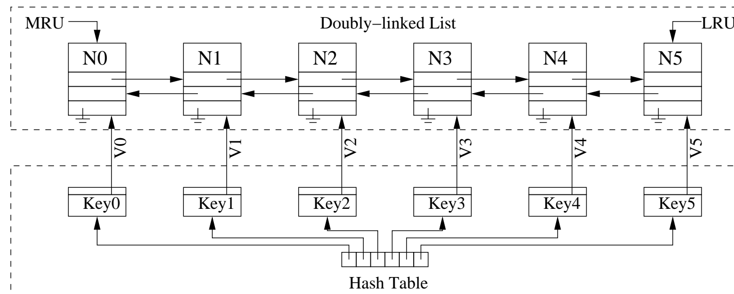

Figure 10. An erroneous `LRUCache` created by inserting 6 different nodes using the faulty `add` method. The `key` pointers of the cache entries are set to `null` rather than to their corresponding key in the hash table.

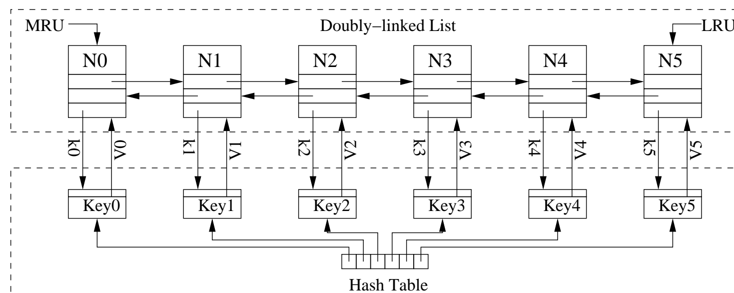

Figure 11. An `LRUCache` example with capacity 6. Each entry in the cache is part of both a hash table and a doubly linked list. The entry is an instance of the `Data` class. It has two pointers to the `next` and `prev` elements in the list, and a `key` pointer to the corresponding entry in the hash table. The doubly linked list reflects the order of access from most recently accessed to least recently accessed.

not be possible, yet the overhead of checking `repOk` with every node addition is high.

The decision of when to check the validity of the structure depends on the type of the structure, and the performance and reliability requirements. It is up to the user to decide when to check for validity. Section 6.7 describes more ways to check for the validity and to trigger repair.

5.5 Case-study: repairing a software cache

In this section, we illustrate how STARC creates a repairable application that can automatically recover from a dynamic error when the application is already deployed. We use STARC to create a repairable software cache, a small, but interesting application with a complex data structure. Software caches are gaining popularity in service oriented applications (SOA) [4] such as online business applications. These applications are expected to be online any time the user requests a transaction. The cost of failure in such applications is prohibitively high. We first describe a faulty implementation for a software cache that implements the least recently used (LRU) [38] replacement algorithm, and then describe how the cache repairs without halting.

Software Cache: A software cache is a data structure that supports constant time addition, access, and removal. To support these requirements, it implements a hash table. The capacity of a cache is limited. Once the cache is full, elements are replaced. The cache described in this section implements the least recently used algorithm. This algorithm requires the cache to keep track of the least and the most recently accessed elements. Thus, in addition to the hash table a software cache implements a doubly linked list to keep track of the order in which data is accessed. When replacing an element, the LRU algorithm removes the entry from both the linked list and the hash table. Thus, each entry in the list should have a pointer to its key in the hash table as shown in Figure 11.

Faulty Implementation: We illustrate how STARC repairs a faulty cache and corrects the program behavior on-the-fly. The class `LRUCache` declares the subject structure:

```
class LRUCache {
    int capacity;
    int size;
    HashTable cache;
    DoublyLinkedList lru;

    static class Data {
        Object value;
        Object key;
        Data next;
        Data prev;
    }
}
```

Each cache has a `capacity` field which represents the maximum number of nodes that the cache can hold, a `size` field to represent the current number of nodes, a hash table to support constant time data access, and a doubly linked list to keep track of the order in which the data is accessed. The inner class `Data` models the entries in the cache. Since each element in the cache is a node in the doubly linked list, each

entry has two fields indicating the `next` and `prev` entries in their recent access order in the cache, a `value` field to represent the saved data, and a `key` field which saves the corresponding key entry in the hash table.

Consider the method `add` that given a key and a value performs the following:

- adds an entry to the cache if the key is not already in the cache and if the cache is not full,
- updates an entry in the cache if the key is already in the cache and tags it as the most recently accessed node, and
- replaces an element in the cache if the key is not found in the cache and the cache is full.

The following code gives an implementation of the `add` method:

```
void faultyAdd(Object key , Object value) {
    // get the data entry if its in the cache
    Data oldData = (Data)cache.remove(key);
    // remove it from the list if it is in the list
    if (oldData !=  null) {
        lru.remove(oldData);
    } else {
        // if the cache is not full, increment the size
        if (size < capacity)
            ++size;
        // if the cache is full, remove the LRU node
        else if (size == capacity) {
            Data removed = lru.removeLast();
            cache.remove(removed.key);
        }
    }
    // add the node in the top of the list
    Data add  = lru.addFirst(key,value);
    // add the entry to the cache
    cache.put(key, add);
}
```

The state of the cache after six additions using the above method is displayed in Figure 10. All the `key` pointers in the `Data` objects are set to `null`. The bug is in the implementation of the `addFirst` method of the doubly linked list. The `key` field of the `Data` class is set to `null` rather than the key entry in the hash table. The program reports an unhandled null pointer exception when adding an additional node in to the cache (trying to replace the LRU entry from the cache).

We instrument the `LRUCache` and the `Data` classes using STARC and run the `add` method again. The repair routine is triggered whenever the null pointer exception is thrown, and restores the cache to a valid state. We randomly add 1,000 entries to the cache and the code reports 89 calls to the repair routine in less than two seconds without any crash in the program.

6. Discussion

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

6.1 Constraint generation

Our repair algorithm expects the user to provide the integrity constraints by writing the `repOk` method. For complex constraints, writing a precise predicate can be error-prone. Existing constraint-synthesis tools can be used to help users formulate the predicates correctly. Daikon [41] (a tool for invariant discovery) stores a dictionary of invariants and uses a learning algorithm to discover what invariants apply to the structure. Cork [26] builds on the garbage collector to take snapshots of the structure being collected. Using these heap snapshots, Cork can learn some properties of the structure We have recently developed Deryaft [34], a tool that specializes in generating constraints of complex data structures. Deryaft takes as input a handful of concrete data structures of small sizes and generates a `repOk` predicate that represents their structural integrity constraints. For example, for all our benchmarks except red-black trees and disjoint set, Deryaft can generate the precise `repOk`'s using five sample structures for each subject. Even in cases when Deryaft is unable to output a complete `repOk` predicate, Deryaft's output helps the users correctly formulate the predicate, say by using the output as a skeletal implementation.

6.2 Completeness of the Repair Algorithm

The Juzi algorithm is based on Korat [5], a systematic search based test input generator. Juzi explores all the non-isomorphic structures that satisfy the integrity constraints. Thus, if there exists a structure that satisfies the integrity constraints, Juzi will find it. We point out that the optimizations added in STARC do not affect the completeness of the original algorithm. Using the recurrent field information, STARC only changes the order of the search and does not skip any valid structure from being explored. The reference constraint solver statically detects and solves constraints that are not yet initialized by the search algorithm. Thus, instrumentation of `repOk` does not affect its behavior.

The algorithm is complete for integer constraints. Once a structure is repaired, the problem of solving the data constraints is decidable. If the path condition is satisfiable then the solver generates a solution to it. It is important to point out that STARC relies on the user to provide correct and satisfiable constraints in `repOk`. If the constraints are not satisfiable, STARC notifies the user after exploring all the search space.

6.3 Reachability of the Repaired Structure

An important characteristic of STARC is that it solves two problems: the structural constraints as described in `repOK` and the reachability of the original structure nodes. Recall in Section 4.1.3 that STARC prioritizes the order of choices according to the type of the faulty field. Using the recurrent analysis information, the recurrent fields are assigned to new non-visited nodes. Thus, STARC first solves the reachability problem, and then satisfies `repOk`. This feature is not present in previous work on assertion based repair, which usually finds the first structure that solves the constraint disregarding the original size and the number of nodes reachable from the root of the structure. For example, a faulty doubly linked list of 100 nodes might be repaired into a list with 10 nodes that satisfies the structural integrity constraints.

Since STARC algorithm is complete, and prioritizes reachability, it will first try to find a solution with all reachable nodes. If none exists, it will satisfy `repOk` with a smaller structure, if possible.

6.4 Generation Based Repair

STARC can easily be used for test input generation [14]. A key observation behind this idea is that while the problem of generating an input that satisfies all the given constraints is hard, generating a structure at random, which may or may not satisfy the constraints but has a desired number of objects is straightforward. Indeed, a structure generated at random is highly unlikely to satisfy any of the desired constraints. However, it can be *repaired* using STARC to transform it so that it to satisfy all the desired constraints.

6.5 Data Repair

Data repair is one of the most challenging problems in 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.

Our approach gives the user some control on how to repair the data. (1) The user can provide in a configuration file, the list of primitive fields that are not to be instrumented, and thus, never changed by the repair algorithm. (2) The user can specify a ranges of data values for primitive fields and use these ranges to constrain the repair algorithm. (3) The user can state specific relations between the values of a corrupted structure and a repaired structure akin to specifying postconditions that relate pre-state with post-state. For example, if the user specifies an order relation between the elements of the structure, say a binary search tree or a sorted list, the repair algorithm will reorder the values in the structure using the order of the values generated by the constraint solver.

6.6 Controlling the Fields to Repair

STARC 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, STARC cannot modify the size field to satisfy other constraints, and if the reachability property is not satisfied, STARC reports the structure as non-repairable. This feature has both its advantages and drawbacks. On one hand, it gives flexibility to the user to specify some fields as concrete all the time of execution, but on the other hand these field will not be repairable and if a fault occurs in these fields, STARC cannot repair it.

6.7 When to Trigger the Repair Routine?

One open question in data structure repair is when to trigger the repair routine. STARC provides the framework for repair, and lets the user decide when to use the repair routines. It is unknown when an error is going to occur in the program, which makes triggering the repair routine a tricky decision, as we do not want to affect the performance of the running applications by continuously checking the validity of the constraints. In Section 5.5, we provided an example where the repair routine is triggered when an exception is thrown due to an error in the implementation of the structure's API. This technique was efficient when repairing the software cache yet in some cases it is unfeasible to wait for an exception to be thrown to perform the repair. For example, I/O operations that are performed on the faulty structure are persistent, and it is very expensive to invalidate such operations. Another approach is to give the Java Virtual Machine (JVM) the control over when to trigger repair, and run the integrity checks periodically analogous to garbage collection. The efficiency of this approach is yet to be studied.

7. Related Work

This section compares STARC to prior work on error recovery, dedicated repair routines, and constraint-based repair.

Error recovery has been part of software systems for a couple of decades [25, 37]. System reboot is a traditional error recovery mechanisms. In this approach, the user reboots the system when it crashes, uses system logs to analyze the cause of the problem, and creates patches to fix the errors. One disadvantage of this approach is that the system state before the crash is lost and the system returns to its initial state. Check-pointing [30] tackles the problem of state loss when rebooting by recovering the program state to the last saved state rather than the initial one [27]. One drawback still exists when persistent, rather than volatile, faults occur in a system. In this case, it is very difficult to automate recovery using traditional approaches.

Another technique for error recovery is to implement dedicated repair routines that are triggered when specific problems occur during a system's execution. For example, file system utilities, such as `fsck` and `chkdsk`, routinely check and correct the underlying file structure. Some commercially developed systems, such as the IBM MVS operating system [36] and the Lucent 5ESS telephone switch [20], provide routines for monitoring and maintaining properties of their data structures. These routines, however, do not perform repair using a description of the data structure constraints, and thus it is hard to build a robust generic repair framework using such approaches, since the developer must envision all possible bugs.

The use of structural integrity constraints as a basis to perform repair is relatively new. Demsky and Rinard [10, 12] are the first to use constraints as repair routines. Their framework performs repairs based on constraints written in a new

declarative language that is similar to the first-order relational language Alloy [24]. Repair is performed by translating the constraints to disjunctive normal form and solving them using an ad hoc search. To help the user formulate constraints correctly, they have taken a promising new approach [11] of integrating repair with dynamic invariant generation using Daikon [16].

Our work differs from theirs as our algorithm allows writing constraints using the language of implementation, while they require using a declarative language. Since this language does not support transitive closure [10], it is very hard to model the complex structural constraints that STARC solves and which are easily expressed in Java.

Previous work presented Juzi [13, 17, 28, 39], a constraint-based repair framework, which uses constraints written as Java predicates. In contrast with Demsky and Rinard's approach where the search is adhoc, Juzi uses a systematic search, which is based on symbolic execution [29, 31] and Korat [5], an efficient tool for constraint-based generation of data structures. To decide the feasibility of path conditions that arise during symbolic execution, Juzi uses the CVC-lite theorem prover [3].

STARC builds on Juzi and introduces a novel static analysis that enables efficient and effective repair. Additionally, STARC implements a dedicated constraint solver, which provides significantly faster constraint solving (for a range of integer constraints) than the automated theorem prover used by Juzi. Experimental results show that STARC can repair structures that are up to 100 times larger in size than those feasibly handled by Juzi.

8. Conclusions

This paper introduced the idea of using static analysis to guide the search and scale the performance of assertion-based repair. It presented STARC, an efficient framework for repairing large data structures. STARC builds on previous work on error recovery. Given a Java predicate that represents the desired structural and data integrity constraints, and a faulty structure, STARC first performs static analysis on the structure to detect the recurrent fields, then performs systematic search of the neighborhood of the faults to find candidates that repair the structure to satisfy the given constraints.

Experiments on repairing data structures using subjects with complex structural and data constraints show that STARC can efficiently repair structures with up to 10,000 nodes. In comparison with previous work on assertion-based repair, STARC feasibly repairs structures that are up to 100 times larger.

We believe that using static analysis is a highly promising approach to improve the performance of constraint-based approaches for error recovery as well as automated testing.

Acknowledgments

We would like to thank Christine Kehyayan, Feras Karablieh, Darko Marinov, and the anonymous referees for their helpful comments on the paper. This work is supported by the EDGE scholar program, NSF ITR-0438967, NSF CCF-0429859, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, Intel, IBM, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

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] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 2005.

[3] 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.

[4] Douglas Barry. *Web Services and Service-Oriented Architectures: The Savvy Manager's Guide.*, chapter Service Oriented Architecture. Morgan Kaufmann Publishers, 2003.

[5] 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.

[6] B. Cahoon and K. McKinley. Recurrence analysis for effective array prefetching in java. *Concurrency and Computation Practice and Experience*, 17, February 2005.

[7] Brendon Cahoon. *Effective Compile-Time Analysis for Data Prefetching in Java*. PhD thesis, University of Massachusetts, Boston, MA, 2002.

[8] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

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

[11] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2006.

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

[13] Bassem Elkarablieh, Iván García, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of structurally complex data. (Under submission).

[14] 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.

[15] Sanguthevar Rajasekaran Ellis Horowitz. *Computer Algorithms*. W. H. Freeman, second edition, 1997.

[16] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[17] Iván García. Enabling symbolic execution of Java programs using bytecode instrumentation. Master's thesis, The University of Texas at Austin, May 2005.

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

[19] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[20] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2), 1985.

[21] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.

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

[23] Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001.

[24] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[25] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV '05)*, 2005.

[26] Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for java. In *Proc. 34th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, 2007.

[27] Feras Karablieh, Rida A. Bazzi, and Margaret Hicks. Compiler-assisted heterogeneous checkpointing. In *Symposium on Reliable Distributed Systems (SRDS)*, October 2001.

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

[29] 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.

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

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

[32] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

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

[34] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid. Generating representation invariants of structurally complex data. In *Proc. 11th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

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

[36] Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, 13(10), 1987.

[37] 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.

[38] William Stallings. *Computer Organization and Architecture.*, chapter Cache Memory. Prentice-Hall, Englewood Cliffs, NJ, 2006.

[39] 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.

[40] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[41] Tao Xie and David Notkin. Tool-assisted unit test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.