Enhancing Constraint-based Repair of Data Structure Errors that Recur using Memoization

Nima Dini
University of Texas at Austin, USA
nima.dini@utexas.edu

Razieh Nokhbeh Zaeem
University of Texas at Austin, USA
nokhbeh@utexas.edu

Sarfraz Khurshid
University of Texas at Austin, USA
khurshid@utexas.edu

ABSTRACT

Data structure repair has been proposed as an error recovery mechanism to increase software resilience when errors happen at runtime for a deployed system. Although substantial work has been done on data structure repair, scalability remains a key challenge and applicability remains rather restricted.

We present two constraint-based data structure repair techniques that build on the well-known Korat solver, which introduced an effective backtracking search to find all object graphs that satisfy user given structural properties within the user given bounds. Our baseline technique repairs an erroneous data structure by adapting the Korat search. While this approach is effective for repairing a data structure, it suffers from re-exploration when applied on data structures with similar errors that recur due to a fault in software or hardware that is exercised repeatedly. We introduce memoization to amortize the cost of a Korat search over repeated repairs that overlap in search. Our experimental results show that our memoized technique is effective for repairing data structure errors that recur.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; **Error handling and recovery**;

KEYWORDS

Data structure repair, Error recovery, Korat

ACM Reference Format:
Nima Dini, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2021. Enhancing Constraint-based Repair of Data Structure Errors that Recur using Memoization. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3442055

1 INTRODUCTION

As software systems become increasingly more complex, reliability becomes harder to achieve. Software testing is a common approach to detect faults early on in software development, yet it cannot eliminate all the bugs. Failures may still occur at runtime, e.g., due to a corrupt data caused by a transmission error.

Data structure repair [2, 3, 8, 9] is an approach for error recovery to make software resilient to runtime failures caused by data corruption. The goal of data structure repair is not to bring an erroneous data structure into an ideal state, which a correct execution would have generated, but to mutate it to a data structure that is acceptable for continuing the program execution [28].

Admittedly, data structure repair is not for every kind of software and every type of error. In some instances, a programmer would rather let the program execution fail. Data structure repair, however, remains a valuable tool for particular types of software (e.g., mission critical systems) and errors (e.g., transmission errors).

The focus of this paper is on reducing the cost of *constraint-based* data structure repair when similar data corruptions recur at runtime due to a fault in software or hardware that is exercised repeatedly. Our repair approach follows the spirit of prior work [2, 14, 17, 29, 33], where *erroneous* states are transformed into *acceptable* [28] states based on user given structural properties, so the program's execution continues instead of being terminated with failures.

Our work is based on the Korat solver [1, 23], which takes a user given Java class with `repOK` and `finitization` methods [21]. The `repOK` is a boolean method that implements the structural properties. The `finitization` specifies bounds on the number of objects for each type, and a *field domain* (a finite list of values) for each field of each type.

Korat internally represents each candidate structure as an integer array, namely a *candidate vector*. Korat starts the search from an initial candidate vector, where each element is set to the first value in its field domain, and exhaustively enumerates all valid structures s within the user given bounds, such that `s.repOK()` returns `true`.

The constraint-based data structure repair problem is defined as follows [8, 17]: given a `repOK` method and erroneous data structure e, such that `!e.repOK()`, mutate e into valid data structure r such that `r.repOK()` and r is *similar* to e. Similarity is a heuristic notion, intended to restrict repair, to avoid unnecessary mutations and to preserve as much of the original data structure as possible.

This paper makes the following contributions:

Data structure repair using Korat. We present our baseline data structure repair technique, which takes Java class `C` (with a user given `repOK`) and erroneous data structure e (of type `C`) as inputs, and adapts the Korat search to efficiently produce valid data structure r of the same size, which has the same payload (i.e., the data stored in the data structure and not checked by `repOK`).

Memoization. Oftentimes, a fault in software or hardware is exercised repeatedly, causing similar errors to recur. Applying the baseline repair technique on each data structure suffers from re-exploration when the underlying Korat search for repeated repairs overlap. Our key insight is to enhance the performance of data structure repair for similar recurring errors using *memoization*.

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442055

```
class List {
  Node head; int size;
  static class Node {
    int value;
    Node next, prev;
  }
  boolean repOK() {
    if (head == null) return false;
    Set<Node> visited = new HashSet<>();
    visited.add(head);
    Node current = head;
    while (true) {  // Check circularity
      Node next = current.next;
      if (next == null) return false;
      // Check prev is the converse of next
      if (next.prev != current) return false;
      current = next;
      if (!visited.add(next)) break;
    }
    return visited.size() == size;  // Check size
  }
  static IFinitization finList(int n) {
    IFinitization f = createFinitization(List.class);
    IObjSet nodes = f.createObjSet(Node.class, n);
    f.set("head", nodes);
    f.set("size", f.createIntSet(n, n));
    f.set(Node.class, "value", f.createIntSet(0, 1));
    f.set(Node.class, "next", nodes);
    f.set(Node.class, "prev", nodes);
    return f;
  }
}
```

Figure 1: The Java class with `repOK` and `finitization` methods modeling doubly linked lists with binary values

We present *unsat region*, our memoization construct that succinctly memoizes consecutive invalid candidate vectors enumerated by a Korat search, as a pair of *start* and *end* candidate vectors, to avoid re-exploring that unsat region when repairing similar data structure errors in the future.

Data structure repair for recurring errors. We present our memoized technique for repairing data structure errors that recur, which maintains a collection of unsat regions in tandem with data structure repair to amortize the cost of a Korat search over repeated repairs. Each data structure repair memoizes a new unsat region or extends an existing unsat region to further benefit future repairs.

Evaluation. We evaluated our techniques on 8 complex data structure subjects studied by prior work [1, 5, 10, 11, 24]. Experimental results show that our memoized technique is faster than our baseline technique for repairing data structure errors that recur, and requires a small amount of storage for maintaining unsat regions.

2 BACKGROUND AND EXAMPLE

We present an overview of Korat [1] and demonstrate how it can be applied in the context of constraint-based data structure repair. We further illustrate how memoization can amortize the cost of a Korat search over repeated repairs by avoiding re-exploration.

2.1 Korat basics

Consider using Korat to enumerate a suite of all doubly linked list structures with binary values. The user input to Korat is a Java class

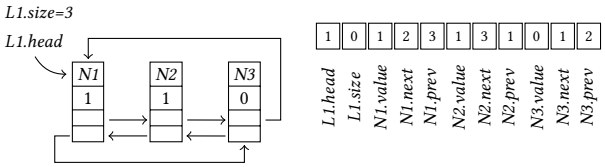


Figure 2: A valid doubly linked list structure L1 with 3 nodes (left) and its candidate vector representation (right)

with `repOK` and `finitization` methods, where `repOK` checks for desired structural properties and `finitization` specifies bounds on the number of objects for each type, and a *field domain* (a finite list of values) for each field of each type, to bound the search.

Korat internally represents a candidate structure as an integer array, namely a *candidate vector* (*cv* in short). Each array element represents a field of an object in the candidate structure by storing the index in the element's field domain that corresponds to the value of that field in the candidate structure.

Korat starts the search by running `repOK` on an initial structure that corresponds to a candidate vector with all elements set to zero (i.e., each field is assigned to the first value in its field domain). Korat monitors field accesses during `repOK` execution to enumerate the next structure by incrementing the candidate vector element for the last accessed field to the next index in its field domain and backtracking the search when it exhausts all choices for that field.

Korat continues this `repOK`-driven search to enumerate every valid structure *c* within the user given bounds such that `c.repOK()` returns `true`. Korat eliminates isomorphic structures to prune the search without compromising completeness.

Figure 1 shows the `List` Java class that models doubly linked lists with binary values. The `List` class declares an internal `Node` class, which holds an integer value `value` and two references `next` and `prev`. A list object has a `head` field and stores the number of nodes reachable from `head` in the `size` field.

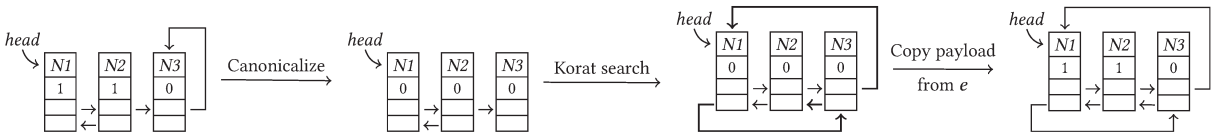
The `repOK` checks (1) circularity along `Node.next`, (2) the converse relation between `Node.next` and `Node.prev`, and (3) the number of nodes reachable from `List.head` matches `List.size`.

The `finitization` (1) takes the number of `Node` objects as input parameter, namely `n`, (2) creates a `Finitization` object, (3) creates a set $s = \{\texttt{null}, N_1, N_2, \cdots, N_n\}$ including `null` and `n` unique `Node` objects, (4) sets the field domain for `List.head` to `s`, (5) sets the field domain for `Node.size` to take only 1 value (i.e., `n`), (6) sets the field domain for `Node.value` to a set of binary values `{0, 1}`, and (7) sets the field domain for `Node.next` and `Node.prev` to `s`.

Figure 2 shows a valid doubly linked list with three nodes that stores binary value 110 with its corresponding candidate vector. For size 3, the search space has $4 \cdot 1 \cdot (2 \cdot 4 \cdot 4)^3 = 2^{17}$ candidate structures. The search space grows exponentially as the size of structures increases. For instance, for size 20 there are 2^{102} candidate structures. Korat prunes the search and finds all 2^{20} valid structures by enumerating 1049262 ($< 2^{21}$) structures in 1.4 seconds.

2.2 Data structure repair using Korat

2.2.1 Our baseline constraint-based repair technique (iRepair^b). Figure 3 illustrates our baseline repair technique for a doubly linked



(a) Erroneous structure e (b) Canonical structure c (c) Valid structure v (d) Repaired structure r

Figure 3: Example data structure repair using our baseline technique (iRepair^b) for a doubly linked list with 3 nodes

list with three nodes, using the `repOK` defined in Figure 1. Figure 3a shows an erroneous data structure e with three `repOK` violations: `N3.prev = null`, `N3.next = N3`, and `N1.prev = null`.

A complete Korat search does not enumerate structure e since the search backtracks on the first violation `repOK` execution detects (i.e., `N3.prev = null`). Hence, we first canonicalize structure e to a structure that is faithful to the Korat search, namely structure c shown in Figure 3b, by setting all fields that were not accessed by a `repOK` execution to the first value in their field domains (i.e., setting `N3.next` to `null` and setting `N1.value` and `N2.value` to 0).

Next, we perform a Korat search starting at structure c and stop the search as soon as Korat enumerates the first valid structure v shown in Figure 3c (where `v.repOK()` returns `true`). The repaired fields, namely `N3.prev = N2`, `N3.next = N1`, and `N1.prev = N3` are shown with bold arrows.

Finally, we copy the payload (i.e., node values), which were reset by canonicalization, from structure e into structure v to reach the repaired structure r shown in Figure 3d. This copy preserves the payload (i.e., the binary sequence 110).

The user given `repOK` in this example (Figure 1) does not have any constraints on `Node.value`. Hence, this field is excluded from our repair algorithm. A different `repOK` that checks node values (e.g., for sorted lists) would consider repairs on node values as well.

2.2.2 Our memoized constraint-based repair technique (iRepair^u). If an error in a program's state is due to a fault in software or hardware, a similar error may occur [34] during the program's execution. For instance, consider repairing an erroneous structure identical to e from Figure 3a but with a different payload (e.g., 101 instead of 110) or an additional field corruption (e.g., a self-loop).

While the underlying Korat search for such repairs would be similar (e.g., identical in the different payload scenario), iRepair^b would still run the Korat search and redundantly enumerates previously explored structures. We illustrate how our memoized technique (iRepair^u) succinctly memoizes a Korat search in tandem with each repair, to enable faster repair of similar errors in the future.

Figure 4 (the left half) shows the sequence of candidate vectors enumerated by a complete Korat search to find all doubly linked lists with three nodes. We use this sequence to illustrate the re-exploration in the underlying Korat search when repairing similar errors and how memoization can avoid this re-exploration.

Figure 4 (the right half) shows the canonical forms for three erroneous doubly linked lists with three nodes, namely c_1, c_2, c_3. Applying either of our techniques on any of these three similar structures would find valid structure v but with different number of enumerations. Specifically, consider the underlying Korat search performed by iRepair^b and iRepair^u for three consecutive repairs.

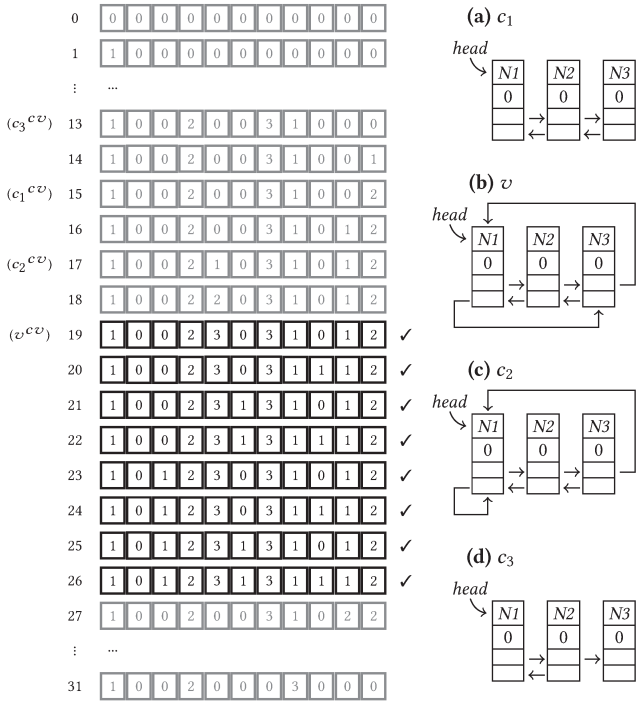


Figure 4: Candidate vectors enumerated by a complete Korat search to find valid doubly linked lists with 3 nodes storing binary values (on the left), illustrating the underlying Korat search for finding the next valid structure v (#19) for canonical erroneous structures c_1 (#15), c_2 (#17), and c_3 (#13). Candidate vectors for valid structures are marked with ✓

Repair scenario 1 (search). Figure 4a shows canonical structure c_1, which corresponds to candidate vector #15 ($c_1{}^{cv}$). Applying iRepair^b performs a Korat search starting at $c_1{}^{cv}$ to find the first valid candidate vector v^{cv} (i.e., #19) by enumerating 5 candidate vectors (#15 to #19), and finds valid data structure v (Figure 4b).

Similarly, iRepair^u enumerates the same candidate vectors, but additionally, memoizes unsat region $[c_1{}^{cv}, v^{cv})$, as a pair of candidate vectors, to avoid re-exploring this region for future repairs when there is an overlap in the underlying Korat search.

Repair scenario 2 (no search). Now consider repairing structure c_2 (Figure 4c). Applying iRepair^b on c_2 enumerates 3 candidate vectors (#17 to #19), which overlaps with the candidate vectors enumerated for repairing c_1 (i.e., a redundant search). In contrast, iRepair^u finds that $c_2{}^{cv}$ belongs to the memoized unsat region $[c_1{}^{cv}, v^{cv})$ and finds v *without* a Korat search.

```
Object iRepair^b(Object s, Method repOK, Finitization f) {
  // Create an instrumented version of "s" needed by the
  // Korat search
  Object e = instrument(s, f);
  // Canonicalize "e" into a candidate vector
  CV canonicalCV = canonicalize(e, repOK, f);
  // Perform a Korat search starting at "canonicalCV" to
  // find "validCV"
  CV validCV = nextValidCandidate(canonicalCV, repOK, f);
  // Copy the data payload from "e" into "validCV"
  copyDataPayload(e, repOK, f, validCV);
  // Construct the repaired structure based on "validCV"
  return buildCandidate(validCV);
}
```

Figure 5: Data structure repair using Korat

Repair scenario 3 (partial search). Now consider repairing structure c_3 (Figure 4d). iRepair^b enumerates 7 candidate vectors (#13 to #19) to find valid structure v. Since ${c_3}^{cv}$ does not belong to the memoized unsat region $[{c_1}^{cv}, v^{cv})$, iRepair^u also runs a Korat search, but keeps track of when the search reaches the start of a memoized unsat region.

After 3 steps (#13 to #15), the Korat search indeed reaches ${c_1}^{cv}$, which is the start of memoized unsat region $[{c_1}^{cv}, v^{cv})$. Hence, iRepair^u skips the rest of the search and returns valid structure v (i.e., 57% fewer enumerations compared to iRepair^b), while extending $[{c_1}^{cv}, v^{cv})$ into $[{c_3}^{cv}, v^{cv})$ to further benefit future repairs.

3 IREPAIR

We first present our baseline constraint-based data structure repair technique that uses Korat. We then introduce our memoization construct, namely *unsat region*, which memoizes the result of a local Korat search as a candidate vector pair, to avoid re-exploring the same region when repairing a similar erroneous data structure in the future. Finally, we present our memoized technique that maintains a collection of unsat regions in tandem with data structure repair to prune the Korat search.

3.1 Data structure repair using Korat (iRepair^b)

Figure 5 shows the pseudocode for our baseline data structure repair technique, which takes an erroneous data structure `s` (of type `C`), the `repOK` predicate that implements desired structural properties (provided by the user in class `C`), and the `Finitization` object that specifies bounds on the state space (provided by the user or constructed dynamically by traversing the object graph of `s` to find the *field domain* for each type). The repair procedure is as follows:

First, we create an instrumented version of class `C`, namely `C'`, by adding a primitive field for each non-static user defined field in `C`, which Korat search needs to monitor field accesses [1, 22]. We then use a work-list based algorithm to traverse the object graph of `s` using Java reflection to create a copy of `s`, namely `e` (of type `C'`).

Second, we canonicalize object `e` into a candidate vector that is faithful to the Korat search (i.e., a candidate vector that a complete Korat search would enumerate), by running `repOK` on `e`, and encoding all fields that were accessed to their *canonical* values based on the linearization algorithm for *rooted graphs* [31], and encoding other fields to the first index in their field domains (i.e., zero).

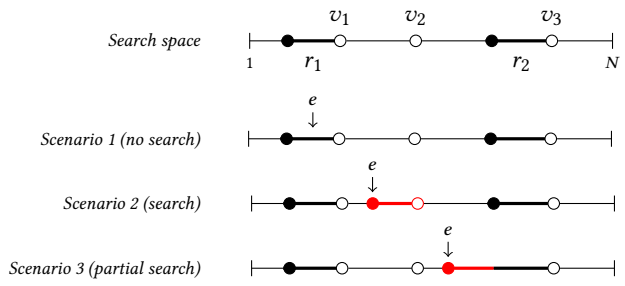


Figure 6: The `memoizedNextValidCandidate` visualization for different repair scenarios

Third, we run a local Korat search, namely `nextValidCandidate`, starting at the canonicalized candidate vector `canonicalCV` until Korat finds the next valid candidate vector `validCV` or terminates without finding a solution. We discuss repair completeness and why this local search is a good repair heuristic later in Section 5.

Next, for fields of `validCV` that are not accessed by a `repOK` invocation, we copy their canonical values from object `e`. This step preserves the data structure payload (i.e., fields that are unchecked by the user given `repOK`) without compromising correctness.

Finally, we construct the repaired structure based on `validCV` using Korat's `CandidateBuilder` API and return the result.

3.2 Unsat regions

For a subject with `repOK` and `finitization`, we define an *unsat region* as a candidate vector pair $[e^{cv}, r^{cv})$, where (1) both candidate vectors are canonicalized, (2) e^{cv} is enumerated before r^{cv} by a Korat search, (3) r^{cv} is valid (i.e., running `repOK` on structure r returns `true`), and (4) e^{cv} and all candidate vectors enumerated between e^{cv} and r^{cv} are invalid (with respect to `repOK`).

We define the following primitive operations for unsat regions:

(1) `lessThan(CV c, CV v)`. Candidate vector c is less than candidate vector v if c is enumerated before v by the Korat search. This can be determined (without a Korat search) by lexicographically comparing the candidate vector elements of the two arrays based on their field access order [1].

(2) `equalTo(CV c, CV v)`. Candidate vector c is equal to candidate vector v if both candidate vectors contain the same number of elements, and all corresponding pairs of elements in the two candidate vectors match.

(3) `inRegion(CV c, R reg)`. Candidate vector c is in region $reg = [e, r)$ if *equalTo*(c, e), or *lessThan*(e, c) and *lessThan*(c, r).

3.3 Memoized data structure repair (iRepair^u)

If an error in a program's state is caused by a fault in software or hardware, a similar error might occur [34] during the program's execution. Our baseline technique applies `nextValidCandidate` on each input structure, which suffers from re-exploration when the underlying Korat search overlaps with a prior search. Our key insight is to use memoization to amortize the cost of a Korat search over repeated repairs of similar erroneous data structures.

Figure 6 visualizes a search space of N candidate vectors that a complete Korat search would enumerate. Consider three valid

```
// Memoized unsat regions. A "key -> value" mapping shows
// unsat region [key, value)
TreeMap<CV, CV> regions = new TreeMap<CV, CV>();

CV memoizedNextValidCandidate(CV e, Method repOK,
     Finitization fin) {
    Map.Entry<CV,CV> r1 = regions.floorEntry(e);
    // Scenario 1 (no search)
    if (r1 != null && inRegion(e, r1))
        return r1.value;
    Map.Entry<CV,CV> r2 = regions.ceilingEntry(e);
    CV q = (r2 != null) ? r2.key : null;
    CV v = nextValidCandidate(e, q, repOK, fin);
    if (v != null) {  // Scenario 2 (search)
        regions.put(e, v);
        return v;
    }
    if (q != null) {  // Scenario 3 (partial search)
        regions.remove(r2.key);
        regions.put(e, r2.value);
        return r2.value;
    }
    return null;
}
```

Figure 7: The `memoizedNextValidCandidate` procedure (used in iRepair^u)

candidate vectors v_1, v_2, v_3, and two memoized unsat regions r_1 and r_2 on a number line. The different scenarios for the search performed by `memoizedNextValidCandidate` (our memoized search) on invalid candidate vector e are discussed below:

- **Scenario 1 (no search).** Since e belongs to a known unsat region, namely r_1, simply return the end of r_1 without a Korat search (i.e., no calls to `nextValidCandidate`).
- **Scenario 2 (search).** Run `nextValidCandidate` and return the solution (i.e., v_2) as the repair, while memoizing $[e, v_2)$ as a new unsat region.
- **Scenario 3 (partial search).** Run `nextValidCandidate` until reaching the start of an unsat region (i.e., r_2). Return v_3 as the repair, while extending r_2 into a larger unsat region $[e, v_3)$ to further benefit future repairs.

Figure 7 shows the `memoizedNextValidCandidate` procedure, which takes a canonicalized candidate vector `e`, the `repOK` predicate, and the `Finitization` object, and returns the next valid candidate vector. This procedure is functionally identical to `nextValidCandidate`, but additionally maintains a map of known unsat regions in tandem with each search as discussed below:

(1) If `e` belongs to a known unsat region, say `r1`, return the end of `r1` (i.e., `r1.value`), without calling Korat (Scenario 1).
(2) Find the first memoized unsat region `r2` in the map that starts after the invalid candidate vector `e` (if any).
(3) Perform a local Korat search starting at `e` until the beginning of `r2` if `r2` exists (to avoid re-exploring `r2`), or else do a Korat search starting at `e` until the next valid candidate vector is found, or the search has completed.
(4) If a valid candidate vector `v` was found, add a new unsat region `[e, v)` to the map of unsat regions (Scenario 2).
(5) If no valid candidate vector was found, while `r2` exists, return `r2.value` as the next valid candidate vector, while extending the beginning of `r2` to `e` in the map, i.e., replacing unsat region `r2=[r2.key, r2.value)` with the larger unsat region `[e, r2.value)`, where `lessThan(e, r2.key)`, to further benefit future repairs (Scenario 3).
(6) Otherwise, return `null`, as `e` cannot be repaired using a Korat search. (See Section 5 for repair completeness.)

The iRepair^u procedure (not displayed) is identical to the iRepair^b procedure shown in Figure 5, except that instead of a call to `nextValidCandidate`, which may suffer from re-exploration, iRepair^u calls `memoizedNextValidCandidate`, which uses memoization to prune a search if it overlaps with a prior search.

4 EVALUATION

To evaluate the efficacy of our repair techniques, we implemented iRepair^b (our baseline technique) and iRepair^u (our memoized technique) by extending Korat [19]. We next discuss our methodology and studied subjects, and then present our experimental results.

4.1 Methodology

We conduct two repair experiments on 8 complex data structure subjects studied by prior work in bounded exhaustive testing and data structure repair [4, 5, 8, 10, 24, 27, 34].

The first experiment repairs all non-isomorphic erroneous data structures for small sizes, ranging from 3 to 19 nodes. This experiment studies the brute-force scenario to evaluate potential gains from memoization for error recovery of a program that encounters many different erroneous data structures.

The second experiment generates a few larger valid data structures ranging from 10 to 1000 nodes. For each valid data structure, we inject an error by randomly corrupting object fields and then repair the erroneous structure. We then repeat the fault injection and repair steps 20 times on the valid structure to simulate similar recurring errors encountered by a program execution. We repeat this procedure for 100 valid data structures (per subject) and report average numbers (of the 100 runs) after each repair.

We next discuss each experiment procedure in detail and present our research questions.

4.1.1 Experiment 1. Repairing all small structures. Recall that Korat, when applied as a test input generation tool [1], enumerates all non-isomorphic structures up to the given bounds. Hence, for each studied subject, we use Korat to generate the erroneous test structures. Our experiment procedure is described below.

(1) Run a complete Korat search on each subject (for each size) and create a sequence of all erroneous data structure objects enumerated by Korat.
(2) Shuffle the sequence with a random seed to ensure we do not repair the structures in the same order as were enumerated by the Korat search, which can unfairly benefit the memoization in iRepair^u.
(3) Repair each erroneous data structure in the sequence once using iRepair^b and next using iRepair^u.

We run this experiment procedure 3 times and report average numbers. Needless to say, each run starts from a clean state with an empty cache of unsat regions (for iRepair^u).

4.1.2 Experiment 2. Repairing a few larger structures. For each studied subject, we run the fault injection experiment procedure below.

(1) Generate a random valid data structure `s` such that `s.repOK()`.
(2) Corrupt a field of `s` at random to create an erroneous data structure `e` such that `!e.repOK()`. A corruption is a tuple of three elements ⟨`o, f, v`⟩, where field `f` of object `o` is assigned to value `v`, which is `null` or another object of the same type for reference types, and another primitive value for primitive types.
(3) Repair the erroneous data structure once using iRepair^b and then using iRepair^u.
(4) Repeat steps 2 and 3 twenty times to introduce recurring errors and report the results after each repair.

We repeat this procedure 100 times and report average numbers. For instance, for the 5th repair (out of the 20 consecutive repairs), we report the average numbers of the 100 runs for their 5th repair. Each run starts from an empty cache of unsat regions (for iRepair^u).

Repair quality. Unlike Experiment 1 that starts the repair procedure from erroneous structures, this experiment starts from a valid data structure and injects faults. Hence, we can measure the repair quality of iRepair by calculating how similar the repaired structure (say r) is to the ideal repair, which is the original valid data structure before the fault injection (say o).

We define similarity between r and o as $S = (L-D)/L \cdot 100$, where D is the number of elements that differ between their candidate vector representations cv^r and cv^o, respectively, and L is the length of the candidate vectors. Note that L is the same for both structures as our repair experiment preserves the structure size. Evidently, higher similarity values are better (e.g., 100% for an ideal repair).

4.1.3 Research questions. Together, these two repair experiments answer the following research questions:

RQ1: What is the iRepair^u speedup (in time) compared to iRepair^b?
RQ2: What are the iRepair^u savings (in search) compared to iRepair^b?
RQ3: What percent of the erroneous data structures repaired by iRepair^u benefit from memoization (cache hit)?
RQ4: What is the size of the unsat regions (cache) maintained by iRepair^u?
RQ5: What is the repair quality of iRepair?

4.2 Subjects

We used 8 complex data structure subjects, namely AVL tree (AVL), binomial heap (BH), binary tree (BT), doubly linked list (DLL), fibonacci heap (FH), red-black tree (RBT), sorted list (SL), and search tree (ST). All subjects except AVL are taken from the publicly available Korat distribution [19] and model standard data structures from Java libraries. For example, DLL and RBT implement `repOK` for `java.util` classes `LinkedList` and `TreeMap`, respectively.

For AVL, we modeled it after a custom AVL tree class from HSQLDB [15], a popular open-source relational database application for Java. In particular, we implemented the `repOK` to specify AVL tree properties for `org.hsqldb.index.IndexAVL`, which is a class to store database indexes using balanced binary search trees.

For Experiment 1, Table 1 shows for each studied subject, its name (and acronym), the structure sizes (in number of nodes), the search space size computed by multiplying the number of choices for each candidate vector element, the number of structures enumerated by a complete Korat search, the number of valid structures found, and the execution time (in milliseconds). The last row (Σ) shows cumulative numbers for each column.

Table 1: Subjects of Study

Subject	Size	Space	Candidates [#]		Time [ms]
			Total	Valid	
AVL tree (AVL)	6	2^{70}	7675	28	116
	7	2^{87}	47762	136	177
	8	2^{104}	199076	288	372
binomial heap (BH)	6	2^{88}	42815	7602	184
	7	2^{109}	261788	107416	333
	8	2^{131}	1323194	603744	1028
binary tree (BT)	9	2^{63}	210444	4862	236
	10	2^{72}	815100	16796	553
	11	2^{82}	3162018	58786	1807
doubly linked list (DLL)	10	2^{82}	1215	1024	90
	11	2^{93}	2275	2048	95
	12	2^{104}	4362	4096	102
fibonacci heap (FH)	3	2^{39}	803	136	127
	4	2^{60}	8634	2310	146
	5	2^{82}	164365	52281	296
red-black tree (RBT)	6	2^{76}	16487	20	152
	7	2^{94}	71704	35	270
	8	2^{112}	322806	64	649
sorted list (SL)	17	2^{220}	525068	65536	738
	18	2^{237}	1114983	131072	1355
	19	2^{254}	2360263	262144	2680
search tree (ST)	6	2^{52}	45233	132	158
	7	2^{64}	340990	429	310
	8	2^{77}	2606968	1430	1479
Σ	n/a	n/a	13656028	1322415	13555

For Experiment 2, we use the same data structure subjects as Experiment 1, but with larger structure sizes, which will be shown later in Figure 9.

4.3 Experimental results

We obtained all data on a machine with 6-core 3.2 GHz Intel Core i7-8700B Processor and 16 GB of RAM, running macOS 10.15.6 and Oracle Java 1.8.0_261.

4.3.1 Experiment 1. Repairing all small structures. We report the results of this experiment in two formats. Table 2 summarizes the cumulative results at the very end of the procedure, when all erroneous structures in the sequence are repaired. Figure 8 plots the results by connecting the data points after each iteration of the experiment procedure (i.e., after each repair) to show the trend of gains (for iRepair^u) as new structures are being repaired.

Table 2 shows for each studied data structure subject, its acronym, the structure sizes (in number of nodes), the number of erroneous structures repaired (by each technique), the total iRepair^b repair time (in seconds), the total number of candidate vectors enumerated by iRepair^b, the iRepair^u speedup in repair time compared to iRepair^b, the iRepair^u savings in number of candidate vectors enumerated compared to iRepair^b, the iRepair^u cache hit (i.e., the percent of repairs that benefited from a memoized unsat region by

Table 2: Experiment 1 Results after Repairing all Non-isomorphic Erroneous Data Structures using iRepair^b and iRepair^u

Subject	Size	Structures [#]	iRepair^b		iRepair^u				
			Time [sec]	Explored [#]	Speedup [×]	Savings [%]	Hit [%]	Cache Unsat Regions [#]	Size [KB/MB]
AVL	6	7647	3.77	9325770	19.84	98.20	91.22	24	10.7 KB
	7	47626	42.76	133666109	42.24	99.21	95.84	119	62.1 KB
	8	198788	629.76	1804622336	108.79	99.66	97.60	256	151.6 KB
BH	6	35213	24.34	91783499	8.97	89.63	82.66	941	461 KB
	7	154372	433.93	1502582258	6.00	83.06	80.97	3627	1.9 MB
	8	719450	9209.86	28775831355	124.34	99.27	89.18	33555	21 MB
BT	9	205582	3.03	5276951	2.18	95.94	89.87	4862	1.5 MB
	10	798304	13.74	22956277	2.28	96.42	90.77	16796	5.6 MB
	11	3103232	62.66	98835513	2.19	96.79	91.50	58786	21.3 MB
DLL	10	191	0.01	14211	5.42	96.36	83.25	1	0.3 KB
	11	227	0.01	20334	4.71	96.88	84.73	1	0.4 KB
	12	266	0.01	28257	4.03	97.30	86.84	1	0.4 KB
FH	3	667	0.02	25018	1.62	90.48	72.96	61	19.3 KB
	4	6324	0.18	422816	2.61	97.03	74.67	765	316.2 KB
	5	112084	4.15	9920729	3.01	98.59	75.24	13646	6.8 MB
RBT	6	16467	9.43	24050422	36.64	99.65	96.90	20	10.8 KB
	7	71669	179.58	442313839	132.55	99.90	98.40	35	21.9 KB
	8	322742	4152.15	9323943416	555.99	99.98	99.20	64	45.7 KB
SL	17	459532	14.19	5073102	1.26	90.27	77.82	32769	25.6 MB
	18	983911	32.82	11566047	1.27	90.92	78.86	65537	54.3 MB
	19	2098119	76.28	26186013	1.25	91.48	79.78	131073	114.5 MB
ST	6	45101	2.51	7831505	8.05	98.86	97.43	132	41.1 KB
	7	340561	44.39	136207181	17.02	99.55	98.88	429	153.7 KB
	8	2605538	832.36	2381620018	33.70	99.83	99.50	1430	579.7 KB
Σ	n/a	12333613	15771.94 (4.38 hr)	44814102976	n/a	n/a	n/a	364930	254.3 MB
avg	n/a	513901	657.16 (10.95 min)	1867254291	46.9	96.1	88.1	15205	10.6 MB

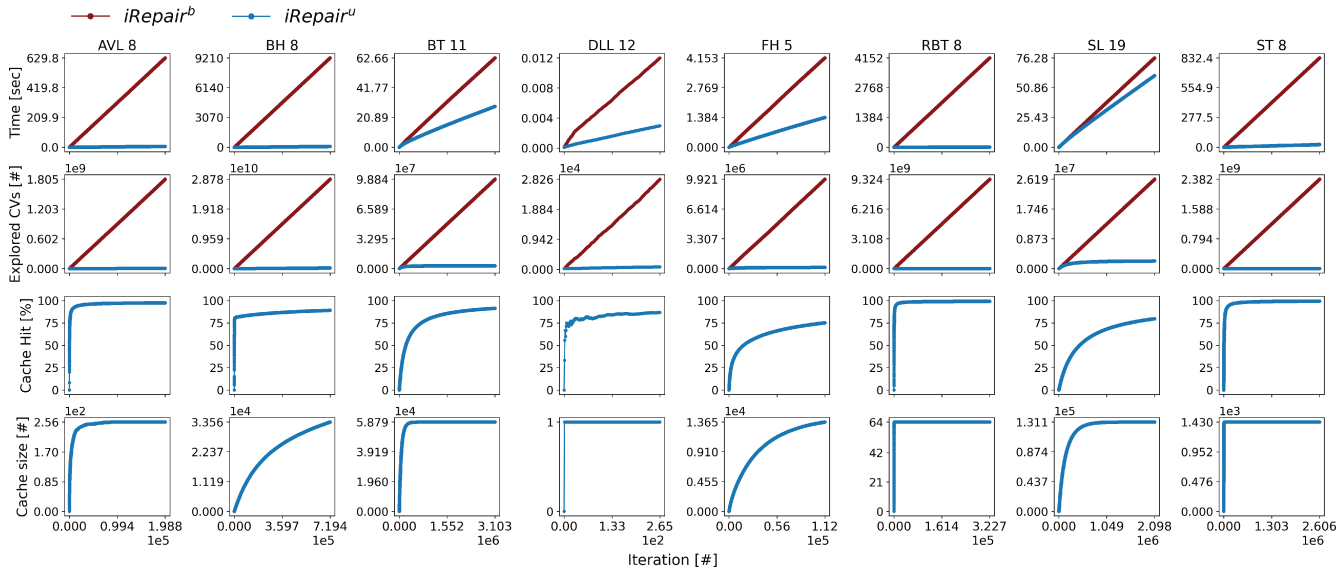


Figure 8: Experiment 1 results for iRepair^b (red) and iRepair^u (blue) after each repair, showing the cumulative repair time (in seconds), the total number of candidate vectors enumerated, the cache hit, and the cache size (in number of unsat regions)

performing a partial search or no search, e.g., as shown in Figure 6), and the cache size (in number of unsat regions and in bytes). The last two rows show the sum (Σ) and average (*avg*) of each column, when applicable.

Figure 8 shows the plots for the largest size of each subject. For each subject (column), there are 4 plots (rows). Row 1 shows the cumulative repair time (in seconds). Row 2 shows the cumulative number of candidate vectors enumerated. Row 3 shows the cumulative cache hit (in percent). Row 4 shows the cumulative cache size (in number of unsat regions). The x-axis shows the iteration number in the experiment procedure, which is common for all plots shown for a given subject. The red and blue curves in the plot correspond to the iRepair^b and iRepair^u, respectively. For each plot, the y value at x_{max} (i.e., the last iteration of the experiment procedure) corresponds to the numbers shown in Table 2.

RQ1: Speedup in time. Column 6 in Table 2 shows on average iRepair^u was 47 times faster than iRepair^b. In total, iRepair^b spent 4.38 hours to repair the sequence of 12333613 erroneous structures studied, while iRepair^u spent 5.48 minutes to repair the same sequence. Row 1 in Figure 8 shows for all subjects, iRepair^u (the blue line) outperforms iRepair^b (the red line).

RQ2: Savings in search. Column 7 in Table 2 shows on average iRepair^u performed 96.1% fewer enumerations compared to iRepair^b. In total, iRepair^b and iRepair^u enumerated 44814102976 and 498258122 candidate vectors, respectively, to repair the same sequence of erroneous data structures. Row 2 in Figure 8 shows for all subjects, iRepair^u enumerates fewer candidate vectors compared to iRepair^b by not re-exploring known unsat regions.

RQ3: Cache hit. Column 8 in Table 2 shows on average 88.1% of repairs made by iRepair^u benefited from a memoized unsat region from an earlier repair (i.e., needed a partial search or no search). Row 3 in Figure 8 shows the cache hit for AVL, RBT, and ST grows faster than other subjects, and peaks at 97.60%, 99.20%, and 99.50%, respectively. We observed from Table 1 that these subjects have sparse state spaces, where there are few valid candidate vectors among a large number of enumerations, a case that benefits memoization in iRepair^u by pruning larger unsat regions.

RQ4: Cache size. Column 9 in Table 2 shows iRepair^u memoized 364930 unsat regions (i.e., 729062 candidate vector pairs), which is 97% smaller than the 12333613 studied erroneous structures. Column 10 in Table 2 shows iRepair^u maintained a cache size of 254 MB, while we measured the total size of invalid candidate vectors for the repaired structures was 1.7 GB. We further observed iRepair^u requires minimal storage for subjects that have a sparse state space like RBT and ST where there are few valid instances (among all enumerations) and DLL where there are few invalid instances (as shown in Table 1).

4.3.2 Experiment 2. Repairing a few larger structures. We chose larger structures ranging from 10 to 1000 nodes. The sizes vary between subjects based on the complexity of `repOK` and the size of the state space. Figure 9 shows the results of repairing 20 similar erroneous data structures (per subject). For each subject (column), there are 5 plots (rows), showing the repair time (milliseconds), the number of candidate vectors enumerated by the underlying Korat search, the cache hit (in percent), the cache size (in number of unsat regions), and repair similarity (as defined in Section 4.1.2).

For **RQ1** and **RQ2**, we observed smaller gains compared to Experiment 1 (because we do fewer repairs). For instance, on average (for the 20 repairs of all subjects), iRepair^u was 2.4 times faster than iRepair^b and enumerated 52.7% fewer candidate structures.

For the first repair of any subject, the underlying search for both techniques enumerated the same number of candidate structures in similar execution times. Starting from the second repair, we see gains from iRepair^u when a Korat search overlaps with a previously memoized unsat region (i.e., a cache hit).

For BT and DLL we see significant gains even after a handful of repairs (e.g., 12.8 times and 35.7 times faster performance, respectively, for the fifth repair). For BH, FH, and SL, we see smaller gains in execution time (e.g., 1.6 times for FH), as the unsat regions for these subjects memoize smaller number of erroneous structures due to the higher ratio of valid to invalid structures in their enumerated search space as was shown in Table 1 (for smaller sizes).

For **RQ3**, we observe an increasing trend in cache hit for all subjects, where it starts from 0% (for the first repair) and increases for future repairs. Some subjects like BT, DLL, and FH have faster increases on the first few repairs. All subjects, except SL, reached above 70% cache hit within the 20 consecutive repairs.

For **RQ4**, we observe the cache size starts from 1 (the first unsat region that is memoized after the first repair by iRepair^u) and increases for all subjects except DLL, due to all valid structures being consecutive for DLL (as shown in the Example from Section 2.2.2). Note that the one unsat region gets extended to larger unsat regions throughout consecutive repairs (i.e., Scenario 3 from Figure 6), but the cache size remains the same (i.e., 1). SL has the highest number of unsat regions, which explains why it has the lowest cache hit as well. This was also expected as this subject had the highest number of unsat regions in Experiment 1 (Table 2).

RQ5: Repair quality. Recall from Section 4.1.2 that we measure similarity between a repaired structure and the original valid structure before fault injection. As shown, both iRepair variants, which create the same repairs, result in high similarity, with minimum, average, and maximum (among the 20 repairs of all subjects) being 79.4%, 92.7%, and 100%, respectively. Hence, `nextValidCandidate`, which performs a local Korat search, is a good repair heuristic.

5 DISCUSSION

Repair heuristic. The `nextValidCandidate`, which is the backbone of iRepair, is a heuristic that performs a local Korat search starting at the canonical form of an erroneous structure to find the next valid structure. Hence, this approach limits unnecessary mutations by nature of the Korat search [1]. Intuitively, this local search is similar to finding one solution for a SAT formula, which makes it computationally more efficient than enumerating all valid instances, which is analogous to model counting [11].

Section 4.3.2 showed this heuristic resulted in repairs with high similarity to the ideal repair (i.e., 92.7% on average). Additionally, we measured the similarity between erroneous structures and their corresponding repaired structures was 91.8% on average. We do not show numbers per subject due to space constraints.

Repair completeness. iRepair uses a backend Korat solver for data structure repair, which performs a forward search, i.e., starts the search from an invalid candidate vector until a solution is found. Hence, it can repair a data structure iff its canonical form appears before the last valid candidate vector that a complete Korat search enumerates. We measured for the subjects and sizes studied, iRepair was able to repair almost all (99.52%) of the studied structures.

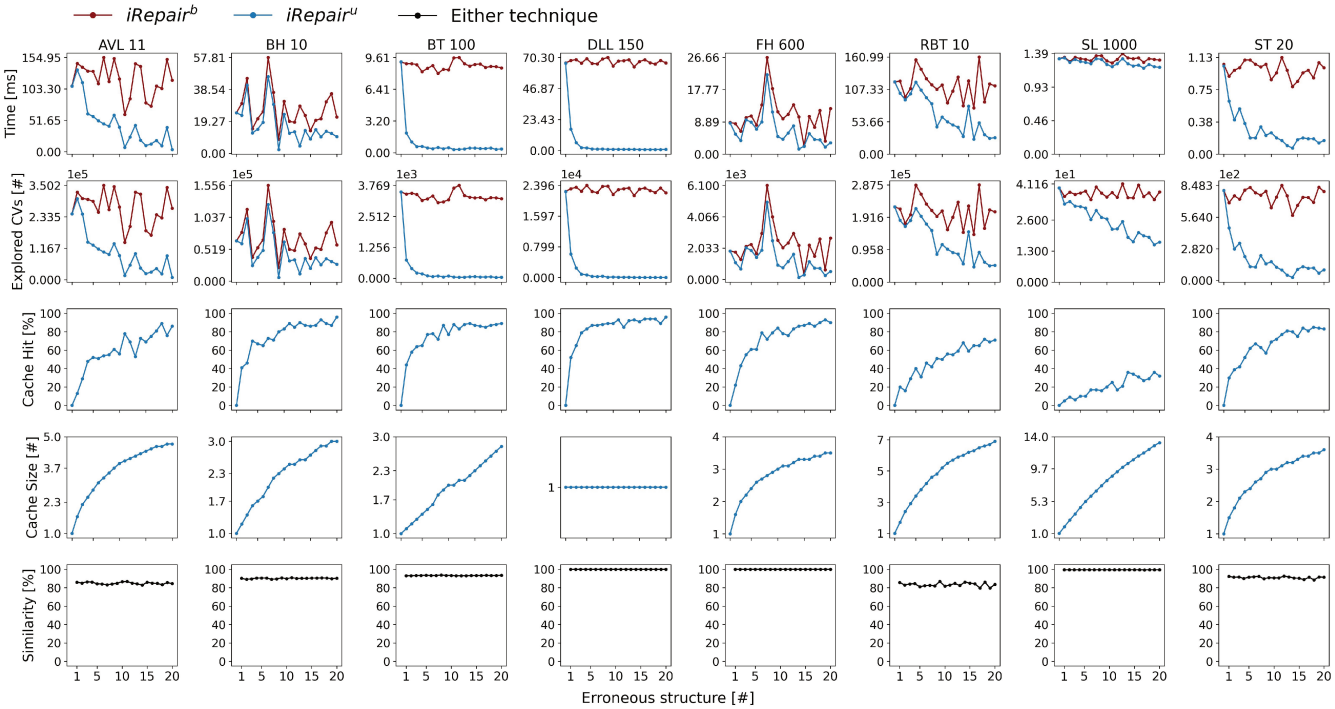


Figure 9: Experiment 2 results for iRepair^b (red) and iRepair^u (blue) showing repair time, number of candidate vectors enumerated, cache hit, cache size (in number of unsat regions), and repair similarity (black), which is the same for either technique

Role of `repOK` in repair. Based on the constraints implemented by the user given `repOK`, iRepair can eliminate violations in both structural properties and the payload stored in the data structure. If `repOK` only checks structural properties (e.g., Figure 1), iRepair preserves the payload. If `repOK` additionally checks constraints on the payload itself (e.g., the *sorted property* for values stored in SL), iRepair can update the payload as well to eliminate data violations.

The repairs made by iRepair depend on how `repOK` is formulated due to the nature of the Korat search that monitors field access order [1]. Hence, two implementations of `repOK` that check constraints in a different order result in different repaired structures. While this might seem as a limitation, it gives users control to define a precedence on what properties the algorithm must repair first. For instance, the `repOK` for SL checks the sorted property after other constraints to mutate the payload only when necessary.

Our memoized technique offers performance gain by not re-exploring known unsat regions. Hence, the larger the unsat regions are, the higher the performance gain would be. Consequently, data structures that have more sophisticated `repOK` methods (i.e., `repOK` returns `true` less often) receive a higher performance gain when there is a cache hit (i.e., Scenarios 1 and 3 from Figure 6). For instance, in both sets of experiments, RBT gets a higher performance boost from iRepair^u over iRepair^b, when compared with SL.

The canonicalization and unsat regions in iRepair rely on the `repOK` implementation to access fields in a deterministic order and deterministically return `true` or `false` when run repeatedly on a given structure. The subjects studied in our evaluation and prior work [1, 5, 11, 24] all use deterministic `repOK`s.

Caching repair memoization: Since repair is a last resort technique and happens infrequently, a part of the memoization cache may be computed offline in order to speed up the initial repairs.

6 RELATED WORK

Data structure repair is an error recovery approach to make a program resilient to runtime data corruptions. While traditional systems like Lucent 5ESS telephone switch [13] and IBM MVS operating system [26] used dedicated repair routines, they lacked generality and extensibility. Demsky et al. [2, 3] introduced the idea of using data structure integrity constraints as a basis for repair.

Juzi [17] introduced the use of imperative predicates as constraints for data structure repair using generalized symbolic execution [18]. Our technique follows the spirit of Juzi. The difference is that iRepair does not require building or solving path conditions that are required in symbolic execution.

DSDSR [14] used dynamic symbolic (or concolic) execution [12, 30] for data structure repair. Tarmeem [33] and PBnJ [29] leveraged the SAT-based Alloy tool-set [16] to enable repair with respect to richer specifications. While rich post conditions allow more accurate repairs (than just `repOK` methods), they require check-pointing pre-states and generally admit less scalable solutions due to the higher complexity of the underlying constraint solving problem.

A number of techniques optimized data structure repair using static analysis [9], abstract undo operations [10], and read/write-barriers [32], which can be applied in tandem with iRepair.

DREAM [34] introduced memoization for data structure repair and defined repair abstractions, which memoize concrete *repair*

actions, i.e., mutations that transform an erroneous state to an acceptable state, to enhance the performance of future repairs when a similar erroneous state recurs. Unsat regions in iRepair^u are orthogonal to repair abstractions and can prune the search even when no repair abstraction is applicable.

Parallel Korat [24] introduced the first parallel technique for test generation and execution using Korat, which finds *equi-distant* candidate vectors during a Korat search, to evenly distribute the search problem among independent workers when the same search problem is being solved repeatedly.

MKorat [6] enhanced Parallel Korat by introducing *invalid ranges* as memoization primitives to prune the search when the same problem is being solved repeatedly. MKorat motivated iRepair^u. The difference is that iRepair^u is an on-the-fly data structure repair tool that performs a local search to find a valid structure in tandem with memoizing that search, while MKorat is a test generation tool that performs a complete search finding all valid structures and then memoizes invalid ranges. Due to its brute-force nature, MKorat does not scale to structures with hundreds of nodes (Figure 9).

SAT solvers [7, 25] offer various approaches for efficient solving of constraints. Incremental SAT and conflict-driven clause learning [20] motivate unsat regions in iRepair^u. The difference is that iRepair^u utilizes a bounded exhaustive approach that operates on user-written Java predicates, which are structurally and semantically different from CNF formulas.

7 CONCLUSION

Data structure repair is a form of error recovery to make software resilient to runtime failures. We presented two constraint-based data structure repair techniques based on the Korat solver. Our baseline technique is effective for repairing a data structure, but suffers from redundant search when applied on similar erroneous data structures. Our memoized technique memoizes the search in tandem with a repair to amortize the cost of a Korat search over repeated repairs. Experimental results showed that our memoized technique outperforms our baseline technique with minimal overhead. This work lays a promising foundation to utilize memoization for constraint-based repair of data structure errors that recur.

ACKNOWLEDGMENTS

We are grateful to Milos Gligoric for in-depth discussions and very valuable feedback on this work. We thank the anonymous reviewers for their useful comments. This work was partially supported by the US National Science Foundation under Grant No. CCF-1718903.

REFERENCES

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis*. 123–133.
[2] Brian Demsky and Martin C. Rinard. 2003. Automatic detection and repair of errors in data structures. In *Conference on Object-oriented Programing, Systems, Languages, and Applications*. 78–95.
[3] Brian Demsky and Martin C. Rinard. 2005. Data structure repair using goal-directed reasoning. In *International Conference on Software Engineering*. 176–185.
[4] Nima Dini, Cagdas Yelen, Zakaria Alrmaih, Amresh Kulkarni, and Sarfraz Khurshid. 2018. Korat-API: a framework to enhance korat to better support testing and reliability techniques. In *Symposium on Applied Computing*. 1934–1943.
[5] Nima Dini, Cagdas Yelen, Milos Gligoric, and Sarfraz Khurshid. 2019. Extension-Aware Automated Testing Based on Imperative Predicates. In *International Conference on Software Testing, Validation and Verification*. 25–36.
[6] Nima Dini, Cagdas Yelen, and Sarfraz Khurshid. 2017. Optimizing Parallel Korat Using Invalid Ranges. In *International Symposium on Model Checking Software*. 182–191.
[7] Niklas Een and Niklas Sorensson. 2003. An Extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*. 502–518.
[8] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. 2007. Assertion-based repair of complex data structures. In *International Conference on Automated Software Engineering*. 64–73.
[9] Bassem Elkarablieh, Sarfraz Khurshid, Duy Vu, and Kathryn S. McKinley. 2007. Starc: static analysis for efficient repair of complex data. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 387–404.
[10] Bassem Elkarablieh, Darko Marinov, and Sarfraz Khurshid. 2008. Efficient Solving of Structural Constraints. In *International Symposium on Software Testing and Analysis*. 39–50.
[11] Antonio Filieri, Marcelo F. Frias, Corina S. Pasareanu, and Willem Visser. 2015. Model Counting for Complex Data Structures. In *International Symposium on Model Checking Software*. 222–241.
[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation*. 213–223.
[13] George Haugk, Frederick M. Lax, Robert D. Royer, and John R. Williams. 1985. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal* 64, 6 (1985), 1385–1416.
[14] Ishtiaque Hussain and Christoph Csallner. 2010. Dynamic Symbolic Data Structure Repair. In *International Conference on Software Engineering*. 215–218.
[15] HyperSQL. 2020. HyperSQL Java database. http://hsqldb.org/.
[16] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*.
[17] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. 2005. Repairing Structurally Complex Data. In *SPIN Workshop on Model Checking Software*. 123–138.
[18] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 553–568.
[19] Korat. 2020. Korat Webpage. http://korat.sourceforge.net.
[20] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. 2006. SMT Techniques for Fast Predicate Abstraction. In *International Conference on Computer Aided Verification*. 424–437.
[21] Barbara Liskov and John Guttag. 2001. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley.
[22] Darko Marinov. 2005. *Automatic testing of software with structurally complex inputs*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/30161
[23] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. 2007. Korat: A Tool for Generating Structurally Complex Test Inputs. In *International Conference on Software Engineering*. 771–774.
[24] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. 2007. Parallel test generation and execution with Korat. In *International Symposium on Foundations of Software Engineering*. 135–144.
[25] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*. 530–535.
[26] Samiha Mourad and Dorothy Andrews. 1987. On the Reliability of the IBM MVS/XA Operating System. *IEEE Transactions on Software Engineering* (1987), 1135–1139.
[27] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. 2020. Unifying execution of imperative generators and declarative specifications. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 217:1–217:26.
[28] Martin Rinard. 2003. *Resilient computing*. Technical Report. MIT Computer Science and Artificial Intelligence Laboratory.
[29] Hesam Samimi, Ei Darli Aung, and Todd Millstein. 2010. Falling Back on Executable Specifications. In *European Conference on Object-Oriented Programming*. 552–576.
[30] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *International Symposium on Foundations of Software Engineering*. 263–272.
[31] Tao Xie, Darko Marinov, and David Notkin. 2004. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *International Conference on Automated Software Engineering*. 196–205.
[32] Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S. McKinley. 2012. History-Aware Data Structure Repair Using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2–17.
[33] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. 2010. Contract-Based Data Structure Repair Using Alloy. In *European Conference on Object-Oriented Programming*. 577–598.
[34] Razieh Nokhbeh Zaeem, Muhammad Zubair Malik, and Sarfraz Khurshid. 2013. Repair Abstractions for More Efficient Data Structure Repair. In *International Conference on Runtime Verification*. 235–250.