History-Aware Data Structure Repair Using SAT

Razieh Nokhbeh Zaeem[1], Divya Gopinath[1], Sarfraz Khurshid[1], and Kathryn S. McKinley[2]

[1] The University of Texas at Austin
{nokhbeh,divyagopinath}@utexas.edu,khurshid@ece.utexas.edu
[2] The University of Texas at Austin and Microsoft Research
mckinley@cs.utexas.edu

Abstract. Data structure repair corrects erroneous executions in deployed programs while they execute, eliminating costly downtime. Recent techniques show how to leverage specifications and a SAT solver to enforce specification conformance at runtime. While this powerful methodology increases the reliability of deployed programs, scalability remains a key technical challenge—satisfying a specification often results in the exploration of a huge state space. We present a novel technique, called *history-aware contract-based repair* for more efficient data structure repair using SAT. Our insight is two-fold: (1) the dynamic program trace of field writes and reads provides useful guidance to repair incorrect state mutations by a faulty program; and (2) we show how to execute SAT using unsatisfiable cores it generates, in an efficient iterative approach on successive problems with increasing state spaces, in order to utilize the history of previous runs as captured in the unsatisfiable core. We implement this approach in a new tool, called Cobbler, that repairs Java programs. Experimental results on two large applications and a library implementation of a linked list show that Cobbler significantly outperforms previous techniques for specification-based repair using SAT, and finds and repairs a previously undetected bug.

1 Introduction

Software systems are pervasive and integrated into almost every aspect of life. Software reliability is essential for life-critical, science, and business applications. Much research addresses producing reliable software in various phases of the software development life cycle before deployment, from analyzing requirements to design, implementation, and testing. However, improving the reliability of an already deployed (possibly faulty) system using error recovery is a less explored area.

In practice, systems are deployed with unknown and known unfixed bugs. When bugs cause failures, the usual tactic is to restart the program because fixing bugs and redeploying software may take months. Although the latter approach may resolve the fundamental source of the problem, system downtime is undesirable and not always feasible. Many *mission critical applications* such as operating systems, may prefer to trade slight deviations in intended functionality for system uptime. Better still, if developers annotate programs with specifications, then the runtime may restore the system state to provide its intended functionality. Continuing program execution by fixing the effect of bugs on-the-fly is called *repair*. Existing techniques for repair have not so far lived up to their full potential, because they are either not general purpose or too inefficient.

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 2–17, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Some critical systems include code that repairs erroneous executions on-the-fly using dedicated application specific repair routines [6,7,13,14]. Recent work introduced general purpose approaches including constraint-based repair [4,8,10] and contract-based repair [18,25], some of which utilize SAT solvers [18,25]. Constraint-based repair emphasizes data structure integrity rules and repairs the data structures when a bug leads to an invariant violation. Contract-based repair adds pre- and post-condition specifications of a method which aid in generating an accurate repair, i.e., a structure that is the same or very close to the one that a correct method would generate. General purpose repair, however, has a huge state space of possible post-states and exploring them to find a solution is currently too expensive to use in practice.

This paper seeks to make repair substantially more efficient by utilizing the *history* of code execution as well as SAT solving. Our insights are two-fold: (1) the dynamic program trace of field writes and reads provides useful guidance to identify incorrect state mutations made by a faulty program; and (2) the unsatisfiable core generated by a SAT run captures core elements of the solver's reasoning, which not only facilitates locating faults but can even be leveraged directly to optimize a successive SAT run. We utilize program traces and unsatisfiable cores in tandem to define an efficient iterative methodology where SAT is run on successive problems with increasing state spaces and each run utilizes the history of the previous run. To our knowledge, our work is the first to use the history of program execution or constraint solving in data structure repair.

History-aware repair utilizes a faulty program execution by focusing repair on fields recently modified or read by the program, thereby reducing the search space for SAT. We record program writes and reads to the key data structure with *barriers*. A barrier is a code sequence that performs an action just prior to a write or read. Barriers are widely available in commercial and research implementations of managed languages, e.g., the HotSpot and Jikes RVM Java Virtual Machines, and the .NET C# system. Our approach inserts barrier instrumentation on writes and reads or piggybacks on existing barriers.

While using the history of program execution aids in improving repair performance, its heuristic nature implies that there exist cases in which we have to perform a broader search and consider fields not included in the execution trace. In such cases, we take advantage of *UNSAT* cores, which are minimal unsatisfiable sub-formulas provided by failed SAT invocations. When SAT invocations fail, we utilize their UNSAT cores to identify faulty fields. A final SAT invocation with the list of faulty fields extracted from the UNSAT core results in a repaired data structure.

We implement repair for Java in a tool called Cobbler. Cobbler uses class invariants and method post-conditions expressed in the Alloy specification language [9]. Cobbler inserts write and read instrumentation for the specified data structures to log dynamic program behavior. When Cobbler detects a violation, it uses a SAT solver to mutate the data structure until it satisfies the specification.

We explore the efficiency and accuracy of Cobbler on microbenchmarks and two open source programs: Kodkod solver [22] and ANTLR [2,16]. We compare our history-aware contract-based repair tool, Cobbler, to contract-based repair alone using PBnJ [18] and Tarmeem [25], two repair tools which leverage user guides and heuristics along with a SAT solver. Cobbler is substantially more efficient and scalable than PBnJ and Tarmeem. We also compare Cobbler with Juzi, which uses data structure

specifications for repair, but does not use method post-conditions [5]. Juzi's dedicated constraint solver is more efficient than Cobbler, but Juzi's repair is applicable to far fewer cases and Cobbler is much more accurate. Our experiments show that for small to moderate instantiations of data structures, Cobbler provides repaired data structures which are 100% to 90% similar to the correct structure in more than 90% of the cases. Cobbler also finds and repairs a previously unknown error in ANTLR.

We make the following contributions: **History-aware contract-based repair** combines the program's dynamic behavior with specifications and the current erroneous state of a program to perform repair. **Read and write barriers for repair** are an unconventional use of barriers to obtain program execution history for repair. **Minimal unsatisfiable cores** provided by SAT solvers help to reduce the search space when a field outside the execution trace should be modified. **Cobbler** is an automated portable framework for repairing Java programs that enhances real applications with repair functionality. **Evaluation** shows that Cobbler efficiently and accurately repairs text-book examples and real world programs. Cobbler's more efficient and accurate repair facilitates the use of repair in real world applications and enhances software reliability.

2 Background

This section describes data structure repair and the Alloy tool-set, which Cobbler uses.

Repair: Data structure repair corrects erroneous executions on-the-fly by enforcing data integrity constraints (also known as `repOK`) and method pre- and post-conditions (contracts). Figure 1 (a) shows the faulty output of a method, which violates the acyclicity constraint as a binary search tree. A repair tool detects the violation and fixes it by removing the dotted edge. Further fixes may be needed to enforce method contracts too.

Alloy tool-set: Alloy is a relational first order logic language [9]. An Alloy model consists of relations and constraints on them. The Alloy Analyzer performs bounded exhaustive analysis of Alloy models. A *bound* is a function which maps each relation to a set of tuples (bound: $R \to 2^T$), where each tuple consists of atoms. For each relation R, two sets are defined: a lower bound $LB(R)$, which includes all tuples that R *must* have in its instance ($inst(R)$), and an upper bound $UB(R)$, which includes all tuples that R *may* have in its instance. Therefore, $LB(R) \subseteq inst(R) \subseteq UB(R)$. Figure 1 (b) shows the relational representation of the Java object graph shown in Figure 1 (a).

We use Kodkod [22], the back-end of Alloy Analyzer, which is a SAT-based constraint solver for first order logic that supports relations, transitive closure, and partial

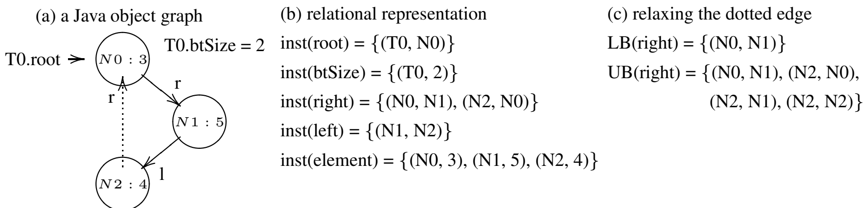


Fig. 1. Relational representation of data structures in Alloy models

models. Kodkod provides a finite model for satisfiable specifications and an UNSAT core for unsatisfiable ones. To perform repair, Kodkod suggests mutations to the data structure such that it meets the Alloy specification. Specifically, given a satisfiable relational formula and the bounds, Kodkod uses a backtracking search to find a satisfying instance. The search space is typically exponential in the number of atoms.

Kodkod allows explicit specification of upper and lower bounds for analysis, which provides partial solutions and restricts the search space. We use this functionality to specify which fields of the state can be mutated by the SAT solver to perform repair. Thus, to *relax* a field in Kodkod means to let the SAT solver suggest different values other than the one present in the faulty post-state, in order to find a satisfiable answer. Relaxing a field, which is a mutation of a field of a specific object, is done through binding a relation to suitable lower and upper bounds. For example, in Figure 1 (a) the dotted edge can be relaxed by setting the lower and upper bounds as shown in Figure 1 (c). Setting both lower and upper bounds to the same set makes it the only answer for that relation, i.e., the set becomes a partial solution for the Kodkod model.

Minimal Unsatisfiable Cores: If Kodkod cannot satisfy the constraints in a model, it produces a minimal unsatisfiable core, which is a subset of the constraints of the model. Given an unsatisfiable CNF formula X, a minimal unsatisfiable sub-formula is a subset of X's clauses that is both unsatisfiable and minimal, which means any subset of it is satisfiable. There could be many independent reasons for a formula's unsatisfiability and hence more than one minimal core. The Recycling Core Extractor algorithm, implemented as the RCE Strategy in Kodkod, returns an unsatisfiable core of specifications written in the Alloy language that is guaranteed to be sound (constraints not included in the core are irrelevant to the unsatisfiability proof) and irreducible (removal of any constraint from the set would make the remaining formula satisfiable).

3 Cobbler Framework

This section describes our history-aware contract-based repair framework.

3.1 Overview

We use class invariants and method post-conditions to detect erroneous executions. Once an error is detected, we utilize two major sources of information about the intended behavior: the specification and the dynamic trace of execution which we obtain through write and read logs. Although the post-condition specifies the expected behavior of the method, there is often a wide range of correct possibilities for a given input since there may be many ways to implement the same specification. Additionally, for a SAT-based repair framework, relaxing all fields of the data structure explodes the search space and is infeasible for real world applications.

We use the program execution to help guide our repair process. In deployed software, the program is expected to contain most of the intended logic. Furthermore, given sufficient pre-deployment testing, there should not be many bugs in the code. By observing the dynamic behavior of a faulty execution, we can substantially reduce the size of the search space and make the repair process more efficient and effective. The

core idea is to focus on fields modified and/or read during the execution. To obtain the execution history, we record write and read actions performed by the program. Our implementation instruments the program, but alternatively the Java Virtual Machine could efficiently provide them [1]. We start by restricting the SAT solver to correcting written fields and values, followed by read fields during the execution, and if the SAT solver has still not found a correction, it utilizes the UNSAT core provided by the previous SAT invocations to identify and mutate faulty fields of the data structure. Hence, our technique handles both errors of *commission* when the programmer writes an incorrect assignment and errors of *omission* when she forgets to update the required fields.

While repair has various applications, it does not suit all types of software systems. For systems that cannot tolerate even slight divergences in the state of the program from the original behavior (e.g, financial systems), it is not advisable to use automatic repair routines unless complete contracts with all the required details are available.

When repair is applicable, this approach has two benefits: (1) it improves the repair performance by reducing the size of the search space, and (2) it reduces the amount of data structure perturbation introduced by the repair process by focusing on fields that a correct method conceivably would modify.

Listing 1.1 shows the repair algorithm in pseudo-code. If an assertion is violated, the repair framework initially only mutates (relaxes) fields in the write log, holding all other data structure fields constant (through providing a partial solution for the SAT solver). It then calls the SAT solver to compute correct values for the relaxed fields. If this step does not yield a structure satisfying the contracts, the next step relaxes the fields in the read and write logs. If it still is unsuccessful, it relaxes fields appearing in the UNSAT core. If the SAT solver finds no solution, there is an inconsistency in the contract itself which the repair framework reports.

```
1 if(!assertContracts()){
2    relaxSAT(writeBarrierLog);
3    if(!assertContracts()){
4       relaxSAT(writeBarrierLog, readBarrierLog);
5       if(!assertContracts()){
6          relaxSAT(unsatCoreFields);
7          if(!assertContracts()){
8             reportModelInconsistency();}}}}
```

Listing 1.1. History-aware contract-based repair using read and write logs and unsatisfiable cores

3.2 Example

Consider a binary search tree example written in Java in Listing 1.2 and its `remove` method. In Cobbler, developers must write a specification in the Alloy relational first order logic language. Listing 1.3 shows the acyclicity and size constraints that describe a correct binary search tree in Alloy. Additional constraints include search relations on the nodes and that the elements are unique. The `repOK` method describes all method-independent constraints. The developer may also express method post-conditions, as shown in the `remove_postcondition` method. This post-condition specifies a correct `remove` with respect to the data structure and the return value from the `remove` method.

```
1  class BinarySearchTree {
2     Node root; int btSize;
3     boolean remove(int x) {
4        if (root == null) return false;
5        else {
6           boolean result;
7           if (root.element == x) {
8              Node auxRoot = new Node();
9              auxRoot.left = root;
10             result = root.remove(x, auxRoot);
11             root = auxRoot.left;
12          } else result = root.remove(x, null);
13          if (result) btSize--;//using uniqueness of elements
14          return result;}}}
15 class Node {
16    Node left, right; int element;
17    boolean remove(int x, Node parent) {
18       if (x < element) {
19          if (left != null) return left.remove(x, this);
20          else return false;
21       } else if (x > element) {
22          if (right != null) return right.remove(x, this);
23          else return false;
24       } else {
25          if (left != null && right != null) {
26             element = right.minNode().element;
27             right.remove(element, this);
28          } else if (parent.left == this) {
29             if (left != null) parent.left = left;
30             else parent.left = right;
31          } else if (parent.right == this) {
32             if (left != null) parent.right = left;
33             //to introduce bug cycle replace with left.right = parent
34             else parent.right = right;}
35          return true;}}
36    Node minNode() {...}}
```

Listing 1.2. A binary search tree implementation in Java

```
1  abstract sig BinarySearchTree {
2     root, root': lone Node,
3     btSize, btSize': one Int}
4  abstract sig Node{
5     left, left', right, right': lone Node,
6     element, element': lone Int}
7  pred repOK(t: BinarySearchTree){ //class invariant
8     //directed acyclicity
9     all n: t.root'.*(left'+right')|n !in n.^(left'+right')
10    //size OK
11    # t.root'.*(left'+right') = int t.btSize'
12    //unique elements
13    ...
14    //search property
15    ...}
16 pred remove_postcondition(This: BinarySearchTree, x: Int, removeResult: (True+
     False)){
17    repOK[This]
18    //correct remove
19    This.root.*(right+left).element - x = This.root'.*(right'+left').element'
20    //correct remove result
21    x in This.root.*(right+left).element <=> removeResult in True}
```

Listing 1.3. A binary search tree specification in Alloy

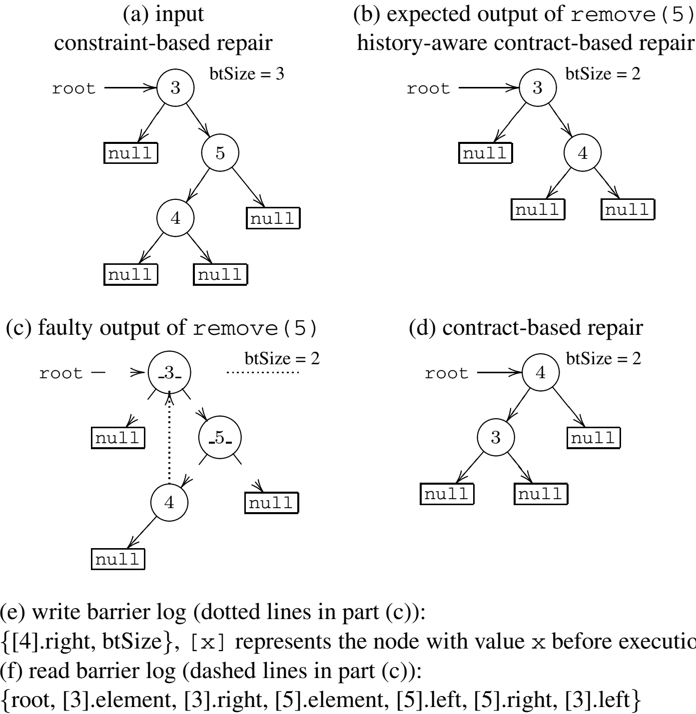


Fig. 2. `cycle` manifested as a faulty output and the repair result

Alloy represents Java classes with signatures (e.g., `sig BinarySearchTree` in Listing 1.3) and field relations with a relational view. The keywords `lone` and `one` for a unary relation denote that the relation may or must not be empty, respectively. Binary relations can be defined as total or partial functions among other options (e.g., `right` is a partial function). We use the syntactic sugar of adding back-tick ('`') to distinguish post-state Alloy relations from pre-state relations. The Alloy `repOK` predicate (`pred`) expresses data structural integrity rules. For instance, the directed acyclicity constraint specifies that for any node reachable from `root` by applying zero or more `left` or `right` pointers, the node cannot reach itself by following one or more `left` or `right` pointers, so it cannot traverse a cycle. `*` and `^` represent "zero or more" and "one or more" applications of a relation. Alloy supports membership, cardinality, and complement, `in`, `#`, and `-` respectively as in the acyclicity, size, and correct remove constraints.

To illustrate our repair process, consider inserting the following bug. **Bug cycle:** in Listing 1.2 line 32, replace the correct statement: `parent.right = left` with the incorrect: `left.right = parent`. For this incorrect implementation, after the method returns, checking the conjunction of `repOK` and the method post-condition indicates that there is an error, triggering the repair process.

To repair the erroneous output of the `cycle` faulty implementation, constraint-based repair methods [4, 8, 10] observe the cycle and remove it from Figure 2(c) to produce Figure 2(a), but fail to remove node `5`. Contract-based repair techniques *without history* [18, 25] may generate Figure 2(d), which although a correct output, is different from what the program would have been generated in the absence of any bugs.

History-aware contract-based repair first invokes the SAT solver and tries to find a solution by only changing the values of the fields which the program writes into during the execution (Figure 2 (e)). These fields are distinguished by dotted lines in the faulty output. In this invocation, it does not find a solution because the program failed to update a field that needs to be modified. Our repair framework next considers changing fields read by the program (Figure 2 (f)) and shown as dashed lines. It invokes SAT to find suitable replacements for the fields written or read by the program. This invocation produces a repaired structure as shown in Figure 2 (b), which is identical to the expected output. Utilizing the barrier logs keeps us from generating Figure 2 (d) since the `left` field of node 4 is not relaxed and is held constant to be null. However, there remains a chance that a field that was not touched at all during the execution needs to be changed. Our repair framework obtains an UNSAT core from the previous SAT invocations. The UNSAT core is the conjunction of contradicting `repOK` and post-condition specifications, which were not satisfiable at the same time. In this example, if we were to proceed to the third SAT call, the UNSAT core would not include, for example, the `correct remove result` post-condition. Therefore, the final invocation of SAT would not relax the `removeResult` field.

3.3 Implementation in Cobbler

Cobbler works as follows. (1) The user provides the Java data structure class and its methods. Cobbler instruments this code with setters and getters to obtain logs of the writes and reads. Cobbler also instruments the program for our experiments to measure the repair time, edit distance and other metrics. (2) Cobbler generates a stub for the `repOK` and method post-conditions for the Java class. Cobbler extracts class-specific relations, types, and properties into the stubs, and the user enhances them with the application specific logic. (3) Cobbler then instruments the program to check the post-conditions and call the repair method when needed. (4) The user executes the Java program inside the Cobbler framework.

Figure 3 shows how the repair framework sits on top of the Java Virtual Machine and executes the Java program. The layers use shared memory to communicate. This design enhances the portability of our framework and makes it independent of JVM and the program. Alternative implementations could implement the framework inside the JVM, which would lower the overhead when programs are correct. When programs need to be repaired, the SAT solving time is orders of magnitude bigger than time saved by merging the repair framework into JVM.

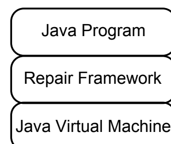


Fig. 3. The relationship between Cobbler, the Java Virtual Machine, and the program

4 Evaluation

The objectives of our evaluation are to empirically validate the hypothesis that using execution history and UNSAT cores improves the efficiency, accuracy, and scalability of contract-based repair with SAT solvers. To this end, we simulated various errors in microbenchmarks and examined two real world applications: Kodkod [22] and ANTLR [2, 16]. Cobbler discovered a previously unreported bug in the `addChild` method of ANTLR version 3.3 that resulted in a cycle in the output Tree. Our repair algorithm fixes this error accurately for a Tree with 300 nodes within 30 seconds.

Throughout the evaluation, we ran each experiment five times and reported the averages. All the experiments used a 2.50GHz Core 2 Duo processor with 4.00GB RAM running Windows 7 and Sun's Java SDK 1.6.0 JVM. All the repair frameworks used their default SAT solvers: Cobbler used MiniSat and MiniSatProver, Tarmeem used DefaultSAT4J, and PBnJ used MiniSat.

4.1 Evaluation Metrics

To evaluate the efficiency of repair, we measured: (1) **logging time:** the overhead due to logging read and write actions; (2) **check time:** the time to detect a contract violation; and (3) **repair time:** the time to search and find a repaired data structure.

To evaluable the accuracy of repair, we measure the *edit distance* between the object graphs of the repaired data structure r, and the expected data structure e that a correct implementation would produce. Note that, r satisfies the method contract but might be different from the expected output. We define edit distance as the minimum number of edge additions/deletions to change a graph to another [19, 25]. We create the correct graphs by a separate correct implementation and then measure the edit distance in set difference operations between two graphs using the relational representation discussed in Section 2. Here $inst_i(R)$ is the instance of relation R in data structure i.

Definition 1. $dist(e, r) = \sum_R (|inst_e(R) - inst_r(R)| + |inst_r(R) - inst_e(R)|)$.

The lower this distance, the closer the repaired data structure is to the expected post-state data structure. We define the similarity percentage between the repaired output r and the expected output e as follows.

Definition 2. $sim(e, r) = (1 - \frac{dist(e,r)}{\sum_R |inst_e(R)|}) \times 100$.

4.2 Subject Programs

We applied Cobbler to (1) the `remove` method of Singly Linked List, (2) the `insert` method of the `Kodkod.util.ints.IntTree` class of the Kodkod solver implementation, and (3) the `deleteChild` and `addChild` methods of `BaseTree` of ANTLR.

Singly-linked list: Linked list is widely used and is a part of libraries such as `java.util.Collection`. The post-condition of the `remove(int value)` method, checks if the method has (1) deleted all nodes with elements equal to the input value, (2) maintained acyclicity, (3) inserted no new nodes, and (4) deleted no other nodes.

Red-black tree of Kodkod: Kodkod [22] is a SAT-based constraint solver for first order logic. It consists of 33,985 lines of Java code in 169 classes. The `IntTree` class with 570 lines of code and 21 methods sits at the core of the Kodkod solver, and is a generic implementation of the red-black tree data structure. Red-black tree comprises complex data structure invariants which include binary search tree invariants: every node has at most two children, key values of the left subtree are smaller and those of the right subtree are greater than the node value, and the tree is acyclic. In addition, constraints are imposed on the color of each node to keep the tree balanced: every node is either red or black, every leaf node is black, no red node has a red parent and all paths from the root to a descendant leaf contain the same number of black nodes. The `insert` method of this class comprises 58 lines of code with 67 branch statements. The post-condition of the `insert(int newKey)` method checks if an element with the new key value has been added without adding or deleting any other elements.

`BaseTree` of ANTLR: We use ANTLR (ANother Tool for Language Recognition) from the DaCapo 2009 benchmark suite, version 9.12 [2, 16]. ANTLR builds language parsers, interpreters, compilers, and translators from grammars. It comprises 29,710 lines of Java code, and has a download rate of about 5,000 per month. Rich tree data structures represent language grammars and are the backbone of this application. The abstract class `BaseTree` is a generic tree implementation. Each tree instance maintains a list of successor children. The `childIndex` represents its position in the list. Each child node is a tree and points back to its parent. Every node may contain a token field that represents the payload of the node. Based on the documentation and the program logic, we derived invariants for the `BaseTree` data structure such as acyclicity through children references, accurate parent-child relationships, and correct values for child indices. The `addChild(Tree node)` and `deleteChild(int childIndex)` methods are the main functions used to build and manipulate all tree structures in ANTLR. The respective post-conditions check that nodes are added or deleted without any unwarranted perturbations to the other nodes.

4.3 Errors

Table 1 enumerates all the inserted faults and, for ANTLR, a detected error. It explains the errors and displays the violated constraints. The accuracy and performance of the repair algorithm depends on which and how many fields are relaxed in each step, and the number of calls to the solver. The data structure size, size of the log, and size of violated constraint formula influence repair accuracy and efficiency. We explore these parameters with a range of errors that violate different constraints and appear in different program statements, such as incorrect field assignments, incorrect branch conditions, and errors of omission. The last column in the table indicates if the field(s) that needs to be corrected appear in the write barrier log (WB), read barrier log (RB), or neither (ALL fields).

The program logic and thus which fields Cobbler logs depends on the input structures. Faults five and six of the red-black tree `insert` method execute the same faulty code versions as that of three and four, but with a different data structure. The program writes and reads different fields on the first and second inputs and Cobbler repairs the outputs by relaxing read and written fields respectively.

Table 1. The injected faults and ANTLR addChild() fault

Method		Fault description	Violates	Error in
SLL remove	Err 1	Sets the header to null	Correct remove, Size	WB
	Err 2	Fails to update the size	Size	ALL fields
	Err 3	Deletes a node with a non-matching element	Correct remove, Size	WB
	Err 4	Introduces a cycle after performing correct remove	Acyclicity	WB
	Err 5	Breaks the list to retain only the first two nodes	Correct remove, Size	WB
	Err 6	Deletes the matching element but adds it again	Correct remove	WB
	Err 7	Fails to remove the element and updates the size incorrectly	Correct remove, Size	WB
RBT insert	Err 1	Creates a cycle of length one	Acyclicity	WB
	Err 2	Sets the color of a node to black instead of red	Color constraints	WB
	Err 3	Adds the new element as right child instead of left	Key constraints	RB
	Err 4	Violates key constraints due to a branch condition error	Key constraints	RB
	Err 5	Same as Err 3 with a different input	Key constraints	WB
	Err 6	Same as Err 4 with a different input	Key constraints	WB
	Err 7	Skips balancing of the tree after insertion	Color constraints	ALL fields
ANTLR deleteChild	Err 1	Skips deletion of the appropriate child	Correct Remove	RB
	Err 2	Skips updating children indices after deletion	Child Index constraints	ALL fields
	Err 3	Sets a wrong child index due to an incorrect branch condition in a loop	Child Index constraints	RB
	Err 4	Sets a node as its own parent	Acyclicity	WB
ANTLR addChild		Adds a node to itself as a child	Acyclicity, Child Index	WB

4.4 Subject Tools

We compare Cobbler to Juzi repair framework, which only uses structural constraints, and to Tarmeem and PBnJ, two repair frameworks that consider post-conditions too.

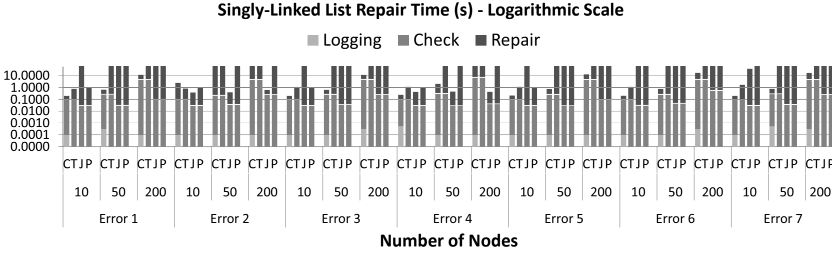
Juzi's assertion-based repair automatically corrects data structure violations in Java programs [5]. Upon detecting a constraint violation, Juzi searches for a repair solution based on the data structure traversal encoded in `repOK` [3]. Juzi further boosts its performance with symbolic execution. Since Juzi does not use a SAT solver, it is generally faster than SAT-based approaches. Juzi however does not consider method post-conditions, which causes it to miss errors that result in well-formed output. Even if `repOK` is violated, without the post-condition, Juzi cannot accurately correct the structure with respect to the contracts as discussed in Section 3.2. To compare Juzi and Cobbler, we manually implemented a check for the post-condition in Juzi by recording the method pre-state and the desired data structure specific post-state.

Our previous work, Tarmeem, uses Alloy contracts and a SAT solver [25]. Tarmeem repairs faulty post-states using automated and user-guided techniques, such as iterative relaxation of relations and error localization in predicates to improve the efficiency of repair. We experimented with all four of Tarmeem's heuristics and picked the best.

Samimi et al. implement a similar technique in PBnJ that executes method specifications when methods fail to produce a correct data structure [18]. They express invariants and specifications in a declarative first order relational logic. Translating them into Java methods and then invoking the methods implements program logic declaratively. This program synthesis approach leverages constraint solving technology.

4.5 Results

Figure 4 compares the performance and accuracy of repair of Cobbler, Tarmeem, Juzi, and PBnJ on the singly-linked list microbenchmark. Logging, check, and repair times



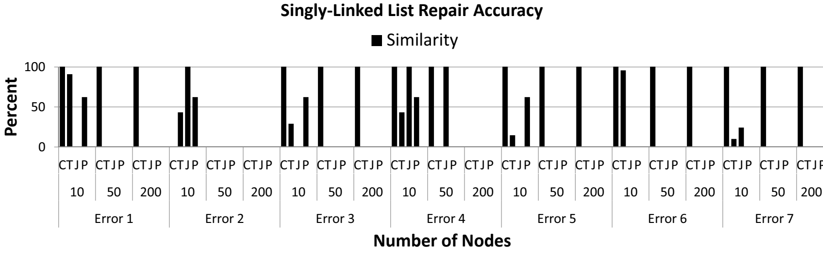


Fig. 4. Performance and accuracy: repairing singly-linked lists with Cobbler (C), Tarmeem (T), an enhanced version of Juzi (J), and PBnJ (P)

are accumulated into a single bar on a logarithmic scale. Logging time is only applicable to Cobbler and is negligible. Tarmeem and Cobbler have the same check time since they both use Kodkod evaluation (not SAT solving) to perform checks after methods execute. Juzi executes `repOK` and PBnJ translates specifications to Java assertions, which more efficiently check the data structure. Cobbler's overhead on an error-free execution includes both logging and check times. Using the approach of PBnJ to translate specifications to assertions could reduce the check time and the total overhead. We timeout after 60 seconds and report zero for accuracy upon a timeout.

Cobbler is substantially faster than all the other tools on five of the seven errors, despite the fact that Tarmeem and PBnJ receive specific user annotations to guide the repair process and Juzi performs symbolic execution. Error two skips a required update to size. Since the size field is not read or written, Cobbler does not correct it until the third call to the SAT solver, which causes its time to exceed the other repair schemes. Error four introduces a cycle. Juzi is tailored for such errors: it satisfies the constraint by breaking cycles quickly and performs better than Cobbler in this case.

Cobbler, except for one case, always produces the most accurate data structure among the four. When Cobbler does not time out, it achieves exactly the same output as expected. The edit distance between the result of a correct implementation and the repaired data structure is zero. This comparison is solely for evaluation, since in the wild, the system would not know the correct implementation.

Because Juzi solely relies on the `repOK` method instead of checking method postconditions, it does not find error six at all. Moreover, Juzi cannot access out of scope nodes that are not reachable from the header. Since Juzi does not consider the execution

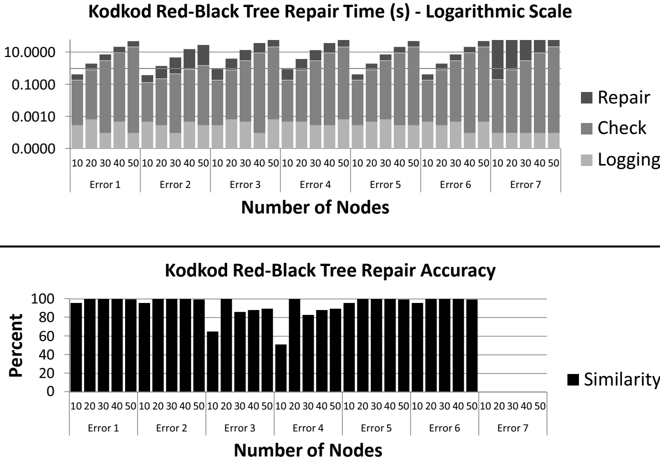


Fig. 5. Cobbler performance and accuracy: repairing Kodkod red-black trees

history, it first explores all the correct data structures nearby, but there is no guarantee that the expected output is close to the faulty one. We could enhance Juzi to work with post-conditions, as we did for evaluation of accuracy, but the original Juzi did not perform any repairs with respect to the post-conditions.

Tarmeem is not very accurate because when it invokes the SAT solver, it relaxes *all* tuples of a relation together, causing unnecessary changes. Cobbler significantly improves efficiency and accuracy over Tarmeem, especially for errors which involve incorrectly updated fields.

PBnJ's performance is similar to Tarmeem at best. The reason is that it always ignores the current faulty state and utilizes SAT to regenerate an acceptable output from scratch. It is however more accurate than Tarmeem in some cases.

Figure 5 shows the performance and accuracy of Cobbler on a faulty Kodkod red-black tree insert method. Figure 6 depicts the results of experimenting with ANTLR. We do not include the other frameworks here for brevity. Juzi always fails to repair correctly when a contract requires the addition of a node and the node is not present, because Juzi only uses those nodes currently accessible from the faulty data structure. When it does not timeout, Cobbler is very accurate on these real world applications.

The results show that the read and write field logs improve the scalability and efficiency of repair. Cobbler repairs linked lists with up to 200 nodes within 20 seconds. It performs well even on more complex data structures. For the red-black tree `remove` method, it repairs up to 50 nodes within 40 seconds and for the `deleteChild` method of ANTLR `BaseTree`, it repairs 40 nodes within 30 seconds. The size of the logs is proportional to the number of reads and writes to the data structure and was usually a few hundred bytes with a maximum of 900 bytes for error four of ANTLR.

For errors that cannot be fixed by relaxing only written and read fields, such as error two of linked list, error seven of red-black tree insert, and error two of ANTLR `deleteChild` (see Table 1), Cobbler uses the UNSAT core to identify which fields

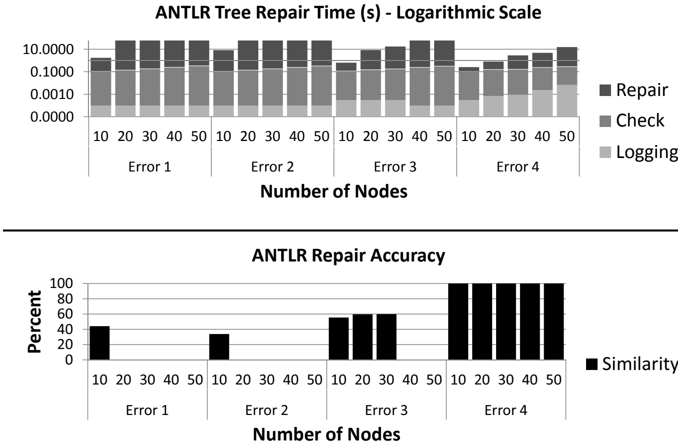


Fig. 6. Cobbler performance and accuracy: repairing ANTLR trees

to modify, and performs better than the other SAT-based tools. These cases however are challenging for Cobbler, because despite barrier logs that indicate fields of specific objects, UNSAT cores identify all fields with the same name as potentially faulty.

4.6 ANTLR `BaseTree addChild`

The public method `addChild` adds child node trees to an ANTLR `BaseTree` object. When the input tree does not have a payload (`isNil`), the method adds the children of the input tree to the children list of the current tree, otherwise, it adds the input tree itself to the children list. In the `addChild` method (v3.3), when the input tree does not have any payload, a check ensures that the current tree is not being added to itself. However, such a check is not performed for input trees with payloads. Hence, when the current tree has a payload, it may be added as a child of itself. Similarly, any tree with a payload which is already an existing child of the current tree may be added as a child again. We generated inputs that caused invariant (such as acyclicity and ascending child indices) violations. Cobbler repairs the Tree structure and restores it back to its pre-state, which is correct. This state would be the output of `addChild` if it had been implemented correctly. Cobbler repairs a tree with 300 nodes within 30 seconds.

5 Related Work

Dynamic repair aims to counteract faults at runtime and prolong system uptime. File system utilities such as fsck [6] and chkdsk [13], database check-pointing and roll-back techniques are application-specific repair routines that monitor and correct system state at runtime. DIRA [20] extends database repair with post-conditions to detect buffer overflow attacks and fix damaged data structures. Clearview [17] and Exterminator [15]

also aid in repairing memory errors at runtime, but none of these techniques are suitable for repairing general purpose complex data structures. On the other hand, some commercially developed systems, such as the IBM MVS operating system [14] and the Lucent 5ESS telephone switch [7], have dedicated routines for monitoring and maintaining properties of their key data structures. These systems are tailor-made for their system structures and cannot be generalized as data structure repair tools.

Demsky and Rinard [4] pioneered constraint-based repair of data structures. Developers provide declarative constraints. The system translates the constraints into disjunctive normal form and repair solves them using an ad hoc search. The Juzi repair technique (described in Section 4.4) detects errors using user-defined `repOK` methods [5]. As we discussed and showed in Section 4.5, the accuracy and efficiency of Juzi suffer for errors that omit nodes and because the repair does not consider method semantics at the entry and exit. Recent improvements include Dynamic Symbolic Data Structure (DSDS) repair which builds a symbolic representation of fields and objects along the `repOK` executed path [8]. Whenever a predicate fails, DSDS solves the conjunction of its negation with the other path conditions. This direct generation of a satisfying result loses accuracy because it is irrespective of the exact location of corrupted object references or fields. A post-condition Java method predicate could be asserted along with the `repOK` to solve this problem. But as the size and complexity of properties and size of the data structure increase such techniques will not scale well.

Tarmeem [25] and PBnJ [18] (both explained in Section 4.4) overcome this limitation by using individual method pre- and post-conditions. As Section 4.5 showed, Tarmeem improves accuracy by tailoring repair to semantics, but is inefficient. PBnJ is not very efficient either, because it ignores both the faulty post-state and implementation. To improve the efficiency of PBnJ, programmers may bound the number of objects and limit changed fields, but for repairing unpredictable code errors, it does not seem feasible. Our approach instead automatically utilizes the faulty data structure and the code that produced it to prune the state space and guide repair to yield a satisfying instance as close as possible to the intended method output.

Our technique is related to, but differs substantially from, automated debugging and repair for use during testing, which focus on how to change the code rather than dynamically changing the heap [12,21,23,24]. However, as Malik et al. propose, dynamic repair actions could translate into program statements [11].

6 Conclusions

This paper introduced the idea of using program execution history for efficient and accurate contract-based data structure repair. We utilize program traces, specifically reads and writes of key fields, to direct repair of erroneous program states. Moreover, we use unsatisfiable cores provided by SAT solvers when we cannot repair the data structure by changing only read and written fields. We implemented this approach in Cobbler. Compared with previous repair techniques, our experimental results show Cobbler provides significant speedups and better accuracy, and finds and repairs a previously undetected bug in the widely used open-source ANTLR program. A promising future research avenue is to abstract concrete successful repair actions and use them to prioritize future repair actions, thus to avoid a costly search and make repair even more practical.

Acknowledgments. We thank the anonymous reviewers for their comments. This work was funded in part by the NSF under Grant Nos. CCF-0845628, IIS-0438967, CCF-1018271, CCF-0811524, and SHF-0910818, and AFOSR grant FA9550-09-1-0351.

References

1. Blackburn, S.M., Hosking, A.: Barriers: Friend or foe? In: ISMM (2004)
2. Blackburn, S.M., et al.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In: OOPSLA (2006)
3. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: ISSTA (2002)
4. Demsky, B., Rinard, M.: Automatic detection and repair of errors in data structures. In: OOPSLA (2003)
5. Elkarablieh, B., Garcia, I., Suen, Y.L., Khurshid, S.: Assertion-based repair of complex data structures. In: ASE (2007)
6. Ext2 fsck. manual page, `http://e2fsprogs.sourceforge.net`
7. Haugk, G., Lax, F., Royer, R., Williams, J.: The 5ESS(TM) switching system: Maintenance capabilities. AT&T Technical Journal 64(6 part 2), 1385–1416 (1985)
8. Hussain, I., Csallner, C.: Dynamic symbolic data structure repair. In: ICSE (2010)
9. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
10. Khurshid, S., García, I., Suen, Y.L.: Repairing Structurally Complex Data. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 123–138. Springer, Heidelberg (2005)
11. Malik, M.Z., Ghori, K., Elkarablieh, B., Khurshid, S.: A case for automated debugging using data structure repair. In: ASE (2009)
12. Mayer, W., Stumptner, M.: Evaluating models for Model-Based debugging. In: ASE (2008)
13. Microsoft. chkdsk manual page, `http://support.microsoft.com/kb/315265`
14. Mourad, S., Andrews, D.: On the reliability of the IBM MVS/XA operating system. IEEE Transactions on Software Engineering 13(10), 1135–1139 (1987)
15. Novark, G., Berger, E.D., Zorn, B.G.: Exterminator: automatically correcting memory errors with high probability. In: PLDI (2007)
16. Parr, T., Bovet, J.: Antlr parser generator home page, `http://www.antlr.org`
17. Perkins, J., et al.: Automatically patching errors in deployed software. In: SOSP (2009)
18. Samimi, H., Aung, E.D., Millstein, T.: Falling Back on Executable Specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
19. Sanfeliu, A., Fu, K.-S.: Distance measure between attributed relational graphs for pattern recognition. IEEE Trans. Systems, Man and Cybernetics 13(3), 353–362 (1983)
20. Smirnov, A., Chiueh, T.-c.: DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In: NDSS (2005)
21. Staber, S., Jobstmann, B., Bloem, R.: Finding and Fixing Faults. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 35–49. Springer, Heidelberg (2005)
22. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
23. Wei, Y., et al.: Automated fixing of programs with contracts. In: ISSTA (2010)
24. Weimer, W.: Patches as better bug reports. In: GPCE (2006)
25. Zaeem, R. Nokhbeh, Khurshid, S.: Contract-Based Data Structure Repair Using Alloy. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 577–598. Springer, Heidelberg (2010)