# Specification-Based Program Repair Using SAT

Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid

The University of Texas at Austin

**Abstract.** Removing bugs in programs – even when location of faulty statements is known – is tedious and error-prone, particularly because of the increased likelihood of introducing new bugs as a result of fixing known bugs. We present an automated approach for generating likely bug fixes using behavioral specifications. Our key insight is to replace a faulty statement that has deterministic behavior with one that has nondeterministic behavior, and to use the specification constraints to prune the ensuing nondeterminism and repair the faulty statement. As an enabling technology, we use the SAT-based Alloy tool-set to describe specification constraints as well as for solving them. Initial experiments show the effectiveness of our approach in repairing programs that manipulate structurally complex data. We believe specification-based automated debugging using SAT holds much promise.

## 1 Introduction

The process of debugging, which requires (1) *fault localization*, i.e., finding the location of faults, and (2) *program repair*, i.e., fixing the faults by developing correct statements and removing faulty ones, is notoriously hard and time consuming, and automation can significantly reduce the cost of debugging. Traditional research on automated debugging has largely focused on fault localization [3, 10, 20], whereas program repair has largely been manual. The last few years have seen the development of a variety of exciting approaches to automate program repair, e.g., using game theory [9], model checking [18], data structure repair [11] and evolutionary algorithms [19]. However, previous approaches do not handle programs that manipulate structurally complex data that pervade modern software, e.g., perform destructive updates on program heap subject to rich structural invariants.

This paper presents a novel approach that uses rich behavioral specifications to automate program repair using off-the-shelf SAT technology. Our key insight is to transform a faulty program into a *nondeterministic* program and to use SAT to prune the nondeterminism in the ensuing program to transform it into a correct program with respect to the given specification. The key novelty of our work is the support for rich behavioral specifications, which precisely specify expected behavior, e.g., the `remove` method of a binary search tree only removes the given key from the input tree and does not introduce spurious key into the tree, as well as preserves acyclicity of the tree and the other class invariants, and the use of these specifications in pruning the state space for efficient generation of program statements.

We present a framework that embodies our approach and provides repair of Java programs using specifications written in the Alloy specification language [6]. Alloy

P.A. Abdulla and K.R.M. Leino (Eds.): TACAS 2011, LNCS 6605, pp. 173–188, 2011.
© Springer-Verlag Berlin Heidelberg 2011

is a relational, first-order language that is supported by a SAT-based tool-set, which provides fully automatic analysis of Alloy formulas. The Alloy tool-set is the core enabling technology of a variety of systematic, bounded verification techniques, such as TestEra [12] for bounded exhaustive testing, and JAlloy [7] and JForge [4] for scope-bounded static checking. Our framework leverages JForge to find a smallest fault revealing input, which yields an output structure that violates the post-condition. Given such a counterexample and a candidate list of faulty statements, we parameterize each statement with variables that take a nondeterministic value from the domain of their respective types. A conjunction of the fixed pre-state, nondeterministic code, and post-condition is solved using SAT to prune the nondeterminism. The solution generated by SAT is abstracted to a list of program expressions, which are then iteratively filtered using bounded verification.

This paper makes the following contributions:

- **Alloy (SAT) for program repair.** We leverage the SAT-based Alloy tool-set to repair programs that operate on structurally complex data.
- **Rich behavioral specifications for program repair.** Our support for rich behavioral specifications for synthesizing program statements enables precise repair for complex programs.
- **Framework.** We present an automated specification-based framework for program repair using SAT, and algorithms that embody the framework.
- **Evaluation.** We use a textbook data structure and an application to evaluate the effectiveness and efficiency of our approach. Experimental results show our approach holds much promise for efficient and accurate program repair.

## 2 Basic Principle

The basic concept of bounded verification of Java programs using SAT is based on the relational model of the Java Heap. First proposed in the JAlloy technique [7], this involves encoding the data, the data-flow and control-flow of a program in relational logic. Every user-defined class and in-built type is represented as a set or a domain containing a bounded number of atoms(objects). A field of a class is encoded as a binary function that maps from the class to the type of the field. Local variables and arguments are encoded as singleton scalars. Data-flow is encoded as relational operations on the sets and relations. For instance, a field deference `x.f`, where `x` is an object of class `A` and `f` is a field of type `B`, is encoded as a relational join of the scalar 'x' and the binary relation f : A→B. Encoding control-flow is based on the computation graph obtained by unrolling the loops in the control flow graph a specified number of times. The control flow from one statement to the next is viewed as a relational implication, while at branch statements, the two branch edges are viewed as relational disjunctions. The entire code of a method can thus be represented as a formula in first-order relational logic such as Alloy [6], P($s_t$,$s'_t$) relating the output state $s'_t$ to its input $s_t$. The conjunction of the pre-condition, the formula representing the code and the negation of the post-condition specification of the method is fed into a relational engine such as Kodkod [16]. The formula is translated into boolean logic and off-the-shelf SAT solvers

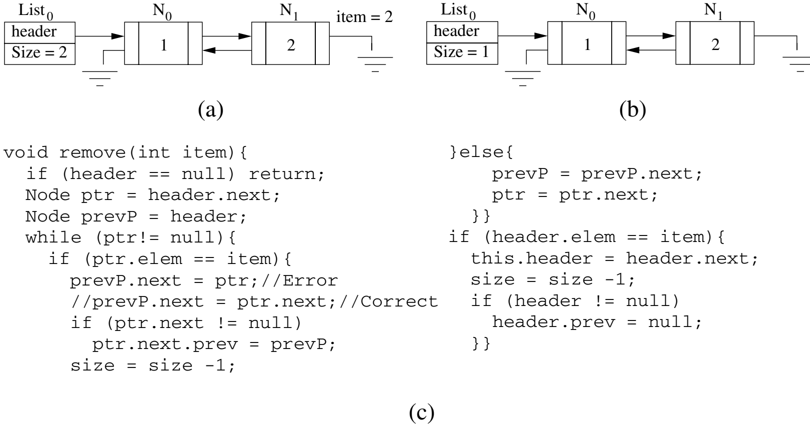


**Fig. 1.** Faulty remove from Doubly Linked List(DLL) (a) Pre-state with input argument 2 (b) Post-state, size updated but node not removed. (c) Faulty DLL `remove` method.

are used to solve it. A solution represents a counter-example to the correctness specification, showing the presence of an input satisfying the pre-condition, tracing a valid path through the code and producing an output that does not satisfy the post-condition. Our technique builds on this idea to employ SAT to automatically perform alterations to program statements which would correct program errors leading to the specification violations. The technique can be best explained using a simple example.

The code snippet in Fig. 1c shows a faulty `remove` method of a doubly linked list data structure. The data structure is made up of nodes, each comprising of a `next` pointer, pointing to the subsequent node in the list, a `prev` pointer pointing to the previous node and an integer element. The `header` points to the first node in the list and `size` keeps a count of the number of nodes. The `remove` method removes the node which has its element value equal to input value. It uses two local pointers, `ptr` and `prevP`, to traverse the list and bypass the relevant node by setting the `next` of `prevP` to the next of `ptr`. A typical correctness specification (post-condition) for this method would check that the specified value has been removed from the list in addition to ensuring that the invariants such as acyclicity of the list are maintained. In the erroneous version, `prevP.next` is assigned to `ptr` instead of `ptr.next`. A scope bounded verification technique ( [4]) can verify this method against the post-condition (specified in Alloy) to yield the smallest input structure which exposes the error. Figures 1a and 1b show the fault revealing input and the erroneous output. The pre-state is a doubly linked list with two nodes ($List_0$ is an instance of a list from the $List$ domain, $N_0$ and $N_1$ are instances from the $Node$ domain). In the corresponding erroneous post-state, node $N_1$ with element value 2 is still present in the list since the next pointer of $N_0$ points to $N_1$ rather than $null$. Let us assume the presence of a fault localization technique which identifies the statement `prevP.next = ptr` as being faulty. Our aim is to correct this assignment such that for the given input in the pre-state, the relational constraints of the path through the code yield an output that satisfies the post-condition. To accomplish this, we replace the operands of the assignment operator with new variables which can take any value from the `Node` domain or can be

$null$. The statement would get altered to $V_{lhs}$`.next = `$V_{rhs}$, where $V_{lhs}$ and $V_{rhs}$ are the newly introduced variables. We then use SAT to find a solution for the conjunction of the new formula corresponding to the altered code and the post-condition specification. The values of the relations in the pre-state are fixed to correspond to the fault revealing input as follows, (`header=(`$List_0$`,`$N_0$`)`, `next=(`$N_0$`,`$N_1$`)`, `prev=(`$N_1$`,`$N_0$`)`, `elem=(`$N_0$`,1)`, `(`$N_1$`,2)`). The solver searches for suitable valuations to the new variables such that an output is produced that satisfies the post-condition for the given fixed pre-state. In our example, these would be $V_{lhs} = N_0$ and $V_{rhs} = null$. These concrete state values are then abstracted to programming language expressions. For instance, in the example, the value of the local variables before the erroneous statement would be, `ptr = `$N_1$,`prevP = `$N_0$ and `t = `$List_0$. The expressions yielding the value for $V_{rhs} = null$ would be `prevP.prev`, `prevP.next.next`, `ptr.next`, `ptr.prev.prev`, `t.header.prev`, `t.header.next.next`. Similarly, a list of possible expressions can be arrived at for $V_{lhs} = N_0$. Each of these expressions yield an altered program statement yielding correct output for the specific fault revealing input. The altered program is then validated for all inputs using scope bounded verification to filter out wrong candidates. In our example, counter-examples would get thrown when any of the following expressions `prevP.prev`, `ptr.prev.prev`, `t.header.prev`, and `t.header.next.next`(for scope greater than 2) are used on the right hand side of the assignment. When no counter-example is obtained, it indicates that the program statement is correct for all inputs within the bounds specified. In this example, `prevP.next = prevP.next.next` and `prevP.next = ptr.next` would emerge as the correct statements satisfying all inputs.

## 3 Our Framework

Fig. 2 gives an overview of our framework for program repair. We assume the presence of a verification module indicating the presence of faults in the program. In our implementation, we have employed bounded verification (Forge framework [4]) which takes in a Java method, bounds on the input size(scope) and number of times to unroll loops and inline recursive calls(unrolls) and generates the smallest fault revealing input($s_t$) within the given scope that satisfies the pre-condition of the method and yields an output($s_t'$) that violates the post-condition. A trace, comprising of the code statements in the path traversed for the given input, is also produced. We also assume the presence of a fault localization scheme, which yields a minimal list of possibly faulty statements. Please note that the technique works on the Control Flow Graph(CFG) representation of the program and the relational model view of the state.

Given a counter-example from the verification module and a list of suspicious statements $S_1, \ldots, S_m$, the first step performed by the repair module is to parameterize each of these statements. The operands in the statement are replaced with new variables. For instance, consider an assignment statement of the form, $x.\{f_1.f_2.\cdots.f_{n-1}\}.f_n = y$. The presence of a commission error locally in this statement indicates that either the source variable $x$, one or more of the subsequently de-referenced fields $f_1, f_2, \ldots, f_{n-1}$ or the target variable $y$ have been specified wrongly. If the field being updated, $f_n$, has

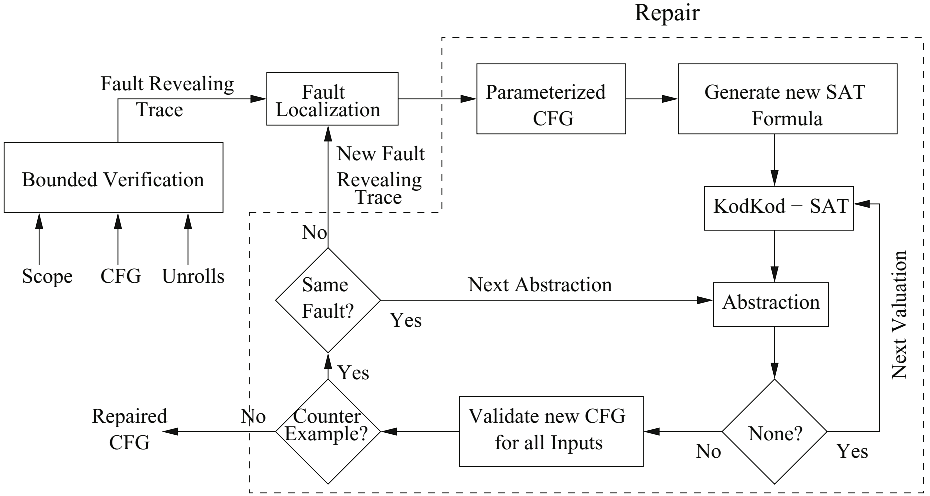


**Fig. 2.** Overview of our repair framework

been specified erroneously, it indicates that the statement to update the correct field (say $f'_n$) has been omitted. Handling of omission errors requires additional user input and is explained later in this section. The altered statement would be $V_{lhs}.f_n = V_{rhs}$, where $V_{lhs}$ and $V_{rhs}$ can be $null$ or can take any value from the domains $A$ and $B$ respectively, if $f_n$ is a relation mapping type $A$ to $B$. Similarly a branch statement such as $x > y$, would be altered to $V_{lhs} > V_{rhs}$ and a variable update $y = x$ to $y = V_{rhs}$. Please note that we attempt to correct a statement by changing only the values read or written by it and not by altering the operators or constant values specified in the statement. We hypothesize that such syntactic and semantic errors can be caught by the programmer during his initial unit testing phase itself, whereas detection of errors relating to the state depends on the input being used. This may get missed during manual review or testing and a bounded verification technique which systematically checks for all possible input structures is adept in detecting such errors. However, if by altering the placement of the operands, the semantics of a wrongly specified operator can be made equivalent to the correct operator, then we can correct such errors. For instance, if $x < y$ has been wrongly specified as $x > y$, the corrected statement would be $y > x$. A new set of variables would thus get defined for each statement in suspect list, $\{V^{S_i}_{lhs}, V^{S_i}_{rhs}\}, \forall i \in \{1, \ldots, m\}$.

New constraints for the code with the newly introduced variables are generated in the next step. The input is fixed to the fault revealing structure by binding the pre-state relations exactly to the values in $s_t$ when supplying the formula to the Kodkod engine. A formula made up of the conjunction of the fixed pre-state, the new code formula and the post-condition, $pre\text{-}state \wedge\ code\text{-}constraints \wedge post\text{-}condition$, is fed to the SAT solver. The solver looks for suitable valuations to the new variables such that a valid trace through the code is produced for the specified input($s_t$) that yields an output($s''_t$) that satisfies the post-condition. If $V^{S_i}_{lhs}$ is a variable of type `Node`, then a

possible solution could bind it to any concrete state value in its domain, $V_{lhs}^{S_i} = n, \forall n \in \{N_0, \ldots, N_{scope-1}\}$. Hence every new variable in every statement in the suspect list would be bound to a concrete state value as follows, $V_{lhs}^{S_i} = C_{lhs}^{S_i}, V_{rhs}^{S_i} = C_{rhs}^{S_i}, \forall i \in \{1, \ldots, m\}$, where $C_{lhs}^{S_i}$ and $C_{rhs}^{S_i}$ are the state values are every statement.

The ensuing abstraction module attempts to map every concrete valuation to local variables or expressions that contain that value at that particular program point. The expressions are constructed by applying iterative field de-references to local variables. For example, if `t` is a local variable containing an instance of a `Tree` and the `left` pointer of its `root` node contains the required concrete value, then `t.root.left` would be the expression constructed. We ensure that we do not run into loops by hitting the same object repeatedly. Stating it formally, if for variable $v$, and fields $f_1, \ldots, f_k$, $v.f_1.\cdots.f_k$ evaluates to $v$, then for any expression $e$ that evaluates to $v$, the generated expression will not take the form $e.f_1.\cdots.f_k.e'$ , rather it will take the form $v.e'$. Please note that when a parameterized statement occurs inside a loop there would be several occurrences of it in the unrolled computation graph. Each occurrence could have different state valuations, however, their abstractions should yield the same expression. Hence we only consider the last occurrence to perform the abstraction. A new CFG is generated with the altered statements. In the event that no abstractions can be produced for a particular concrete valuation, we invoke SAT again to check if there are alternate correct valuations for the given input.

Please note that if SAT is unable to find suitable valuations that yield a satisfying solution for the given input or none of the valuations produced by SAT can be abstracted, it indicates that the fault cannot be corrected by altering the statements in the suspect list. The reason for this could either be that some statements had been omitted in the original code or that the initial list of suspicious statements is inaccurate. At this stage, manual inspection is required to determine the actual reason. Omission errors could be corrected by the tool using "templates" of the missing statements inserted by the user at appropriate locations. In case of the suspect list being incomplete, a feedback could be provided to the localization module to alter the set of possibly faulty statements.

In the validation phase, we use scope bounded checking to systematically ensure that all inputs covering the altered statements yield outputs satisfying the post-condition. If a counter-example is detected, the new fault revealing input and the corresponding trace, along with the altered CFG are fed back into the fault localization module. Based on the faults causing the new counter-example, an altered or the same set of statements would get produced and the process is repeated until no counter-example is detected in the validation phase. There could be faults in the code not detected during the initial verification or during the above mentioned validation process. These may be present in a portion of code not covered by the traces of the previously detected faults. Hence as a final step, we verify that the CFG is correct for all inputs covering the entire code. Thus, the repair process iteratively corrects all faults in the code. In case, all the faulty statements can be guaranteed to be present in the suspect list, the entire CFG can be checked whenever an abstraction is generated. When a counter-example is detected, the subsequent abstractions for the same set of statements could be systematically checked until one that satisfies all inputs is produced.

```
bool insert(Tree t, int k){
  Node y = null;
  Node x = t.root;
  while (x != null)
  //while(x.left != null) (4c)
  { y = x;
    //y = t.root; (3b)(8)(9)
    if (k < x.key)
    //if(k < t.root.key) (9)(13)
      x = x.left;
      //x = x.right; (3a)
      //x = x.left.right (13)
    else
    { if (k > x.key)
      //if (k < x.key)     (4a)
      //if(k > t.root.key) (4b)
        x = x.right;
      else
        return false;}}
  x = new Node();
  x.key = k;
  if (y == null)
  //if(x != null) (10)
    t.root = x;
  else
  { if (k < y.key)
    //if(k > y.key) (2a)
    //if(k < x.key) (2b)(5)(11)(12)
      y.left = x;
      //y.left = y; (6)(10)(11)
    else
      y.right = x;}
      //y.right = y;}(10)(12)
  x.parent = y;
  //x.parent = x; (1)(7)(8)
  //y.parent = x; (5)(6)(10)
  /*x.parent = y;*/ //Omission Err(14)
  t.size += 1;
  return true;}
```

(a)

```
void addChild(Tree t){
  if ( t == null} return;
  BaseTree childTree = (BaseTree)t;
  if (childTree.isNil()) {
    if (this.children != null &&
        this.children == childTree.children)
      throw new RuntimeException("...");
    if (childTree.children != null) {
      int n = childTree.children.size();
      for (int i = 0; i < n; i++) {
      //for (int i = 0,j = 0; i < n; (i = j + 1)) { (1)
        Tree c = childTree.children.get(i);//list get
        this.children.add(c);//list add
        c.setParent(this);
        //j = i + 1; (1)}}
    else this.children = childTree.children;}
  else {
    if (children == null)
    //if (childTree.children == null) (2)
        children = createChildrenList();
    children.add(t); //list add
    childTree.setParent(this);
    //childTree.setParent(childTree); (2)}}
```

(b)

**Fig. 3.** Code Snippets of (a) `BST.insert` (b) ANTLR `BaseTree.addChild` with error numbers. Repaired version produced in the CFG form, manually mapped back to source code.

# 4 Evaluation

This section presents two case studies on (i) Binary Search Tree `insert` method and (ii) `addChild` method of the ANTLR application [13]. The aim of the first study was to simulate an exhaustive list of error scenarios and measure the ability of the technique to repair faulty statements accurately and efficiently. The second evaluation focuses on studying the efficacy of the technique to scale to real world program sizes.

## 4.1 Candidates

**Binary Search Tree (BST)** is a commonly used text-book data structure. BST has complex structural invariants, making it non-trivial to perform correct repair actions to programs that manipulate its structure. The tree is made up of nodes, each comprising of an integer `key`, `left`, `right` and a `parent` pointer. Every tree has a `root` node which has no `parent` and a `size` field which stores the number of nodes reachable from the `root`. The data structure invariants include uniqueness of each element, acyclicity with

respect to `left`,`right` and `parent` pointers, size and search constraints (`key` value of every node needs to be greater than the `key` values of all nodes in its `left` sub-tree and lesser than all keys in its `right` sub-tree). `Insert(int item)` is one of the most important methods of the class, which adds a node to the tree based on the input item value. The post-condition specification checks if the functionality of the method is met in addition to ensuring that the data structure invariants are preserved. It has considerable structural and semantic complexity, performing destructive updates to all fields of the tree class and comprising of 20 lines of code (4 branch statements and 1 loop).

**ANother Tool for Language Recognition (ANTLR) [13]**, a part of the DaCapo benchmark [2], is a tool used for building recognizers, interpreters, compilers and translators from grammars. Rich tree data structures form the backbone of this application, which can be major sources of errors when manipulated incorrectly. Hence ANTLR is an excellent case study for repair on data structure programs. `BaseTree` class implements a generic tree structure to parse input grammars. Each tree instance maintains a list of children as its successors. Each child node is in turn a tree and has a pointer to its `parent` and a `childIndex` representing its position in the children list. Every node may also contain a `token` field which 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, accurate parent-child relations and correct value of child indices. The `addChild(Tree node)` is the main method used to build all tree structures in ANTLR. The respective post-conditions check that the provided input has been added without any unwarranted perturbations to the tree structure.

### 4.2 Metrics

The efficiency of the technique was measured by the total time taken to repair a program starting from the time a counter-example and suspect list of statements were fed into the repair module. This included the time to correct all faults (commission) in the code satisfying all inputs within the supplied scope. Since the time taken by the SAT solver to systematically search for correct valuations and validate repair suggestions is a major factor adding to the repair time, we also measured the number of calls to the SAT solver. The repaired statements were manually verified for accuracy. They were considered to be correct if they were semantically similar to the statements in the correct implementation of the respective algorithms.

Our repair technique is implemented on top of the Forge framework [4]. The pre, post-conditions and CFG of the methods were encoded in the Forge Intermediate Representation(FIR). Scope bounded verification using Forge was used to automatically detect code faults. The suspect list of possibly faulty statements was generated manually. The input scope and number of loop unrolling were manually fixed to the maximum values required to produce repaired statements that would be correct for all inputs irrespective of the bounds. MINISAT was the SAT solver used. We ran the experiments on a system with 2.50GHz Core 2 Duo processor and 4.00GB RAM running Windows 7.

### 4.3 Results

Table 1 enlists the different types of errors seeded into the `BST insert` and `ANTLR addChild` methods. Figures 3a and 3b show the code snippets of the methods with the errors seeded. The errors have been categorized into 3 scenarios as described below.

**Table 1.** Case Study Results: P1 - `BST.insert`, P2 - ANTLR `BaseTree.addChild`. Errors categorized into 3 scenarios described in section 4.3. The number of actually faulty and correct statements in the suspect list of fault localization(FL) scheme are enumerated. Description highlights the type of the faulty statements. Efficiency measured by Repair Time and number of SAT Calls. Accuracy measured by (i) whether a fix was obtained, (ii) was the repaired statement exactly same as in correct implementation. Every result is an average of 5 runs(rounded to nearest whole number).

| Name | Scr# | Error# | FL Scheme Output | | Type of Stmts | Repair Time(secs) | # SAT Calls | Accuracy |
|---|---|---|---|---|---|---|---|---|
| | | | # Faulty | # Correct | | | | |
| P1 | 1 | 1 | 1 | 0 | Assign Stmt | 3 | 2 | $\surd$, Same |
| | | 2a | 1 | 0 | Branch stmt | 34 | 114 | $\surd$, Diff |
| | | 2b | 1 | 0 | Branch stmt | 4 | 2 | $\surd$, Same |
| | | 3a | 1 | 0 | Assign stmt | 5 | 2 | $\surd$, Diff |
| | | 3b | 1 | 0 | Assign stmt | 5 | 4 | $\surd$, Same |
| | | 4a | 1 | 0 | Branch stmt | 12 | 96 | $\surd$, Diff |
| | | 4b | 1 | 0 | Branch stmt | 4 | 2 | $\surd$, Same |
| | | 4c | 1 | 0 | Loop condition | 1 | 2 | $\surd$, Same |
| | | 5 | 2 | 0 | Branch, Assign stmts | 7 | 5 | $\surd$, Same |
| | | 6 | 2 | 0 | Assign stmts | 5 | 3 | $\surd$, Same |
| | 2 | 7 | 1 | 2 | Branch, Assign stmts | 15 | 21 | $\surd$, Same |
| | | 8 | 2 | 1 | Branch, Assign stmts | 6 | 2 | $\surd$, Same |
| | | 9 | 1 | 1 | Assign stmts | 11 | 2 | $\surd$, Same |
| | 3 | 10 | 4 | 0 | Branch, Assign stmts | 6 | 8 | $\surd$, Same |
| | | 11 | 2 | 0 | Branch, Assign stmts | 26 | 9 | $\surd$, Same |
| | | 12 | 2 | 1 | Branch, Assign stmts | 33 | 14 | $\surd$, Same |
| | | 13 | 2 | 1 | Assign, Branch stmts | 14 | 24 | $\surd$, Same |
| | | 14 | 0 | 2 | Omission error | NA | NA | NA |
| P2 | 1 | 1 | 2 | 0 | Assign Stmt | 71 | 2 | $\surd$, Diff |
| | 2 | 2 | 2 | 2 | Branch, Assign stmts | 1 | 5 | $\surd$, Same |

**Binary Search Tree insert**

*Scenario 1: All code faults identified in the initial verification, suspect list contains the exact list of faulty statements.* In the first eight errors only 1 statement is faulty. In errors 1 and 3, the fault lies in the operands of an assignment statement. Error 1 involves a wrong update to the `parent` field causing a cycle in the output tree. The faults in the latter case, assign wrong values to local variables `x` and `y` respectively, which impact the program logic leading to errors in the output structure. For instance, in error 3a, the variable `x` is assigned to `x.right` instead of `x.left` if the input value `k < x.key` inside the loop. This impacts the point at which the new node gets added thus breaking the constraints on the `key` field. The repaired statement produced is `x = y.left` which is semantically similar to the expected expression, since the variable `y` is always assigned to `x` before this statement executes. In error 3, both the faults being inside a loop, require a higher scope to get detected as compared to the fault in error 1. This increases the search space for the solver to find correct valuations for the fault revealing input, resulting in higher repair time. Errors 2 and 4 involve faulty branch statements present outside and inside a loop respectively. In Errors 2a and 4a, the comparison operator is wrong and both the operands are parameterized as $V_{lhs} > V_{rhs}$. The search is

on the integer domain and passes through many iterations before resulting in valuations that produce an abstraction satisfying all inputs. The final expression is semantically same as the correct implementation, with the operands interchanged (`y.key > k vs k < y.key`). In errors 2b and 4b, the expression specified in the branch condition is faulty. Hence only the variable in the expression is parameterized (`k <` $V_{rhs}$`.key`). The search on the node domain results in correctly repaired statements with lesser number of SAT calls than a search on the bigger integer domain. In errors 5 and 6, combinations of branch and assignment statements are simultaneously faulty. Since the number of newly introduced variables are higher, the repair times are higher than the previous errors. Also, when there are more than one statements in fault, combinations of abstractions and valuations corresponding to each statement, need to be parsed to look for a solution satisfying all inputs.

*Scenario 2: Fault localization scheme produces a possibly faulty list.* In this scenario, the suspect list also includes statements which are actually correct. For instance, in error 7, the fault lies in an assignment statement which wrongly updates the `parent` of the inserted node similar to error 1, but the suspect list also includes 2 branch statements before this statement. In this case, all the operands of the 3 statements are parameterized leading to an increase in the state space and hence the repair time. It can be observed that as the percentage of actually faulty statements increases, the number of SAT calls decreases. In error 9, an assignment statement inside a loop wrongly updates a variable which is used by a subsequent branch statement. Owing to the data dependency between the statements, the number of possible combinations of correct valuations are less resulting in just 2 SAT calls, however, the large size of the fault revealing input increases the search time. The results were manually verified to ensure that the expressions assigned to the actually correct statements in the final repaired program were the same as before.

*Scenario 3: There could be other faults than those revealed by the initial verification.* Not all code faults may get detected in the initial verification stage. For instance, in error 10, an input structure as small as an empty tree can expose the fault in the branch statement. However, when the correction for this fault is validated using a higher scope, wrong updates to the `parent` and the `right` fields get detected. The new fault-revealing input and the CFG corrected for the first fault are fed back into fault localization scheme and the process repeats until a program which is correct for all inputs passing through the repaired statements is produced. However, when the entire program is checked, a wrong update to the `left` field of the inserted node gets detected. Since this assignment statement is control dependent on the branch which directs flow towards the update of the `right` field, it is not covered by the traces of the previous fault revealing inputs. The process repeats as explained before until the last fault is corrected. This explains the 8 calls to SAT to correct this error scenario. Owing to the small size of the fault revealing inputs, the suspect list containing only the erroneous statements and the fact that in most runs, the first abstraction for every faulty statement happened to be the correct ones, the total repair took only 5 seconds. However, correction of subsequent errors necessitate more number of iterations leading to higher repair times, which got exacerbated when the suspect list also included statements not in error. In the last

case (error 14), the statement to update the `parent` field is omitted from the code, the localization scheme wrongly outputs the statements which update the `left` and `right` fields as being possibly faulty. The repair module is unable to find a valuation which produces a valid output for the fault revealing input. It thus displays a message stating that it could possibly be an omission error or the statements output by the localization scheme could be wrong.

**ANTLR addChild:** The `addChild` method is much more complex than BST insert consisting of calls to 4 other methods, nested branch structures and total source code LOC of 45 (including called methods). The first fault consists of a faulty increment to the integer index of a loop. An extra local variable `j` is assigned the value of integer index (`i`) plus 1 inside the loop. After every iteration, `i` is assigned `j + 1`, erroneously incrementing the index by 2 instead of 1. This fault requires an input of size 4 nodes and 2 unrolls of the loop to get detected. The error can be corrected by assigning `i` instead of `i + 1` to `j`, or `j` to `i` after every iteration, or `i + 1` to `i`(original implementation). We assumed that the suspect list includes both the assignment statements. In the repaired version, `i` was assigned `i + 1` after every iteration and the assignment to `j` was not altered. The correction was performed in just 2 SAT calls but consumed 71 seconds due to the large state space. The second scenario simulates a case wherein both a branch statement checking whether the current tree has any children or not and a statement updating the `parent` field of the added child tree are faulty. These faults are detected with a scope of 3 and unrolls 1. Two actually correct update statements are also included in the suspect list. Our technique is able to repair all the statements in one second.

## 5 Discussion

As can be observed from the results of the evaluation, our repair technique was successful in correcting faults in different types of statements and programming constructs. The technique was able to correct up to 4 code faults for `BST insert` within a worst case time of 33 seconds. The technique consumed a little more than a minute (worst case) to correct faults in `ANTLR addChild`, highlighting its ability to scale to code sizes in real world applications. Overall, we can infer that the repair time is more impacted by the size of the fault revealing input rather than the number of lines of source code. It has been empirically validated that faults in most data structure programs can be detected using small scopes [1]. This is an indication that our technique has the potential to be applicable for many real world programs using data structures.

One of the shortcomings of the technique is that the accuracy of the repaired statements is very closely tied to the correctness and completeness of the post-condition constraints and the maximum value of scope used in the validation phase. For instance, in the second error scenario of `ANTLR addChild`( 3b), when the post-condition just checked the addition of a child tree, an arbitrary expression yielding `true` was substituted in place of the erroneous branch condition. However, only when the constraints were strengthened to ensure that the child had been inserted in the correct position in the children list of the current tree, an accurate branch condition same as that in the original correct implementation was obtained. This limitation is inherited from specification-based bounded verification techniques which consider a program to be correct as long

as the output structure satisfies the user-specified post-condition constraints. Hence, the technique can be relied upon to produce near accurate repair suggestions, which need to be manually verified by the user. However, the user can either refine the constraints or the scope supplied to the tool to refine the accuracy of the corrections. Following points are avenues for improvements in the repair algorithm; Erroneous constant values in a statement could be corrected by parameterizing them to find correct replacements but avoiding the abstraction step. Erroneous operators in a statement could be handled with the help of user-provided templates of possibly correct operators, similar to the handling of omission errors as described in Section 3. The number of iterations required to validate the abstractions for a particular valuation can be decreased by always starting with those involving direct use of local variables declared in the methods. Methods that manipulate input object graphs often use local variables as pointers into the input graphs for traversing them and accessing their desired components. Hence the probability of them being the correct abstractions would increase.

## 6 Correctness Argument

In this section, we present a brief argument that our repair algorithm is accurate, terminates and handles a variety of errors. We make the assumptions that the faults are not present in the operator or constant values specified in a statement. The basic intuition behind parameterizing possibly faulty statements is that any destructive update to a linked data structure should be through wrong updates to one or more fields of the objects of the structure. Hence we try to correct the state by altering the values read and written by statements directly updating fields. We also cover cases where wrong local variable values may indirectly lead to wrong field updates. We also consider erroneous branch condition expressions which may direct the flow towards a wrong path. The variables introduced during parameterizations are constrained exactly at the location of the faulty statements. Further, the state values assigned to them need to be reachable from the local variables at that program point. The fact that the input is fixed to a valid structure and that the statements before and after the faulty statements are correct, ensures that finding an abstraction at the respective program points should lead to a valid update to the state such that the output apart from being structurally correct, is in alignment with the intended program logic. An abstraction yielding a correct output for a particular input structure need not be correct for all input structures covering that statement. There could be bigger structures with higher depth or traversing a different path before reaching the repaired statement, which may require a different abstraction. There could also be more than one correct valuation which yield correct output for a particular input. For instance, when both operands of a branch statement are parameterized, the variables could take any value that results in the correct boolean output to direct the control flow in the right direction. These cases are detected in the abstraction and validation stages and other valuations and/or abstractions for a particular valuation are systematically checked. Provided that there are no new code faults and all the faulty statements are guaranteed to be in the suspect list, a valuation and/or abstraction satisfying all inputs should be in the list obtained from the initial fault revealing input. Hence, the repair process would terminate.

The guarantee on the completeness and correctness of the repaired statements is the same as that provided by other scope bounded verification techniques. The repaired statements are correct with respect to the post-condition specifications for all inputs within the highest scope used to validate the repaired program.

## 7 Related Work

We first discuss two techniques most closely related to our work: program sketching using SAT [15] (Section 7.1) and program repair based on data structure repair [11] (Section 7.2). We next discuss other related work (Section 7.3).

### 7.1 Program Sketching Using SAT

Program synthesis using sketching [15] employs SAT solvers to generate parts of programs. The user gives partial programs that define the basic skeleton of the code. A SAT solver completes the implementation details by generating expressions to fill the "holes" of the partial program. A basic difference between our work and program sketching stems from the intention. We are looking to perform repair or modifications to statements identified as being faulty while sketching aims to complete a given partial but fixed sketch of a program. A key technical difference is our support for specifications or constraints on the concrete state of the program. Therefore, the search space for the SAT solver is the bounded state space of the program, instead of the domain of all possible program expressions as in sketching. To illustrate the difference between repair and sketching, consider the doubly linked list example. The sketching process starts with a random input structure. Assume that this is the same input structure with two nodes as in Fig. 1. Assume also that the user specified the details for the entire `remove` method except for the statement that updates `prevP.next`). The user would then need to specify a generator stating all possible program expressions that can occur at the "hole" (RHS of the assignment statement). Following the sketch language described in the synthesis paper, this would be specified using generators as shown:

$\#define$ `LOC` $=$ {(`head`|`ptr`|`prevP`).(`next`|`prev`)?.(`next`|`prev`)?|`null`}
`prevP.ptr = LOC;`

This bounds the possible expressions to be all values obtained by starting from `header`, `ptr`, `prevP` and subsequent de-referencing of `next` or `prev` pointers (`header, header.next, ptr.next.prev, prevP.prev.prev` so on). Even for this small example with just 1 unknown in 1 statement, the number of possible program expressions to be considered is more than the list obtained by mapping back from the correct concrete state. As the program size and complexity increases, the set of all possible program expressions that could be substituted for all the holes can become huge. Further, to obtain a correct program, the user needs to estimate and accurately specify all the possible expressions. In such cases, it would be faster to look for a correct structure in the concrete state space. The number of expressions mapping to the correct state value at a program point would be much lesser in number. Our technique automatically infers the correct expressions at a point and does not require the user to specify them. The synthesis technique employs an iterative process of validation and refinement (CEGIS)

to filter out sketches that do not satisfy all inputs, similar to our validation phase. However since synthesis starts with a random input which may have poor coverage, it would require higher number of iterations to converge to a correct sketch. Since we start with the input that covers the faulty statement, the iterations should converge to a correct abstraction faster. Further, if the statements produced by fault localization can be guaranteed to be the only ones faulty, the correct expression must be present in the list of abstractions for the first input. Hence scanning this list would be faster than having to feedback every new counter-example into the system.

### 7.2 Program Repair Using Data Structure Repair

On-the-fly repair of erroneous data structures is starting to gain more attention in the research community. Juzi [5] is a tool which performs symbolic execution of the class invariant to determine values for fields that would yield a structurally correct program state. A recent paper [11] presents a technique on how the repair actions on the state could be abstracted to program statements and thus aid in repair of the program. The main drawback of this technique is that it focuses on class invariants and does not handle the specification of a particular method. Hence the repaired program would yield an output that satisfies the invariants but may be fairly different from the intended behavior of the specific method. Further, in cases where the reachability of some nodes in the structure gets broken, Juzi may be unable to parse to the remaining nodes of the tree and hence fail to correct the structure. Error 4c ( 3a) in the `BST insert` method, highlights such a scenario wherein the loop condition used to parse into the tree structure being wrong, the new node gets wrongly added as the root of the tree. Our technique looks for a structure that satisfies the specific post-condition of method, which requires that the nodes in the pre-state also be present in the output structure. Hence a correct output tree structure gets produced.

### 7.3 Other Recent Work on Program Correction

Our technique bears similarities with that of Jobstmann et al [9]. Given a list of suspect statements, their technique replaces the expressions of the statements with unknowns. It then uses a model checker to find a strategy that can yield a solution to the product of a model representing the broken program and an automaton of its linear time logic (LTL) specifications. Though the basic method of formulating the problem is similar, their problem space is very different from ours since their technique is targeted and evaluated on concurrent, finite state transition systems against properties written in temporal logic, whereas our technique is developed for sequential programs that operate on complex data structures. Extending their approach to the domain of complex structures is non-trivial due to the dissimilarity in the nature of programs under repair as well as the specifications. Moreover, our approach of bounded modular debugging of code using declarative specifications likely enables more efficient modeling and reasoning about linked structures. Additionally, translation of the specifications and the entire code into a SAT formula aids in exploiting the power of SAT technology in handling the size and complexity of the underlying state space.

In contrast to specification-based repair, some recently developed program repair techniques do not require users to provide specifications and instead multiple passing

and failing runs with similar coverage. Machine learning [8] and genetic programming based techniques [19] fall in this category. However, these techniques have a high sensitivity to the quality of the given set of tests and do not address repairing programs that operate on structurally complex data. The recently proposed AutoFix-E tool [17] attempts to bridge the gap between specification-based and test-based repair. Boolean queries of a class are used to build an abstract notion of the state which forms the basis to represent contracts of the class, fault profile of failing tests and a behavioral model based on passing tests. A comparison between failing and passing profiles is performed for fault localization and a subsequent program synthesis effort generates the repaired statements. This technique however only corrects violations of simple assertions, which can be formulated using boolean methods already present in the class, but not rich post-conditions.

## 8 Conclusion

This paper presents a novel technique to repair data structure programs using the SAT-based Alloy tool-set. Given a fault revealing input and a list of possibly faulty statements, the technique introduces nondeterminism in the statements and employs SAT to prune the nondeterminism to generate an output structure that satisfies the post-condition. The SAT solution is abstracted to program expressions, which yield a list of possible modifications to the statement, which are then iteratively refined using bounded verification. We performed an experimental validation using a prototype implementation of our approach to cover a variety of different types and number of faults on the subject programs. The experimental results demonstrate the efficiency, accuracy of our approach and the promise it holds to scale to real world applications.

Our tool can act as a semi-automated method to produce feedback to fault localization schemes such as [14] to refine the accuracy of localization. The concept of searching a bounded state space to generate program statements can also be used to develop more efficient program synthesis techniques. The idea of introducing nondeterminism in code can aid in improving the precision of data structure repair as well. We also envisage a possible integration of our technique with other contract and test suite coverage based techniques like AutoFix-E [17]. We believe our approach of modeling and reasoning about programs using SAT can aid in improving the repair efficiency and accuracy of existing tools.

## Acknowledgements

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967 and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

## References

1. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the "Small Scope Hypothesis". Technical report, MIT CSAIL (2003)
2. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA (2006)

3. Collofello, J.S., Cousins, L.: Towards automatic software fault location through decision-to-decision path analysis
4. Dennis, G., Chang, F.S.-H., Jackson, D.: Modular verification of code with SAT. In: ISSTA (2006)
5. Elkarablieh, B., Khurshid, S.: Juzi: A tool for repairing complex data structures. In: ICSE (2008)
6. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT-P, Cambridge (2006)
7. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA (2000)
8. Jeffrey, D., Feng, M., Gupta, N., Gupta, R.: BugFix: A learning-based tool to assist developers in fixing bugs. In: ICPC (2009)
9. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
10. Jones, J.A.: Semi-Automatic Fault Localization. PhD thesis, Georgia Institute of Technology (2008)
11. Malik, M.Z., Ghori, K., Elkarablieh, B., Khurshid, S.: A case for automated debugging using data structure repair. In: ASE (2009)
12. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs (2001)
13. Parr, T., et al.: Another tool for language recognition, `http://www.antlr.org/`
14. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE (2003)
15. Solar-Lezama, A.: The sketching approach to program synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 4–13. Springer, Heidelberg (2009)
16. 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)
17. Wei, Y., Pei, Y., Furia, C.A., Silva, L.S., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: ISSTA (2010)
18. Weimer, W.: Patches as better bug reports. In: GPCE (2006)
19. Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE (2009)
20. Weiser, M.: Programmers use slices when debugging. Commun. ACM (1982)