# Automatic Fault Location for Data Structures

Vineet Singh
University of California, Riverside, USA
vsing004@cs.ucr.edu

Rajiv Gupta
University of California, Riverside, USA
gupta@cs.ucr.edu

Iulian Neamtiu
New Jersey Institute of Technology, USA
ineamtiu@njit.edu

## Abstract

Specification-based data structure verification is a powerful debugging technique. In this work we combine specification-based data structure verification with automatic detection of faulty program statements that corrupt data structures. The user specifies the consistency constraints for dynamic data structures as relationships among the nodes of a memory graph. Our system detects constraint violations to identify corrupted data structures during program execution and then automatically locates faulty code responsible for data structure corruption. Our approach offers two main advantages: (1) a highly precise automatic fault location method, and (2) a simple specification language. We employ incremental constraint checking for time efficient constraint matching and fault location. On average, while Tarantula statistical debugging technique narrows the fault to 10 statements, our technique narrows it to $\approx 4$ statements.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability, Validation; D.2.5 [*Software Engineering*]: Testing and Debugging—Tracing; F.3.1 [*Logics and Meanings of Programs* ]: Specifying and Verifying and Reasoning about Programs—Specification techniques

***General Terms*** Languages, Debugging, Verification

***Keywords*** Data structure error, memory graph, constraint checks, fault location

## 1. Introduction

Faulty code often leads to data structure corruption. Heap-allocated data structures can be easily modeled using Memory Graphs [30] – heap allocations and pointers between heap elements correspond to nodes and edges. The structural definition of a data structure can be expressed via *consistency constraints* that describe the allowed relationships among the nodes of the memory graph. The *data structure errors* that violate these constraints can be detected and located by evaluating the constraints.

As an example, consider the widely-used memory allocator in the GNU C library (glibc) that maintains a doubly-linked list to track free memory chunks. The structural *consistency constraints* of this list are often violated by bugs in client programs (e.g., heap overflow bugs) leading to a program crash. Examples of bugs in popular software that do exactly the above include: Kate (KDE bug #124496), Kdelibs (KDE#116176), Kooka (KDE#111609), Open office (Open office#77015), GStreamer (GNOME#343652), Doxygen (GNOME#625051), Rhythmbox (GNOME#636322), Evolution (GNOME#338994, GNOME#579145) [1–3].

Motivated by the above observations, we have built a system that (1) allows the user to specify the consistency constraints for dynamic data structures as relationship rules among the nodes of a memory graph, (2) automatically detects any violation of these rules at runtime, and (3) locates the faulty code. Designing such a system is challenging because data structure invariants are routinely broken, albeit temporarily, during operations on the data structure. For example, the structural invariant for a doubly-linked list, *if element* ***e*** *points to element* ***e'*** *then* ***e'*** *should point back to* ***e***, is violated temporarily during the insertion of a new node in the list. The presence of temporary constraint violations makes it crucial for the fault location system to be able to differentiate between a constraint violation caused by a fault and a temporary legitimate violation. Moreover, the fault location system must deliver ease of use and runtime efficiency.

Our system provides support for specification-based fault location via two main constructs. First, a simple yet effective specification language for writing consistency constraints for data structures. Second, a directive for specifying ***C-points***, i.e., program points where our system will check at runtime whether the constraints are satisfied; at such points, the data structure is supposed to be in a consistent state with respect to the provided specification. *C-points* are akin to transactions hence emerge naturally, e.g., at the beginning and end of functions that modify the data structure. *C-points* allow us to detect data structure corruption early, before it gets a chance to turn into further state corruption or crash. In addition, if the program crashes then the crash point is used as a *C-point*. Our technique can work even with a single *C-point*, while more *C-points* imply greater precision.

As the program executes, our system traces the evolution history of the data structures. Once a constraint violation is detected, it identifies the corrupted data structures and the set of inconsistencies. It traces back through the evolution history, searching for program points where inconsistencies were introduced and collecting a list of faulty program statements. We use optimizations based on temporal and spatial knowledge of the data structures to narrow down the search space, making the trace back efficient. We also use a compact memory graph representation and employ an incremental constraint matching algorithm to make the technique space and time efficient. Although specification-driven detection of data structure errors has been proposed before [12], our approach offers two main advantages:

*1. Highly Precise Fault Location:* our system offers more than just error detection, as it locates the execution point and program statement that caused a rule violation. Prior work in handling data structure errors has been limited to detecting the erroneous program state—after detection the user must find the fault manually. For example, Archie [14] localizes the error to the region of the ex-



*CC'16*, March 17–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-4241-4/16/03...$15.00
http://dx.doi.org/10.1145/2892208.2892215

ecution between the first call to the constraint checker that detects an inconsistency and the immediately preceding call. This compels the user to insert frequent time-consuming consistency checks and locate the error manually. When compared to dynamic slicing and the Tarantula statistical debugging technique, our approach is much more effective.

*2. Simple Yet Expressive Specification:* the design of our specification language makes it easier for the user to specify the rules. Prior works have proposed specification languages that are rich but complex. For example, in the approach by Demsky and Rinard [12], specifying a doubly-linked list takes 14 lines of specification whereas in our approach it takes just 4 lines. Moreover, our language is powerful so it allows specifying data structures such AVL trees, red-black trees, that other languages do not.

In addition, we employ a space efficient unified memory graph representation [34] from which the memory graph at any prior execution point can be extracted. It also allows constraint checks to be performed incrementally.

Our implementation uses the Pin dynamic instrumentation framework [25] for instrumenting Linux executables for the IA-32 architecture. We have evaluated the efficiency and effectiveness of our techniques; we now highlight the results. Our specification language allows constraints for widely-used data structures to be defined using just 2–9 lines of specification code. Experiments show that our approach narrows down the fault to 1–10 statements, which is much smaller than results of dynamic slicing and tarantula statistical technique. Fault location is also performed efficiently: following a substantial execution, constraints are checked in typically a second or less, and faults are localized in less than 2 minutes. Finally, our incremental checking optimizations substantially reduce the time and memory cost associated with checking. This allows users to perform more frequent and finer-grained checks, accelerating fault detection.

Our contributions include:

- **An error detection and fault location system** that, given a data structure consistency specification, automatically detects data structure errors, and upon detection, traces errors back to their source.
- **A new data structure consistency constraint specification language** that allows the user to easily and concisely express structural consistency properties.
- **Data structure-specific trace back.** Our fault location system identifies corrupt data structure(s) out of multiple program data structures and uses this information for efficiently locating faulty code.
- **An incremental method of checking consistency constraints** that uses information from the previously performed checks to reduce time and memory overhead.

## 2. Overview of Our Approach

In this section we provide an overview of our approach and highlight its key features using an example.

The example is centered around quad trees—a widely used data structure for spatial indexing. A quad tree is a tree with internal nodes having four children and the data stored at the leaf nodes. Thus one of the key structural consistency constraints for this data structure is: *for any internal element* ***e****, the number of children is four.* In Figure 1 we show example code that creates and manipulates quad trees, and contains a bug which leads to a violation of the consistency constraint. The quad tree definition (lines 2–7) contains nine fields: the first five fields store data about the node, while the next four fields, child[4], point to children in the quad tree. The main function reads coordinates (x,y) from a file and populates the tree

```
1  struct pt { int x, y };
2  struct qdtree {
3      int posX, posY;
4      int width, height;
5      struct pt * point;
6      struct qdtree * child [4];
7  }
8  struct qdtree * root;
9  int main()
10 {
11     ...
12     while (fscanf( file , "%d%d",&x,&y)!=EOF)
13         insert(x,y, root);
14 }
15 void insert(int x, int y, struct qdtree * root)
16 {
17     ##C−POINT(qdtree)
18     if (root == NULL) {
19         root = (struct qdtree *)malloc(sizeof(qdtree));
20         temp = create_point(x, y);
21         root→ point = temp;
22     }
23     else {
24         n = search(x, y, root);
25         if (n→ pt != NULL)
26             split (x, y, n);
27     }
28     ##C−POINT(qdtree)
29 }
30
31 void split (int x, int y, struct qdtree * node)
32 {
33     struct qdtree * temp;
34     for (i = 0; i < 4; i++) {
35         temp = (struct qdtree *)malloc(sizeof(qdtree));
36         set_node_fields(temp, i, node);
37         node→ child[i] = temp;
38         if (x==node[i]→ posX && y==node[i]→ posY) {
39         ...
40             parent_node→ child[0]= NULL;
41         }
42         else {
43             move_value(node, node→ child[i]);
44             assign_value(x, y, node→ child[i]);
45         }
46     }
47 }
```

**Figure 1.** Faulty *Quad Tree* implementation.

$qdtree$ FIELD 9 EDGE 5;
/*# total fields and # pointer fields */
$qdtree$ X; ..($c_1$)
X.ISROOT == FALSE ⇒ X.INDEGREE == 1;

$qdtree$ X; $qdtree$ Y; ..($c_2$)
X ↠ Y ⇒ Y ↛ X;

$qdtree$ X; $qdtree$ Y; ..($c_3$)
X→ Y ⇒ X.OUTDEGREE == 4;

**Figure 2.** Consistency constraints of a *Quad Tree*.

by calling the insert function. In insert, we first search for an existing node that is suitable for coordinates (x,y). If the resulting node n already has a point stored in it, then four children of n are created, one for each quadrant. The point at node n and the newly-read point are then inserted into the quad tree rooted at n. The statement at line 40 in function split which sets an edge to NULL is faulty.

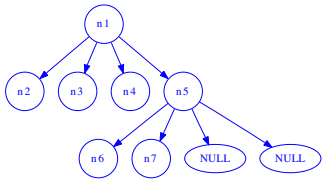
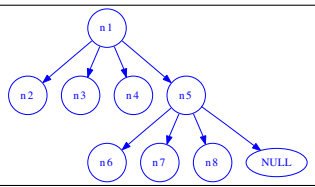
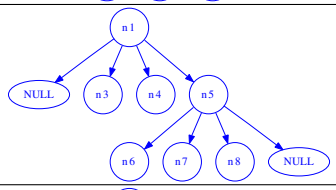
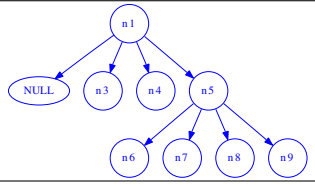
| Execution trace | Inconsistencies | Memory Graph |
|---|---|---|
| 1 temp = (**struct** qdtree ∗)malloc(**sizeof**(qdtree));<br>2 set_node_fields(temp, 1, node);<br>3 node−>child[1] = temp;<br>4 move_value(node, node−>child[1]);<br>5 assign_value(x, y, node−>child[1]);<br>6 i++;<br>∗ ∗ ∗∗ Execution Point 1 | Inconsistences@Point 1<br>$S_1$ = {<br>$\langle c_3, \{n5 \rightarrow n6\}\rangle, \langle c_3, \{n5 \rightarrow n7\}\rangle$<br>} |  |
| 7 temp = (**struct** qdtree ∗)malloc(**sizeof**(qdtree));<br>8 set_node_fields(temp, 3, node);<br>9 node−>child[3] = temp;<br>∗ ∗ ∗∗ Execution Point 2 | Inconsistencies@Point 2<br>$S_2$ = {<br>$\langle c_3, \{n5 \rightarrow n6\}\rangle, \langle c_3, \{n5 \rightarrow n7\}\rangle,$<br>$\langle c_3, \{n5 \rightarrow n8\}\rangle$<br>} |  |
| 10 parent_node−>child[0]= NULL;<br>11 i++;<br>∗ ∗ ∗∗ Execution Point 3 | Inconsistencies@Point 3<br>$S_3$ = {<br>$\langle c_3, \{n1 \rightarrow n3\}\rangle, \langle c_3, \{n1 \rightarrow n4\}\rangle,$<br>$\langle c_3, \{n1 \rightarrow n5\}\rangle, \langle c_3, \{n5 \rightarrow n6\}\rangle,$<br>$\langle c_3, \{n5 \rightarrow n7\}\rangle, \langle c_3, \{n5 \rightarrow n8\}\rangle$<br>} |  |
| 12 temp = (**struct** qdtree ∗)malloc(**sizeof**(qdtree));<br>13 set_node_fields(temp, 3, node);<br>14 node−>child[3] = temp;<br>15 move_value(node, node−>child[3]);<br>16 assign_value(x, y, node−>child[3]);<br>∗ ∗ ∗∗ Execution Point 4 - C-POINT | Inconsistencies@Point 4 - C-POINT<br>$S_c$ = {<br>$\langle c_3, \{n1 \rightarrow n3\}\rangle, \langle c_3, \{n1 \rightarrow n4\}\rangle,$<br>$\langle c_3, \{n1 \rightarrow n5\}\rangle$<br>} |  |

**Figure 3.** Memory Graph at different program points.

| Execution Point | Inconsistencies | Pending Inconsistencies | Relation |
|---|---|---|---|
| Execution Point 4 | $S_4$ = {<br>$e_1 = \langle c_3, \{n1 \rightarrow n3\}\rangle$,<br>$e_2 = \langle c_3, \{n1 \rightarrow n4\}\rangle$,<br>$e_3 = \langle c_3, \{n1 \rightarrow n5\}\rangle$ } | @4<br>$P = \{e_1, e_2, e_3\}$<br>$FS(FaultyStatements) = \phi$ | |
| Execution Point 3<br>$O_3$: $n5$ →child[3] = $n9$;<br>Execution Point 4 | $S_3$ = {<br>$e_1 = \langle c_3, \{n1 \rightarrow n3\}\rangle$,<br>$e_2 = \langle c_3, \{n1 \rightarrow n4\}\rangle$,<br>$e_3 = \langle c_3, \{n1 \rightarrow n5\}\rangle$,<br>$e_4 = \langle c_3, \{n5 \rightarrow n6\}\rangle$,<br>$e_5 = \langle c_3, \{n5 \rightarrow n7\}\rangle$,<br>$e_6 = \langle c_3, \{n5 \rightarrow n8\}\rangle$ } | @3<br>$P = \{e_1, e_2, e_3\}$<br>$FS = \phi$ | $\{e_4, e_5, e_6\} \wr \{e_1, e_2, e_3\}$ = FALSE<br>$O_3 \rightsquigarrow e_1$ = FALSE<br>$O_3 \rightsquigarrow e_2$ = FALSE<br>$O_3 \rightsquigarrow e_3$ = FALSE |
| Execution Point 2<br>$O_2$: $n1$ →child[0] = NULL;<br>Execution Point 3 | $S_2$ = {<br>$e_4 = \langle c_3, \{n5 \rightarrow n6\}\rangle$,<br>$e_5 = \langle c_3, \{n5 \rightarrow n7\}\rangle$,<br>$e_6 = \langle c_3, \{n5 \rightarrow n8\}\rangle$ } | @2<br>$P = \phi$<br>$FS = \{O_2\}$ | $\{e_4, e_5, e_6\} \wr \{e_1, e_2, e_3\}$ = FALSE<br>$O_2 \rightsquigarrow e_1$ = TRUE<br>$O_2 \rightsquigarrow e_2$ = TRUE<br>$O_2 \rightsquigarrow e_3$ = TRUE |

**Figure 4.** Fault location on Figure 1.

***Specification of Consistency Constraints.*** Our specification language enables the developer to express the data structure constraints directly in terms of the relationships among heap elements which makes the specifications compact and their writing intuitive. In prior approaches [12, 14], the developer must first convert their data structure definition into a high-level model and then express the constraints in terms of that model. This makes writing data structure specifications complex and error-prone.

Figure 2 specifies the consistency constraints for a quad tree in our language. The user specifies the structure of each node in the memory graph by declaring a node type (*qdtree*) that contains 9 data fields and 5 pointer fields – (FIELD 9) and (EDGE 5); constraints are specified next. The constraint specification involves declaring node variables and specifying the relationships among them. For quad tree we specify three constraints in terms of variables X and Y of node type *qdtree*. The constraint $c_1$ indicates that all nodes besides the root have an indegree of 1. The constraint $c_2$ indicates that if there is a path from node X to node Y, then there cannot be a path from node Y to node X. The constraint $c_3$ specifies that, if any node X points to another node Y (i.e., the node X is internal) then the outdegree of X is 4. We have chosen a simple constraint for the purpose of understanding, our language can handle variety of complex constraints (as explained in section 4).

***Tracing data structure evolution history.*** We execute the program and trace the evolution history of the program data structure using binary level dynamic instrumentation. The instrumentation is independent of the constraints specified. We only instrument the allocation/deallocation calls and memory writes to the allocated memory. Once the program execution completes; normally or be-

cause of a program crash, the traced information is used to construct memory graph.

***Fault Location.*** We match the specified constraints over the memory graph at program points corresponding to *C-points*. The memory graph must be in a consistent state with respect to the specified constraints at these program points. Once we encounter a consistency constraint violation, the process of fault location begins. The process of fault location involves tracing back the program execution. We analyze the effect of each operation (on the memory graph) on the inconsistencies present in the memory graph. The statements corresponding to the operations contributing to the inconsistencies are added to list of potentially faulty statements.

Our system performs consistency checks on the program memory graph at the beginning and at the end of function insert (lines 17 and 28) indicated by *C-points*. In Figure 3, the first column shows an execution trace of the code in Figure 1, while the second and third columns contain the set of constraints violated at selected program execution points and the corresponding quad tree. Note that execution point 4 is a *C-point* and we find that there are multiple violations of constraint $c_3$ from Figure 2 at this point. The set of violations is represented by the set $S_c$ and each violation is described in form of a tuple $\langle C, G\rangle$, where $G$ is the sub-graph over which constraint $C$ is violated. In the Figure 3 example, the constraint violations include: $\langle c_3, \{n1 \rightarrow n3\}\rangle$, $\langle c_3, \{n1 \rightarrow n4\}\rangle$, and $\langle c_3, \{n1 \rightarrow n5\}\rangle$. We perform consistency checks at earlier execution points, computing the violations into set $S_i$ where $i$ is the execution point. The fault location algorithm finds the operations contributing to inconsistencies in $S_c$ by examining the $S_i$'s.

In our example, the fault location algorithm determines that at program point 3, the operation of setting $n1 \rightarrow$child[0] to NULL introduces all the inconsistencies present in the set $S_c$. The statement corresponding to the operation is output as a faulty statement. Our algorithm also determines that none of the inconsistencies in $S_2$ are related to inconsistencies in $S_c$ and therefore there is no need to further search for faulty statements. Note that there are other inconsistencies present at different program points that are not related to the inconsistencies at *C-points*. For example, at program point 3, the inconsistency $\langle c_3, \{n5 \rightarrow n8\}\rangle$ is temporary and ignored in the search for faulty statements.

***Optimizations.*** We have optimized our fault location system in terms of both memory and time costs. For fault location, we need the memory graph at each execution point so that the constraint violations at each execution point can be detected. To avoid saving memory graphs at all execution points we employ a unified memory graph representation [34] that combines memory graphs at all execution points into one and distinguishes subgraphs via association of timestamps with graph components (nodes and edges). Thus, portions of the graph that do not change across many execution points are stored only once. Given a unified memory graph representation at program point $P$, the memory graph for any earlier program point can be reconstructed using the process of *rollback* (explained in section 5). We employ an incremental algorithm which avoids redundant constraint evaluations over unchanged parts of the memory graph.

## 3. Fault Location

Our location algorithm is based on the observation that there are two kinds of inconsistencies in the memory graph (MG): a *temporary* kind of inconsistencies that are removed prior to reaching *C-points*, and an *error* kind of inconsistencies, which are present at *C-points* and are caused by faulty statements. The algorithm identifies the statements responsible for the *error* kind of inconsistencies. We first define the key concepts and then present the algorithm.

**Definition 1.** An *inconsistency e* at execution point $i$ is a tuple $\langle c_j, G_i\rangle$ where $c_j$ is a constraint that is violated when evaluated over $G_i$, a subgraph of the memory graph at execution point $i$.

**Example 1.** Consider $\langle c_3, G_4\rangle$ at execution point 4 in Figure 3 where $c_3$ is

qdtree X; qdtree Y;

X→Y ⇒ X.OUTDEGREE == 4;

and $G_4$ is $\langle\{n1 \rightarrow n3\}\rangle$. Then $\langle c_3, G_4\rangle$ is an *inconsistency* at execution point 4 because $c_3$ is violated when $X = n1$ and $Y = n3$.

Note that there can be multiple inconsistencies corresponding to the same constraint violation as the constraint check can fail over multiple sub-graphs.

Consider the execution of operation $O$ such that *beforeO* and *afterO* denote the execution points just before and after execution of $O$. Next we provide the conditions under which inconsistencies at *beforeO* and *afterO* are related (denoted by '≀') to each other and the conditions under which $O$ is considered to be a potentially faulty operation.

**Definition 2.** An inconsistency $\langle c_j, G_b\rangle$ at execution point *beforeO* is said to be *related to* (denoted by '≀') an inconsistency $\langle c_k, G_a\rangle$ at execution point *afterO* **iff** the operation $O$ has modified the arguments to the constraint check of $c_j$ over $G_a$ as well as the arguments to check $c_k$ over $G_b$.

**Example 2.** In Figure 3, the inconsistency $e_1 = \langle c_3, \{n5 \rightarrow n8\}\rangle$ at program point 2 (after) is related to $e_2 = \langle c_3, \{n5 \rightarrow n6\}\rangle$ at program point 1 (before) because the operation of setting of edge for $n5$ to $n8$ modifies $n5$.OUTDEGREE which is an argument to check $c_3$ over the subgraph for both $e_1$ and $e_2$. In other words, $e_1 \wr e_2$ is TRUE.

**Definition 3.** Operation $O$ is said to have *contributed to* (denoted by '⇝') an inconsistency $\langle c_j, G_a\rangle$ present at program point *afterO* if the arguments to the constraint check of $c_j$ over subgraph $G_a$ at program point *beforeO* are not equal to the arguments to the constraint check of $c_j$ over subgraph $G_a$ at program point *afterO*.

**Example 3.** The operation $n1 \rightarrow$child[0] = NULL (statement 10) in Figure 3, contributes to the inconsistency $e = \langle c_3, \{n1 \rightarrow n3\}\rangle$ present in row 3 because it has modified $n1$.OUTDEGREE which is an argument to check $c_3$.

Our fault location algorithm is presented in Algorithm 1. It begins by initializing the set of pending inconsistencies with inconsistencies at *C-point*(line 4). The system rolls back memory graph one operation at a time (line 7) and checks if the rolled back operation has contributed to any of the pending inconsistencies (line 10). Statement corresponding to contributing operation is output as faulty (line 11). Inconsistencies in the new MG which are related to the pending inconsistencies are added to the pending set (line 14). Any pending inconsistencies no longer present in the MG are removed from the pending set (line 18). When the set of pending inconsistencies reduces to $\emptyset$, the algorithm stops. Note that our algorithm only considers operations contributing to inconsistencies as faulty, rather than marking every statement that modifies the inconsistency subgraph as faulty; this helps increase precision.

Figure 4 illustrates the fault localization algorithm for the fault in Figure 1. The set of pending inconsistencies P is initialized with inconsistencies at execution point 4 (C-point) ($P = \{e_1, e_2, e_3\}$). The operation $O_3$: $n5 \rightarrow$child[3] = $n9$ is not faulty because it does not contribute to any of the inconsistencies in $P$, i.e, it does not modify arguments to any of the inconsistencies present in $P$. The

**Algorithm 1** Fault Location

```
1: P: Set of pending inconsistencies;
   S_i denotes inconsistencies in MG_i at execution point i;
   O_i: Operation on MG performed at execution point i;
   Check_Constraints(MG, C): checks constraints in C for MG
   and returns the set of inconsistencies found;
   Roll_Back(MG_{i+1}): rolls back MG_{i+1} by one operation.
2: INPUT: Memory Graph MG_c at execution point c for a C-
   point and constraint specification C ⟨c_1, ...c_n⟩.
3: Fault_Location()
4:    i ← c; P = Check_Constraints(MG_c, C)
5:    do{
6:       i ← i − 1
7:       MG_i = Roll_Back(MG_{i+1})
8:       S_i = Check_Constraints(MG_i, C)
9:       for each e ∈ P do
10:         if e ⇝ O_i == true then
11:            Output(statement(O_i))
12:            for each e' ∈ S_i do
13:               if (e ≀ e')&&(e! = e') then
14:                  P = P ∪ {e'}
15:               end if
16:            end for
17:            if !(e ∈ S_i) then
18:               P = P − {e}
19:            end if
20:         end if
21:      end for
22:   }
      while P! = {φ}
```

next operation $O_2 : n1 \rightarrow$child[0] = NULL contributes to inconsistencies $\{e_1, e_2, e_3\}$ in $P$ and hence is a faulty statement. Thus, $O_2$ is added to the $FS$ set. None of the inconsistencies in $S_2$ are related to those in $P$; hence no new inconsistencies are added to $P$. The inconsistencies $\{e_1, e_2, e_3\}$ are removed from $P$ causing it to become empty and the search for faulty statements terminates. In other words, $O_2$ is identified as faulty.

***Identifying corrupted data structures.*** Matching a constraint with the entire program memory graph will lead to false positives due to violations of a data structure constraint when it is evaluated for other unrelated data structures. To avoid this problem we track the identity of the data structure associated with each memory graph node during program execution. Using the data structure identity, we only evaluate constraints relevant to the memory graph node involved avoiding false positives. Furthermore, knowing the identity of the corrupted data structure helps us during trace back for faults as we limit our search only to the corrupted data structure instead of the whole program memory graph.

## 4. Constraint Specifications

Before introducing our constraint specification language, it is important to mention the following apparent alternatives and explain why we have not used them for our system:

**Archie** [14] uses constraint-based specification for data structure repair. The specification contains a model the data structure must satisfy. When the model is violated, Archie repairs the data structure to satisfy the model. In our system, the model of the constraints is already fixed in terms of the memory graph. Specifying the model again puts significant extra burden on the programmer.

**Alloy** [17] is a rich object modeling language for expressing high-level design properties. In comparison, our language is centered around logical, arithmetic, layout and graph constraints at the data structure level.

**Table 1.** *Standard* Attributes.

| Attribute | Type | Represents |
|---|---|---|
| n.INDEGREE | INT | Indegree of the node n |
| n.OUTDEGREE | INT | Outdegree of the node n |
| n.EXTERNAL | BOOL | (n.OUTDEGREE == 0) ∨ (n.INDEGREE ==0 ) |
| n.INTERNAL | BOOL | (n.INDEGREE != 0) ∧ (n.OUTDEGREE != 0 ) |
| n.ISROOT | BOOL | (n.INDEGREE == 0) |
| n.ISLEAF | BOOL | (n.OUTDEGREE == 0) |

**Programming languages** can be used to specify constraints (e.g., repOK [24]). The constraints written need to executed along with the program and are not useful to us as we need to match constraints during the trace back. Writing constraints in programming languages is verbose and error prone.

Our language is based on the same principles as the aforementioned ones but is designed specifically for the purpose of specifying data structure constraints for debugging. Taking on this specific problem makes our language simpler. In our approach, the relevant program state at an execution point is captured by the *memory graph*, and *consistency constraints* for a data structure are specified in terms of relationships among nodes and edges of the memory graph. We provide the user with a C-like syntax so the language is easy to use with minimum learning requirement. In this section we first define our constraint specification language and demonstrate that our language is both expressive and simple to use. That is, we can handle a variety data structures with equal or less burden (in comparison to other languages) on the programmer using our specification language.

The *memory graph*, at each point in the execution, consists of nodes corresponding to allocated memory regions and the edges are formed by pointers between the allocated memory regions. Each node (representing an allocation) has fields corresponding to the fields of the data structure for which the memory was allocated. The structure of the memory graph corresponds to the shape of the data structure; hence violations in data structure constraints can be detected by evaluating those constraints for the memory graph.

Our specification language is designed to provide an easy way to express the *structural* form of the memory graph for a data structure. In other words, the programmer simply expresses how the data structure can be visualized in the memory, which makes specification writing very intuitive. Specifying data structure constraints involves three steps. First step is specifying the types of nodes in the memory graph. Second (optional) step is specifying any special node attributes which may be involved in the constraints. Third, specifying the constraint using variables of declared types. The grammar of our specification language consists of corresponding three components: *structure*, *model*, and *constraints* (Figure 5).

***Structure specification.*** The nodes in a memory graph can correspond to an *array* or a *structure*. Structures are defined in terms of the number of *fields* and *edges* present in the memory graph nodes. The structure specification declares the types of the memory graph nodes present in the specifications. The specification of the *quad tree* in Figure 2 shows that each node has 9 fields and 5 edges.

**Example 4.** The structure specifications of *B-tree* and *AVL-tree* are:

– struct btree{ int count; int key[2]; struct btree * child[3];}
btree FIELD 6 EDGE 3;

– struct avltree{ int val; struct avltree * right, * left;}
avltree FIELD 3 EDGE 2;

***Attributes and model specification.*** We provide several node attributes (shorthands) that simplify the task of writing the specifications and making them concise. Table 1 contains the list of provided node attributes along with their meaning. Standard attributes are valid for any type declared in the structure specification.

When the standard attributes are not adequate, user-defined node attributes are introduced via the *model* part of the specification language in Figure 5(b). User define node attributes are specific to a node type. Specifying a custom node attribute involves declaring the name ($h$) of the node attribute along with the node type it is associated with ($f$). The declaration is followed by the rules for assigning the attribute value for each node. Assignment rules ($r$) consist of *guard* ($g$), *terminal assignment* ($a$), and *non-terminal assignment*($a$). A *guard* is a precondition that, when true, leads to *terminal assignment* otherwise *non-terminal assignment* is followed. Assignment statement is assignment of an arithmetic expression to the node attribute. Note that the user-defined attributes can only be used for acyclic data structures. Therefore, when such attributes are used, our implementation performs an acyclicity check because bugs may lead to formation of cycles in data structures that are supposed to be acyclic. The model specification allows the user to create node attributes corresponding to real world node properties and constraints can be specified in terms of these node properties.

**Example 5.** The height of an *AVL-tree* node is specified as:

avltree.HEIGHT; avltree X;
X.ISLEAF == true ⇒ X.HEIGHT = 0 ||
X.HEIGHT = (X[2].HEIGHT≥X[3].HEIGHT) ?
X[2].HEIGHT+1 : X[3].HEIGHT+1;

Similarly, for a red black tree node with structure
– struct rbtree{ int color; struct avltree * right, * left;}
the black height derived from the right child is specified as:

rbtree.BHEIGHT; rbtree X;
X.ISLEAF == true ⇒ X.BHEIGHT = 0 ||
X.BHEIGHT = (X[2].(1)!= BLACK) ?
X[2].BHEIGHT : X[2].BHEIGHT+1;

***Constraint specification.*** Our language allows the user to write both inter-node and intra-node constraints. *Inter-node constraints*, defined via the grammar in Figure 5(c) are composed of *declarations*($d$), *guard* (optional), and *body*($g$). A *guard* is a precondition that must be true in order for the constraint to be applicable. The *body* is composed of one or more constraint statements joined by the boolean operator AND. Three types of constraint statements are allowed: boolean($be$), arithmetic($ae$), and connection($ce$). Connection statements indicate the following: $X \rightarrow Y$ (edge allowed), $X \not\rightarrow Y$ (edge not allowed), $X \twoheadrightarrow Y$ (path allowed), and $X \not\twoheadrightarrow Y$ (path not allowed).

Let us consider constraint specifications for the *B-tree* data structure, shown in Figure 6 top part. The first two constraints ensure that the structure represents a tree and are thus the same for all trees (including *Quad-tree* shown earlier). Constraint 1 uses a guard (X.ISROOT == FALSE) to identify non-root nodes and indicate that their indegree must be 1. Constraint 2 uses a guard to indicate that if there is a path from $X$ to $Y$ then there is no path from $Y$ to $X$. The additional constraints in the specification of *B-tree* follow. Constraint 3 and 4 restrict the outdegrees of internal nodes in *B-tree* while constraint 5 ensures that the number of children is 1+ number of stored keys (value stored in the first field of the *B-tree* structure).

**Example 6.** The balanced height constraint for *AVL-tree* is expressed using the user-declared node attribute *HEIGHT* below.

| | | | |
|---|---|---|---|
| Types | $t$ | ::= | $f$ FIELD $n$ EDGE $n$ ; |
| | | \| | ARRAY $f$ ; |

(a) Structure Specification

| | | | |
|---|---|---|---|
| User defined node attribute | $m$ | ::= | $name\ r$ |
| | $name$ | ::= | $f\ .\ h$ ; |
| Rules | $r$ | ::= | $d\ r \mid g\ =>\ a \mid\mid a$ ; |
| Assigment | $a$ | ::= | $x.h\ =\ ve$ |
| | $ve$ | ::= | $ve + ve \mid ve - ve$ |
| | | \| | $ve * ve \mid ve/ve$ |
| | | \| | $\lvert ve\rvert \mid (ve)$ |
| | | \| | $(ae)?ve : ve$ |
| | | \| | $av$ |

(b) Model Specification

| | | | |
|---|---|---|---|
| Intra-node constraint | $c'$ | ::= | $d\ q\ ,\ b$ ; |
| | $q$ | ::= | for I in $n$ to $n$ |
| | $b$ | ::= | $x[e]\ op\ e$ |
| | $e$ | ::= | $e + e \mid e - e \mid e * e$ |
| | | \| | $e/e \mid (e) \mid \lvert e\rvert$ |
| | | \| | $x[e] \mid$ I $\mid n$ |
| Inter-node constrant | $c$ | ::= | $d\ c \mid d\ g\ =>\ g\ ; \mid d\ g$ ; |
| Constraint expression | $g$ | ::= | $g$ and $g \mid g$ or $g$ |
| | | \| | $ae \mid be \mid ce$ |
| | $ae$ | ::= | $av\ op\ xe$ |
| | $xe$ | ::= | $xe + xe \mid xe - xe$ |
| | | \| | $xe * xe \mid xe/xe$ |
| | | \| | $(xe) \mid \lvert xe\rvert$ |
| | | \| | $av$ |
| | $be$ | ::= | $bv\ ==$ true $\mid bv\ ==$ false |
| Edge | $ce$ | ::= | $v \rightarrow x$ |
| | | \| | $v \not\rightarrow x$ |
| Path | | \| | $v \twoheadrightarrow x$ |
| | | \| | $v \not\twoheadrightarrow x$ |

(c) Constraint Specification

| | | | |
|---|---|---|---|
| Variable declaration | $d$ | ::= | $f\ x$ ; |
| Boolean Value | $bv$ | ::= | $v$.EXTERNAL |
| | | \| | $v$.INTERNAL |
| | | \| | $v$.ISLEAF \| $v$.ISROOT |
| Arthmatic Value | $av$ | ::= | $v$.INDEGREE |
| | | \| | $v$.OUTDEGREE |
| | | \| | $v[n] \mid v.h \mid n$ |
| Vertex | $v$ | ::= | $x \mid (x[n])$ |
| Rel. Operators | $op$ | ::= | $==\mid\neq\mid\leq\mid\geq\mid<\mid>$ |
| String | $f, h$ | | |
| Integer | $n$ | | |
| Variable | $x$ | | |

**Figure 5.** Specification language.

avltree X;
(X[2]).HEIGHT - (X[3]).HEIGHT ≤ 1 *and*
(X[2]).HEIGHT - (X[3]).HEIGHT ≥ -1;

The above examples illustrate that our language is powerful enough to express the constraints embodied by commonly used data structures and at the same time it is intuitive for the program-

btree FIELD 6 EDGE 3;

*– Data structure is a tree*

1. btree X; btree Y;
   X.ISROOT == FALSE $\Rightarrow$ X.INDEGREE == 1;
2. btree X; btree Y;
   X $\twoheadrightarrow$ Y $\Rightarrow$ Y $\not\twoheadrightarrow$ X;

*– Internal nodes have 2 or 3 children*

3. btree X; btree Y;
   X $\rightarrow$ Y $\Rightarrow$ 2 $\leq$ X.OUTDEGREE;
4. btree X; btree Y;
   X $\rightarrow$ Y $\Rightarrow$ X.OUTDEGREE $\leq$ 3;

*– Number of children is 1 + number of stored keys*

5. btree X;
   X.OUTDEGREE == X.[1] + 1;

**Figure 6.** Specification for *2-3/B-Tree*

mer to use. While we have shown only non-nested data structures, nested structures can be handled by flattening of structure fields.

***Intra-node constraints.*** Intra-node constraint specifications are aimed at handling array-based implementations of data structures. Our language supports expressing relationships between array elements. *Intra-node constraints*, defined via the grammar given in Figure 5(c) are composed of *declarations*($d$), *range*($q$), and *body*($b$). A *range* gives the *min* to *max* values of node field index (I) on which the constraint will be applicable. The body of the constraint is a relational expression in terms of the value of the field in question.

**Example 7.** Consider the shard graph representation [22], implemented as an array, storing 8 entries (each entry has source node, source value, edge value, and destination node). The constraint that the source nodes should be ordered is represented as:

ARRAY shard;
shard X;
for I in 0 to 8, X[I*4 +1] < X[(I+1)*4 +1];

Note that we do not specify the types of structure fields in our specification language. The user can specify constraint to check the type of a field specific to the implementation. Following is an example of type checking for a linked list.

**Example 8.** Consider a linked list implementation using the following structure
– struct node{ int value; struct node * next;}

Following is the constraint specification for checking the type-safety of the $next$ field.

node FIELD 2 EDGE 1;
node X; node Y;
X[2] $\neq$ NULL$\Rightarrow$ (X[2])==Y;

The constraint statement states that if the $next$ field of node X is not equal to NULL, if must point to another node Y.

***Comparison with Archie.*** We compared specification in our language with that written in Archie [14] for several data structures and summarize the results, i.e., the number of statements required to express various data structures, in Table 2. The table shows the compact and expressive nature of our specification in comparison to Archie. The empty rows indicate that Archie is not able to express three data structures. One of the data structures that Archie cannot specify is the AVL tree for which our specification was shown earlier. Archie cannot handle *global constraints*, e.g., that the difference in heights between the left subtree and right subtree of an AVL tree node should not be more than 1. Our specification allows the user to specify global constraints via user-defined node attributes.

**Table 2.** Specification size comparison.

| **Data Structure** | Number of Statements | |
|---|---|---|
| | **Ours** | **Archie [14]** |
| Circular Linked List | 3 | 7 |
| Doubly-linked List | 2 | 8 |
| Binary Tree | 3 | 13 |
| Binary Heap | 4 | 14 |
| B-tree | 6 | 21 |
| Quad Tree | 4 | 23 |
| AVL Tree | 6 | – |
| Red-Black Tree | 9 | – |
| Leftist Heap | 5 | – |
| Full K-ary Tree | 4 | 4*K+7 |

## 5. Optimizations

Our system is optimized to reduce memory and time overhead. First, we have employed a compact memory graph representation. Using this representation, $MG_t$ stores the program state at each program point from 0 to $t$ in one memory graph. Second, we use incremental constraint checking to reduce constraint matching overhead. Third, we employ prediction to reduce time spent in fault location. Next we briefly describe these techniques.

***Memory Graph Representation.*** Memory graphs [30] have been used in prior approaches to facilitate program understanding and detect memory bugs. We employ a unified memory graph representation [34] for capturing the evolution history of the memory graph and hence that of the data structure(s) it represents. From the memory graph at a given program execution point, we can derive its form at all earlier execution points. The graphs provide mappings between changes in the memory graph and the source code statements that caused the changes to assist with fault location. A Memory Graph $\mathbf{MG} = (\mathbf{V}, \mathbf{E})$ is defined as follows:

•$\mathbf{V}$ is a set of nodes such that each node $v \in \mathbf{V}$ consists of $\langle T_v; S_v; H_v \rangle$, where $H_v$ is a set of heap addresses $\{h_v^1, h_v^2, \ldots\}$ in ascending order that the node represents. $T_v$ is the timestamp at which the node was created, i.e., an integer which marks the order of events in the memory graph. The timestamp is initialized at start of memory graph construction and is incremented with each change to the graph. $S_v$ is the *source code statement* which led to creation of the node $v$; the statement is identified by its location in the source code, i.e., <file name:line number>. The memory graph also consists of data nodes which contain scalar information and are used to show the value stored at a heap address. An edge from a heap address to the data node implies that the data value in the data node is stored in the heap address.

•$\mathbf{E}$ is a set of directed edges such that edge $e \in \mathbf{E}$ is represented as $H_u.h_u^i \rightarrow v$ and has a label $\langle T_e : S_e \rangle$, where $H_u.h_u^i$ is the $i^{th}$ heap address inside node $u$ and stores a pointer to the heap address of node $v$; $T_e$ is the timestamp at which the edge was created; and $S_e$ is the *source code statement* that created the edge. An edge may also point to a data node that corresponds to non-heap data, or to NULL. A heap address may point to different nodes at different execution points. The memory graph captures all the corresponding edges and the edge with the largest time stamp represents the current outgoing edge.

Figure 7 (b) shows the unified MG that results from executing the Figure 3 program from point 1 to point 4. Each allocation site, in this case each tree element, corresponds to a node in the memory

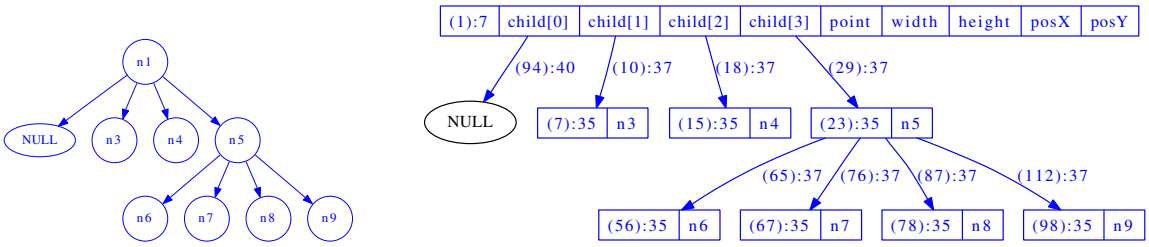



**Figure 7.** Memory Graph (a) and Unified Memory Graph representation (b) for memory graph at execution point 4 in Figure 3.

graph. Each node consists of a list of heap addresses; we omit them in Figure 7 for simplicity. Each node and edge has a time stamp and a statement number associated with it. The top node $n1$ in the quad tree is: ⟨*T= (1); S= 7; H= {child[0], child[1], child[2], child[3], point, width, height, posX, posY}*⟩ which means the node was created at time stamp 1 by statement number 7 and has a fields *child[0], child[1], child[2], child[3], point, width, height, posX, posY*. Note that in the actual memory graph these fields will be heap addresses corresponding to the listed field labels. The edge from node $n5$ to the node $n6$ has label ⟨*T=(65):S=37*⟩ which means the edge was created at time stamp 65 by statement number 37. The time stamp information enables us to extract the memory graph of the program at any previous execution point (1, 2, and 3) from the unified MG at program point 4.

**Algorithm 2** Memory Graph Rollback

1: $MG_{result<V_r,E_r>} \leftarrow NULL$
2: **INPUT:** Memory Graph $MG_f < V_f, E_f >$ at time stamp $ts_{final}$, target time_stamp $ts_t : ts_t \leq ts_{final}$
3: **Graph_Reconstruct()**
4: **for** All nodes $v \in V_f$ and Edges $e \in E_f$ having time_stamp $> ts_t$ **do**
5: $V_r = V_f - v$
6: $E_r = E_f - e$
7: **end for**
8: **for** All Heap address $h$ such that edge $e$:$h \rightarrow v$ was deleted **do**
9: **if** $h$ has an out going edge **then**
10: Set the edge with highest time stamp out going of $h$ as the current edge
11: **end if**
12: **end for**
13: **return** $MG_{result}$

Algorithm 2 gives a simplified version of rollback process given in [34]. The procedure for *rolling back* the MG to a previous timestamp, i.e., rolling back to the MG at time stamp $t$ given a memory graph at time stamp $t'$ such that $t < t'$, has two steps. In the first step (line 4-7), all the nodes and edges having time stamp larger than the target time stamp are removed from the graph. In the second step(line 8-12), for the deleted destination nodes, among the outgoing edges from the source address, the edge with the highest time stamp is set as the current edge.

***Incremental constraint checking.*** Constraint checking is a critical operation, and containing the cost of checks on large data structures allows our approach to scale well. We reduce the cost of checking via the use of incremental on-demand checks: we keep a mapping between constraint atoms and dependent nodes (nodes involved in the constraint); when the MG is modified, we map modified nodes to affected constraint atoms and invalidate those atoms.

***Efficient traceback.*** When tracing back for faults, performing rollback and constraint checking in a naïve way would significantly affect scalability due to the high cost of constraint checking. We use two techniques to make trace back efficient. First, knowing which data structure is corrupted, we limit our constraint matching (during trace back) to the data structure in question. Second, we use *modification prediction* to check if a rolled back operation can contribute to the inconsistencies present in the pending inconsistency list. We perform the rollback and consistency check on the MG only when the prediction for the operation returns true. Modification prediction is based on the spatial locality. We can predict that an operation $O$ operating on a sub-graph $G_o$ will not affect an inconsistency $\langle c, G \rangle$, based on the properties of constraint $c$ and the relationship between sub-graphs $G_o$ and $G$. For example, an operation $O$ of creating edge $n_1 \rightarrow n_2$ will not affect inconsistency $\langle c, n_3 \rangle$, where constraint $c$ checks the value of a node field. We create a list of nodes (dependency list) for each inconsistency $\langle c, G \rangle \in P$ (set $P$ in the algorithm 1) based on constraint $c$. Modification in the a node present in the dependency list can affect the inconsistency. A check is performed if any of the inconsistent nodes for the pending inconsistency list (set $P$) is modified by the operation or is dependent on any of the nodes (i.e., in its dependency list) modified by operation $O$. Algorithm 1 is modified to directly roll back before an operation only when the modification prediction returns true for the operation.

## 6. Evaluating Fault Location

Next we evaluate the precision and cost of our fault location technique. Our implementation consists of a binary instrumenter, constraint matcher generator, memory graph constructor and a fault locator. The *binary instrumentation* is based on Pin-2.6 [25] – only allocation calls and memory writes are instrumented. To reduce the size of the execution trace, at runtime, Pin keeps track of allocated heap addresses and outputs only the instructions which write allocated heap addresses. The execution trace contains the timestamp information, and the statement identifier for each of allocation and memory writes. We have used the source code location of memory allocation as a unique identifier of the data structure type for each memory graph node as each allocation site belongs to a unique data structure. The execution trace drives the construction of the memory graph. The *constraint matcher generator* produces the constraint matcher based on the input constraint specifications. If an error is detected during constraint matching, the *fault locator* searches for the root cause and outputs a list of candidate faulty statements. Measurements were performed on an Intel Core 2 6700 @ 2.66GHz with 4 GB RAM, running Linux kernel version 2.6.32. All benchmarks were written in C.

***Precision of Fault Location.*** The strength of our technique lies in its ability to carry out highly precise fault location using a single test case during which constraints are violated – note that program

**Table 3.** Precision and overhead of fault location.[1]

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| **Data structure** | **Lines of code** | **Statements examined** | | | **Program execution time (ms)** | | | **Const. Match. time** (ms) | **Fault Loc. time** (sec) |
| | | **Ours** | **Dyn. Slice** | **Tarantula** | Original | Null Pin | Instrume. | | |
| Circular linked list | 160 | 6 | 7 | 7 | 0.5 | 645 | 664 | 1 | 0.75 |
| Ordered list | 172 | 2 | 20 | 7 | 0.9 | 638 | 677 | 1 | 0.02 |
| Doubly-linked list | 203 | 5 | 38 | 17 | 0.7 | 653 | 673 | 1 | 0.25 |
| Quad tree | 294 | 1 | 58 | 9 | 2.6 | 713 | 885 | 36 | 9.22 |
| AVL tree | 243 | 4 | 9 | 10 | 1.6 | 739 | 864 | 25 | 1.18 |
| B tree | 405 | 6 | 8 | 8 | 1.5 | 722 | 767 | 4 | 73.39 |
| Red-Black tree | 395 | 10 | 24 | 13 | 0.4 | 661 | 711 | 31 | 72.21 |
| Leftist heap | 274 | 1 | 28 | 11 | 0.4 | 664 | 720 | 27 | 2.21 |
| Bipartite graph | 284 | 2 | 32 | 8 | 8.2 | 659 | 748 | 2 | 3.32 |

execution may or may not lead to a program crash. Table 3 presents the results of our fault location technique for implementations of several data structures whose program sizes are given in the second column. In each case, the data structure was first initialized to a base size of 1,000 nodes. Next, 500 operations (inserts and deletes) were performed on the data structure along with random injection of 10 faults. In each case, the injected fault leads to violation of the data structure constraint. The program crash point was used as *C-point* in cases where program crashed. The end of execution was used as *C-point* in cases where program terminated normally with wrong output. Column 3 shows the number of faulty statements our technique detected. The numbers of faulty statements range from 1 to 10 while program sizes range from 160 to 405 lines of code. This indicates that our technique narrows down the fault to a very small code region. In all cases the fault was captured by the identified faulty statements. We also computed the dynamic slice of the faulty statement instances. The fourth column shows the number of distinct program statements in the largest dynamic slice among faulty instances' slices. These numbers show that without the knowledge of data structure constraints, the set of potentially faulty statements identified can be quite large ($\geq$ 20 for 6 programs). The fifth column reports the number of statements that must be examined to find the faulty statement using the ranking produced by Tarantula [18]. Tarantula is a statistical technique that uses information from multiple runs on different inputs – we used 1 failing run and 9 successful runs in this experiment. The results show that our approach requires fewer statements to be examined although it is based upon a single run as opposed to Tarantula that used 10 runs.

***Overhead of Fault Location.*** Table 3 column 6-10 gives the overhead in terms of time and space for using our fault location implementation. The 6th column shows the execution time of the buggy program. The 7th and 8th columns show the execution times when the benchmarks run under Pin without instrumentation ("Null Pin" column) and with instrumentation ("Instrumented" column). Although instrumentation overhead is significant, it is acceptable for debugging purposes. The 9th column shows that the time to perform constraint matching (after error detection) is typically a second or less. The 10th column shows the time for fault location—this is very efficient, 0.2–73 seconds in our tests.

## 7. Experience with Real Programs

For each application we defined consistency constraints for the main data structures. Next, we used fault injection to simulate common programming errors that lead to data structure corruption. Then, using our technique, we identified the buggy statements in the program. Table 4 shows our findings. The execution times are for the buggy version of the programs. Column 4 gives the number of faulty statements our technique found. Our fault location technique captured the faulty statement precisely (less than 5 statements) in all cases. In all applications, the program crash point was used as the single, automatically-inserted *C-point*, hence programmer effort was limited to specifying constraints and indicating allocation sites for the data structures.

***ls.*** GNU *ls* [5] lists information about files including directories. The program source code consists of 4,000 lines of C code. It uses a linked list internally to store information about the remaining directories when run in recursive mode. We inserted a bug in the code where the next pointer's value is a non-NULL non-heap address. Due to this bug the program crashes. The violated constraint here is that the next pointer needs to be either a heap address or NULL. As input, we used the GNU coreutils-8.0 source code directory. Our technique traced this fault to 4 statements.

***403.gcc.*** A benchmark program from SPEC CINT 2006 benchmark suite [6], 403.gcc is based on Gcc version 3.2 set to generate code for an AMD Opteron processor. The program uses splay trees, a form of binary search trees optimized to access to recently-accessed elements (splaying is the process of rotating the tree to put a key to the tree root).

To inject a bug, we replaced the right rotate function of the tree by left rotate function. We used the SPEC training input for this experiment, which leads to program crash. The violated constraint here is the binary search tree invariant: *Key(root)* $>$ *Key(left child)* and *Key(root)* $<$ *Key(right child)*. Our method traced the fault to the 1 statement.

***464.h264ref.*** This benchmark program from the SPEC CINT 2006 benchmark suite [6] is a reference implementation of Advanced Video Coding, a video compression standard. The benchmark uses a pointer-based implementation of a multidimensional array, with each dimension being a level of a full and complete tree.

To inject a bug we set an internal pointer to NULL which caused a crash. The SPEC test input was used in this experiment. The violated constraint here, based on the OUTDEGREE attribute, is that the tree is full and complete. Our fault location method traced the fault to 1 statement.

***GNU Bison.*** Bison(3.0.4) [4] is a general-purpose parser generator that converts an annotated context-free grammar into a parser. The application uses a graph to store the symbol table where each node is a token or a grammar non-terminal and the node attributes are assigned accordingly.

Our bug injection simulates a programming error where wrong attributes are assigned to nodes, leading to a crash. We used the C language grammar file as input. The violated constraint here is the

[1] First row in Table 3 and Table 4 is column numbering.

**Table 4.** Experience with real world programs.[1]

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| **Program** | **Lines of code** | **Data Structure** | **Statements Examined** | **Program execution time (sec)** | | | **Const. Match. time** (sec) | **Fault Loc. time** (sec) |
| | | | | Original | Null Pin | Instrume. | | |
| ls | 3.5K | Linked List | 4 | 0.19 | 1.10 | 1.36 | 0.003 | 0.05 |
| 403.gcc | 365K | Splay Tree | 1 | 0.04 | 5.83 | 10.76 | 0.028 | 0.23 |
| 464.h264ref | 32K | Multidimen. Array | 1 | 15.08 | 38.17 | 2703.69 | 0.008 | 0.29 |
| Bison | 17K | Graph | 3 | 0.18 | 1.48 | 4.99 | 0.426 | 16.48 |

correctness of the node attributes. Our method traced back the fault to 3 faulty statements.

**Scalability of the technique.** There are two sources of time overhead incurred by our technique, first the time taken in collecting the execution trace and second the time taken during the trace back. We now explain why both these slowdowns are unavoidable and we believe the overhead is acceptable. First, dynamic analysis (collecting the execution trace) is inherently slow. Table 3 column 7 and Table 4 column 6 list the cost of running application under Pin without any instrumentation which itself is a significant slowdown. We only instrument allocation/deallocation calls and references to allocated regions (section 6, paragraph 1). This limits instrumentation cost while still capturing data structure faults.
Second, the alternative to automatically search the program execution trace is manually narrowing down the fault by running the application multiple times. We have introduced a number of optimizations to speed up the trace back as explained in section 5 .

Programmers can use our tool for debugging larger programs by capturing the execution trace for just the relevant sections of program execution hence limiting the search space. As shown in the evaluation, our tool can handle large enough search spaces for practical debugging purposes.

Our technique is easily applicable to parallel programs, as that would only require modifications to the Pin-based tracing mechanism. While tracing a multi-threaded program can potentially increase the overhead, this problem can be abated using selective record and replay, e.g., PinPlay [29].

## 8. Related Work

Our system uses a novel fault location technique and a new specification language. In this section we review previous work related to specification languages, automatic fault location, specification-based testing and uses of memory graph.

**Fault location.** Various general approaches for fault location that do not rely on data structure information have been developed (e.g., statistical techniques [7, 23, 31], dynamic slicing [21, 36], or combinations of the two [37]). Statistical techniques require running the program on a suite of test cases. Our approach is more comparable with dynamic slicing as they both perform debugging by analyzing a single program run during which a fault is encountered. Jose and Majumdar [19] use MAX-SAT solvers while Sahoo et al. [32] use dynamic backward slicing and filtering heuristics for software fault location. The focus of our work is specifically on data structure errors. Taylor and Black [35] examine a number of structural correction algorithms for list and tree data structures. Bond et al. [9] track back undefined values and NULL pointers to their origin. In contrast, our system concentrates on fault location for violation of high-level data structure properties. We consider the concept of temporary violation of high-level data structure constraints. This helps us locate errors early and trace them to faults precisely.

**Specification-based error detection.** A wide range of specification techniques have been used to specify correctness properties from which monitors for runtime verification of those properties are generated. For example, MOP [11] allows correctness properties to be specified in LTL, MTL, FSM, or CFG; though MOP is geared more towards verifying protocols or API sequences rather than data structures. Similarly, a specification technique for easily handling memory bugs has also been developed [36]. Our work focuses on data structure correctness and hence is closest to Archie [12] and Alloy [17]. Our data structure specification language differs from languages used by Archie and Alloy: their modeling languages let the developer specify high-level design properties in terms of a model while our language enables developers to express high-level data structure properties in terms of the memory graph of the program. This makes specification writing in our language easy and concise. Berdine et al. [8] and Chang and Rival [10] have introduced predicate-based specification languages for shape analysis. Customized versions of these languages can be used for our technique. Malik et al. [26], Juzi [15] and Demsky et al. [12, 13] use constraint-based error detection for data structure repair while Gopinath et al. [16] combine spectra-based localization with specification matching to iteratively localize faults. Malik et al. [27] proposed the idea of using using data structure repair for repairing faulty code. Jung and Clark [20] applied invariant detection on memory graphs to identify the data structures used in the program. Our system uses a similar concept of matching constraints over program state to detect violations. Our approach goes one step further and maps the dynamic data structures constraint violations at runtime back to the source code. Also, we give a method for incremental matching based on the timing information which makes it feasible for the developers to perform constraint checks more frequently. This leads to *early* detection of errors. Zaeem et al. [28] use history of program execution (field reads and writes) for data structure repair but do not capture structural information.

DITTO [33] performs incremental structure invariant checks for JAVA. It incurs additional overhead for storing all the computations from the last check, as these computations are later used for the incremental checking. Our system reuses the time stamp information from the memory graph and stores only the timestamp of the previous check for incremental constraint matching.

## 9. Conclusions

We have presented an approach for specifying and verifying consistency constraints on data structures. The constraints are verified during execution using an appropriately constructed memory graph, as well as a suite of optimized checks. If the specification is violated, a fault location component traces the inconsistency to faulty statements in the source code. Experiments with specification and fault location on several widely-used data structures indicate that the approach is practical, effective and efficient. Most importantly, our constraint specification language is both expressive and easy to use.

## Acknowledgments

This work is supported by NSF Grants CCF-1318103, CCF-1524852, and CCF-1149632 to UC Riverside.

## References

[1] Gnome bug tracker. https://bugzilla.gnome.org/. Accessed: 03/2014.

[2] Kde bug tracker. https://bugs.kde.org/. Accessed: 03/2014.

[3] Open office bug tracker. https://issues.apache.org/ooo/. Accessed: 03/2014.

[4] Bison-gnu parser generator. http://www.gnu.org/software/bison/.

[5] Gnu core utilities. http://www.gnu.org/software/coreutils/.

[6] Spec cint2006 benchmarks. https://www.spec.org/cpu2006/cint2006/.

[7] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Dependable Computing, 2006. PRDC '06. 12th Pacific Rim International Symposium on*, pages 39–46, Dec 2006. doi: 10.1109/PRDC.2006.18.

[8] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings*, chapter Shape Analysis for Composite Data Structures, pages 178–192. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73368-3. doi: 10.1007/978-3-540-73368-3_22. URL http://dx.doi.org/10.1007/978-3-540-73368-3_22.

[9] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: Reporting the origin of null and undefined value errors. In *OOPSLA*, pages 405–422, 2007. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297057. URL http://doi.acm.org/10.1145/1297027.1297057.

[10] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In G. C. Necula and P. Wadler, editors, *POPL*, pages 247–260. ACM, 2008. ISBN 978-1-59593-689-9. URL http://dblp.uni-trier.de/db/conf/popl/popl2008.html#ChangR08.

[11] F. Chen and G. Rosu. Mop: An efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, 2007.

[12] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003. ISBN 1-58113-712-5. doi: 10.1145/949305.949314. URL http://doi.acm.org/10.1145/949305.949314.

[13] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, pages 176–185, 2005. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062499. URL http://doi.acm.org/10.1145/1062455.1062499.

[14] B. Demsky, C. Cadar, D. Roy, and M. Rinard. Efficient specification-assisted error localization. In *WODA'04*.

[15] B. Elkarablieh and S. Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 855–858, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368222. URL http://doi.acm.org/10.1145/1368088.1368222.

[16] D. Gopinath, R. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE'12*.

[17] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. ISSN 1049-331X. doi: 10.1145/505145.505149. URL http://doi.acm.org/10.1145/505145.505149.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM ASE*, ASE '05, pages 273–282, New York. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101949. URL http://doi.acm.org/10.1145/1101908.1101949.

[19] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. PLDI'11, pages 437–446.

[20] C. Jung and N. Clark. Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *IEEE/ACM MICRO*, pages 56–66, 2009. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669122. URL http://doi.acm.org/10.1145/1669112.1669122.

[21] B. Korel and J. Laski. Dynamic program slicing. In *Information Processing Letters*, volume 29, pages 155–163, 1988.

[22] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387884.

[23] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug location. In *PLDI*, 2005.

[24] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000. ISBN 0201657686.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[26] M. Malik, J. Siddiqi, and S. Khurshid. Constraint-based program debugging using data structure repair. ICST, pages 190–199, 2011.

[27] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *ASE*, pages 620–624, 2009.

[28] R. Nokhbeh Zaeem, D. Gopinath, S. Khurshid, and K. S. McKinley. History-aware data structure repair using sat. TACAS'12, pages 2–17.

[29] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772958. URL http://doi.acm.org/10.1145/1772954.1772958.

[30] E. Raman and D. I. August. Recursive data structure profiling. In *MSP Workshop*, pages 5–14, 2005. ISBN 1-59593-147-3. doi: 10.1145/1111583.1111585. URL http://doi.acm.org/10.1145/1111583.1111585.

[31] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *IEEE ASE*, pages 30–39, 2003.

[32] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *ASPLOS'13*.

[33] A. Shankar and R. Bodík. Ditto: automatic incrementalization of data structure invariant checks (in java). In *PLDI*, 2007.

[34] V. Singh, R. Gupta, and I. Neamtiu. Mg++: Memory graphs for analyzing dynamic data structures. In *IEEE SANER*, 2015.

[35] D. Taylor and J. Black. Principles of data structure error correction. *IEEE Trans. on Computers*, C-31(7):602–608, 1982. ISSN 0018-9340. doi: 10.1109/TC.1982.1676057.

[36] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. AADEBUG'05, pages 33–42.

[37] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, 2006.