Generating Test Data for Functions with Pointer Inputs

Srinivas Visvanathan
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721
srini@cs.arizona.edu

Neelam Gupta
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Abstract

Generating test inputs for a path in a function with integer and real parameters is an important but difficult problem. The problem becomes more difficult when pointers are passed as inputs to a function. In this case, the shape of the input data structure as well as the data values in the fields of this data structure need to be determined for traversal of the given path. The existing techniques to address this problem are inefficient since they use backtracking to simultaneously satisfy the constraints on the pointer variables and the data values used along the path.

In this paper, we develop a novel approach that allows the generation of the shape of an input data structure to be done independently of the generation of its data values so as to force the control flow of a function along a given path. We also present a new technique that generates the shape of the input data structure by solving a set of pointer constraints derived in a single pass of the statements along the path. Although simple, our approach is powerful in handling pointer aliasing. It is efficient and provides a practical solution to generating test data for functions with pointer inputs.

Keywords - *Test data generation, path testing, dynamic data structures, iterative relaxation methods.*

1 Introduction

Generating test data to force the control flow of a function along a given path in a function is a challenging problem. The presence of pointers as function input parameters introduces additional difficulties for test data generation. A pointer represents a pair of data items, i.e., a data value and its address. Although pointer inputs for programs are not meaningful, they can be input parameters to the functions to be tested at unit level testing. For integer or real inputs, generating test data involves generating the integer and floating point values that should be used as input to a function. However, the pointer addresses are allocated dynamically during program execution. The test data generation for functions with pointer inputs does not refer to generating the pointer addresses. Instead, the test data generation problem in this case is to generate the shape of the input data structure and the values in the data fields of this data structure so that the given path is executed by the function.

If the input parameter to a function is a pointer to an integer or a real value, the existing test data generation methods [1, 2, 3, 4, 5, 6, 7, 8] can be used to generate the desired integer or real value. If the input pointer points to a collection of related items as in a record, the above mentioned existing test data generation techniques can be used by treating each field in the record as a separate input variable for test data generation.

However, if the input pointer points to a dynamically linked data structure such as a linked list or a binary tree, not only the values in the fields of the input data structure should be determined, but also the shape of the data structure required to force execution through the given path must be determined. We illustrate this with a simple code segment shown in Figure 1. We represent a test path by an ordered list of statements that must be executed to traverse the path. Consider a test path P=$\{0, P1, 1, 2, P2, 3, 4\}$ in Figure 1 that requires both the predicates $P1$ and $P2$ to evaluate to $true$. The input data structures shown in Figures 2(a) and 2(b) will execute P whereas the input data structure in Figure 2(c) will not execute P. Note that there are several data structures that will execute this test path. Similarly, there are several data structures that will not execute this test path.

Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE'02)
1527-1366/02 $17.00 © 2002 **IEEE**

```
node=record;
      int data;
      node * left;
      node * right;
end=record;
0:    func(node * root);
P1:  if (root→right ==root→right→right)
1:          printf("P1 true");
2:    endif
P2:  if (root→left→left == root→left→right)
3:          printf("P2 true");
4:    endif
```

Figure 1. A code segment using pointers.

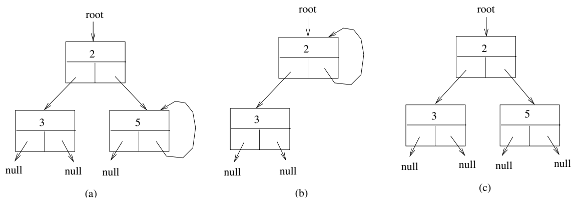


Figure 2. Input data structures for function $func$ given in Figure 1.

Let us now modify the function $func$ in Figure 1 by appending the following three lines (line numbers $P3$, 5 and 6) at the end of the function.

```
P3:  if (root→data < root→right→data)
5:          printf("P3 true");
6:    endif
```

If the test path now requires all three predicates $P1$, $P2$ and $P3$ to evaluate to $true$, then the data structure in Figure 2(a) will execute the test path whereas the data structures in Figures 2(b) and 2(c) will not execute the test path. Thus, test data generation for functions with dynamic data structures as input requires, (i) the generation of a suitable shape (i.e., the number of distinct nodes and how they are connected) of the input data structure and (ii) the generation of the values in the fields of the data structure, so as to execute the given test path.

In [7], Korel presents a backtracking based approach for simultaneously generating the shape and the values in the fields of the data structure for functions with pointer inputs. The method starts with an arbitrary input data structure with the fields initialized with arbitrarily chosen values. It varies the value of only one input variable at a time to satisfy a branch predicate along the test path and uses backtracking. Although this approach is an important contribution to automated test data generation, it is inefficient since it tries to the generate the shape as well as the values simultaneously. If it has to backtrack to some predicate because of an incorrect choice about the shape at that point, the values in the data structure that were generated subsequent to this incorrect choice also become useless. In addition, the backtracking can be extensive in presence of pointer aliasing.

In this paper, we develop a new approach that first generates a suitable shape for the input data structure and then generates the data values in the fields of the data structure to force execution through the test path. The reason for this two phase approach is as follows. The constraints on the pointers deal with addresses of memory locations used by the statements along the path whereas the constraints on the data values such as integer and real values deal with the actual values being used in the computation. Therefore, the solutions of these two types of constraints are in two different domains, namely the $address\ domain$ and the $value\ domain$. The solution in address domain determines the shape of the input data structure and the solution in the value domain determines the values in the data fields of the input data structure. Since these are disjoint types of domains, the constraint sets in these two domains are solved separately. However, for the same test path in a function, different sets of pointer constraints and hence different shapes can be generated by different techniques. For example one technique may generate the shape in Figure 2(a) and another may generate the shape in Figure 2(b), both satisfying the pointer constraints $P1$ and $P2$ in the function in Figure 1. But only the shape in Figure 2(a) satisfies the constraint $P3$ on the data values in the nodes of the data structure. Therefore, it is not obvious whether (and if so, how) a suitable shape for the traversal of a given path can be generated independently of the values in the data fields of the input data structure.

A comparison of shapes in Figure 2(a) and 2(b) provides an insight into our approach. Although both the shapes satisfy pointer constraints $P1$ and $P2$, the shape of input data structure in Figure 2(a) is *less restrictive* than the shape in Figure 2(b). The shape in Figure 2(b) coalesces the two nodes ($root$ node and its right child) of the shape in Figure 2(a) into a single node ($root$ node). Therefore, it restricts the data value in the right child of the root node to be identical to the data value in the root node. However, the shape in Figure 2(a) allows different or identical values to be stored in these two nodes. So, the shape in Figure 2(a) will be suitable whether the constraints on the data values in these two nodes require them to have identical values or different values. Therefore, a technique that generates the *least restrictive shape* (i.e., with the maximum number of nodes that can be referenced by the statements along the path) satisfying the constraints on the pointer fields used along the path does not need to take into consideration the constraints on the data values in the nodes. In constrast, a technique that attempts to generate a suitable shape with fewer number of nodes needs to consider the constraints on the data values in the nodes to determine whether some nodes can be coalesced or not. Our shape generation algorithm constructs the least restrictive shape satisfying the pointer constraints along the test path. It constructs the shape in Figure 2(a) for this example. Note that some of the pointer addresses in the least restrictive shape may never be referenced by the statements along the path. The least restrictive shape generated by our algorithm sets these unreferenced pointers in the shape to $NULL$ to avoid any uninitialized pointers in the generated shape. Thus, our approach separates the problem of generating a suitable shape for the input data structure from the generation of desired values for the integer and real fields in the data structure.

In this paper, we also present a new algorithm to generate the least restrictive shape of the input data structure for the traversal of a given path in a function. Our algorithm collects the constraints on the pointer fields in the desired input data structure in a single pass of the statements along the path. These constraints are solved efficiently using a technique developed in this paper. The solution of the constraints gives the relative (not absolute) addresses for the pointer fields. These relative addresses are then used to generate the shape of the data structure that satisfies the constraints on the pointer fields used by the statements along the test path. Finally, the desired values for the integer and real data fields in the above data structure are generated using the test data generation techniques described in [5, 6].

An advantage of our approach is that the constraints on pointer fields are simple and can be easily solved to compute a consistent solution or detect inconsistency. An inconsistency in these constraints implies a contradiction in the constraints imposed on the pointer references by the branch predicates along the path. Such a path is infeasible and no shape can be generated to execute this path. Thus our algorithm always either generates a suitable shape of the input data structure for the given path or it guarantees that no such shape exists for the given path. It is efficient as it always generates the least restrictive shape or guarantees infeasibility in a single pass of the statements along the path.

Note that if a suitable shape of the input data structure can be generated for a given path, it does not imply that the path is feasible. It only implies that the pointer constraints along the path are consistent. However, a path with consistent constraints on pointer references can still be infeasible due to some conflicting constraints along the path that do not refer to the pointer fields of the input data structure.

Another important reason for generating the shape first and values of data elements later, is to avoid aliasing problems. The test path may contain statements where two pointers are used to dereference data elements e.g. $p1 \rightarrow data$ and $p2 \rightarrow data$. If $p1$ and $p2$ were the same, the data element being referenced would also be the same. Otherwise, the data elements being referenced would be different. We cannot predict anything about aliasing until we know the shape of the data structure.

We have implemented our shape generation technique. Our experiments show that our approach is efficient in generating a suitable shape for the input data structure. The important contributions of this paper are:

- A novel approach that first generates a suitable shape of the input data structure and then generates the integer and real values in the data fields of the data structure.
- A new algorithm for generating constraints on the pointer addresses in the input shape by a single pass of the statements along the test path.
- An efficient technique to solve the pointer constraints arising in shape generation.
- It either generates a suitable shape for input data structure or guarantees no such shape exists.
- It handles pointer aliasing efficiently.
- It is easy to automate and provides a practical solution to shape generation for pointer inputs to functions.

The organization of this paper is as follows. An overview of our approach is presented in section 2. The algorithm for shape generation is described in section 3. The implementation and experiments are discussed in section 4. Related work is discussed in section 5. The important features of our method are summarized in section 6.

2 Overview

Given a test path P in a function F that accepts linked data structures as part of its input, we derive suitable shapes for the input data structures so as to satisfy the constraints imposed by the pointer references along the path P. The input to our shape generation method is the list of statements along P and the output is a function F' that constructs the required data structures using a series of node allocation and assignment statements. F' also sets up the input pointer parameters of F with the addresses of appropriate nodes of the input data structures. After generating the suitable shapes for input data structures, the appropriate values of data fields in the nodes of these data structures (e.g. $p1 \rightarrow data$) and other data arguments of F are generated using the technique described in [5, 6] so that P is executed with this generated input.

We use the following terminology to refer to pointers. We call the pointer parameters of F as $argument\ pointers$. The statements along P may refer to a pointer variable such as *ptr*, which may be a local pointer variable or an argument pointer. We call these *simple* pointers. The statements may also use the pointers that are dereferenced using other pointers e.g., $ptr \rightarrow next$. We call them *complex* pointers. Note that a complex pointer may be dereferenced using a simple pointer e.g. $ptr \rightarrow next$ or dereferenced using another dereferenced pointer e.g., $ptr \rightarrow next \rightarrow prev$.

Now we give an overview of our technique for generating a suitable shape for an input data structure of a function F for execution of a given path P in F. Our shape generation method essentially consists of the following three steps:

- Collect constraints on pointer addresses used by statements along the test path P in the function F.
- Solve the above constraints on pointer addresses.
- Use the above solution to output the function F' that constructs the desired data structure.

We illustrate our approach by generating the shape of input data structure for traversal of path $P = \{1, P1, 2, P2, 6, P1, 9, 10, 11\}$ in the function $func1$ in Figure 3. It takes as input a pointer argument *root* pointing to a linked data structure. We assume that loops are unrolled at the time of defining the test path.

2.1 Collection of Constraints

By examining the statements along the test path, we derive a set of constraints in terms of node-addresses that must be satisfied in order for the linked data structure passed as input to execute the test path. These constraints do not refer to actual addresses used at run time. Instead they capture the relationships between the node-addresses in the data structure. In other words, these constraints define the shape of the input data structure. For example, $A = B$ indicates that both node-addresses A and B should be assigned the same address since they refer to the same node in the data structure. The actual address assigned is not important as long as this constraint is satisfied. We use the domain of $whole\ numbers$ as our address space. The $NULL$ pointer is assigned value 0 in our address space. So, if there is a pointer constraint $C = 0$, no memory allocation should be done in the data structure for this node-address. Instead, it represents a $NULL$ pointer in the data structure. We classify the statements into the following types to derive the constraints on the node-addresses referred by the statements along P.

1. Statements that *do not* use any pointer variables e.g., $x = 2$. These statements do not access the data structure and we ignore them.
2. Statements that *do* use pointer variables e.g., all statements along P in Figure 3(a) fall into this category. Since these statements refer to elements of the data structure, we try to derive constraints from them. These statements are further classified as following:
   (a) Statements in which the actual element being operated on is a pointer e.g. $curr = prev \rightarrow next$. The actual elements operated on in these statements are memory address values in the pointer fields of nodes and pointer variables.
   (b) Statements in which the actual element being operated on is not a pointer e.g. $prev \rightarrow next \rightarrow data = d$. These statements also provide information about the shape of the data structure since a node that is dereferenced cannot be $NULL$.

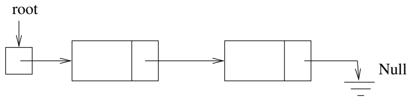
Note that in a function that uses pointers to access dynamic data structures, the only operations that are performed on pointer variables are (i) pointer assignment, (ii) pointer comparison (either: == or !=) in a branch predicate (also called an assertion), (iii) dereferencing and (iv) alloca-

```
        struct node
        {   int data;
            node *next; }
0:      void func1(node *root, int d)
1:          node *prev = root, curr;
P1:         while (prev → next != NULL)
2:              curr = prev → next;
P2:             if (curr→ data == d)
3:                  prev→next = curr→ next;
4:                  free(curr);
5:              else
6:                  prev = curr;
7:              endif
8:          endwhile
9:          prev→ next = malloc(node);
10:         prev→ next→ data = d;
11:         prev→ next→ next = NULL;
12:     endproc
```

(a)

```
0:  void F′(node **root)
1:      p1 = malloc(node);
2:      p2 = NULL;
3:      p3 = malloc(node);
4:      p4 = NULL;
5:      p5 = NULL;
6:      p6 = NULL;
7:      p1→next = p3;
8:      p3→next = p5;
9:      *root = p1;
10: endproc
```

(b)



(c)

Figure 3. Test path P =$\{1, P1, 2, P2, 6, P1, 9, 10, 11\}$ in the function $func1$, the output function F' generated by our method, and the corresponding data structure generated by F'.

tion/deallocation. [1] We now show how the constraints are derived from these four operations. Let p and q be two pointer variables that point to the *node type* as defined for the function $func1$ in Figure 3. Let X and Y be the node-addresses assigned to two nodes referenced by p and q. Assume that the next field in the node corresponding to X contains the address of the node Y. This information is represented in a table shown below. We refer to the table below as $initial$ table later in this section to illustrate the effect of processing various statements on the contents of this table.

NA	new	del	ptrVar	next	Constraints
X			p	Y	
Y			q		

In this table (called the $Node\ Address\ Table$), each row stores the information about a node-address. The column labeled *NA* lists the node-addresses. If a node-address is generated by a $malloc$ statement along P, the column labeled *new* is $marked$ in row for this node-address. If a node is deleted by a statement along the path, the column labeled *del* is marked for the row corresponding to this node-address. If a statement on the path refers to a node-address with *del* column marked, a message is displayed that the test path refers to a deleted node and the shape generation process terminates. If the node corresponding to a node-address is being referenced by a pointer variable e.g., p holds the address of the node corresponding to X, then the variable p is listed in the column named $ptrVar$ in the row for X. The $ptrVar$ column for a node-address may be empty, if no pointer is referencing the node or may contain more than one pointer if multiple pointers are pointing to the node. The column labeled $next$ corresponds to the $node\ {*}next$ pointer field defined in $struct\ node$ in Figure 3(a). This field holds the node-address being referenced by the $next$ pointer field. If a node has two pointer fields, say $left$ and $right$, then there will be two columns in place of the $next$ column. Thus the presence of this column varies with the definition of the *node-type* of the input linked data structure. Therefore, a separate table is maintained for each argument pointer pointing to a different *node-type*. However, a single table is sufficient when several argument pointers point to the same *node-type*. The last column called $Constraints$ contains the constraints on each node-address and they are generated as the statements are examined. The NULL pointer is assigned a node-address equal to 0 in our domain of relative node-addresses. We use the notation based on column name such as $Constraints(R)$, to denote the entry in the $Constraints$ column of a row with the node-address R. We use similar notation to refer to entries in other columns of a node-address.

Our method updates the Node Address Table as it scans the statements along the test path. The constraints on node-addresses are collected from the final table obtained after scanning all statements along the path. For each type of pointer operation

[1] This algorithm is not designed to handle functions that use pointers to access arrays. In such functions other operations like ++, > etc. may be applied on pointer variables. [5] explained how arrays may be handled

we show how the contents of the table are updated.

Assignment (p = q): As a result of this assignment, both p and q point to the same node i.e., node corresponding to Y, and the *initial* table is changed to:

NA	new	del	ptrVar	next	Constraints
X				Y	
Y			p, q		

Inequality Comparison(p != q): If an assertion *p* != *q* is present in a statement, a constraint $X \neq Y$ for the node-addresses corresponding to *p* and *q* is added to the rows for X and Y, and the *initial* table is changed to:

NA	new	del	ptrVar	next	Constraints
X			p	Y	$X \neq Y$
Y			q		$X \neq Y$

Equality Comparison(p == q): If an assertion *p* == *q* is present in a statement, the two nodes referred by *p* and *q* should be same for traversal of P. Hence we eliminate one of the node-addresses involved and make both *p* and *q* refer to the same node-address. If Y is eliminated in the *initial* table, every occurrence of Y in the table is replaced by X and the constraint set of Y is merged with the constraint set of X. This procedure (called $Compact(X, Y)$) is recursively carried out for the node-addresses in the $next(X)$ and $next(Y)$. Since the $next(Y)$ is empty, the compaction stops after substituting Y by X and deleting the row for Y.

NA	new	del	ptrVar	next	Constraints
X			p, q	X	

Dereferencing(→): If the statement is operating on a data value dereferenced from a pointer as in *p*→*data* != *d*, *p* must refer to a valid node i.e the node-address cannot be $NULL$. Hence a constraint $X \neq 0$ is generated using the node-address of *p*. If a node that does not yet have a node-address is dereferenced, a new node address is created. For a statement *if(p*→*data* != *d)*, the *initial* table changes to:

NA	new	del	ptrVar	next	Constraints
X			p	Y	$X \neq 0$
Y			q		

If a statement is operating on a pointer dereferenced from another pointer as in *q*→ *next* = *NULL*, it is treated as a combination of dereferencing and whatever operation is being done on the pointer; assignment in the given example. Hence the *initial* table is updated to:

NA	new	del	ptrVar	next	Constraints
X			p	Y	
Y			q	Z	$Y \neq 0$
Z					$Z = 0$

Allocation/Deallocation: For an allocation of the type *r* = *malloc(node)*, a new node-address, say N is created and a new row is added in the table. The constraint $N \neq 0$ is added to the $Constraints(N)$. We also mark the $new(N)$. For a deallocation of the type *free(p)*, the *del* column of the row corresponding to node-address of *p* is marked.

The Node Address Table discussed so far needs a refinement. The column named $next$ needs to be split into two subcolumns called $init$ and $curr$. The reason for this is as follows. At any given time while scanning the statements along P, the Node address table reflects the current state of the input data structure. The $next$ field of a node-address may change as the statements along the path are scanned. Hence $current$ node-address in the $next$ field of a node-address may be different from the $initial$ node-address (i.e, the one assigned to the $next$ field for the first time). At the time of constructing the input data structure, the $next$ fields of the nodes in the input data structure must be set using the $initial$ node-address that was assigned to the $next$ field and not the $current$ node-address in this field (which may be different from the initial node-address). We remember these initial node-addresses in the $next$ fields by splitting the $next$ column into $init$ and $curr$ subcolumns.

NA	new	del	ptrVar	next		Constraints
				init	*curr*	
X			p	Y		
Y			q	Z		$Y \neq 0$
Z						$Z = 0$

The Node Address Table obtained after scanning all the statements along P in Figure 3 is shown below. The steps of deriving this table are given in the Appendix A.

NA	new	del	ptrVar	next		Constr- -aints
				init	*curr*	
A			root	C		$A \neq 0$
B						
C			curr, prev	E	F	$C \neq 0$
D						
E						$E = 0$
F	*			G		$F \neq 0$
G						$G = 0$

2.2 Solving the Constraints

The constraints derived in the $Constraints$ column of the Node address table, after scanning all the statements along the path, describe the shape of a linked data structure. For each node-address N in the Node Address Table, which does not have any constraints defined, we add the constraint $N = 0$ to $Constraints(N)$. We solve these constraints by whole number assignments to the various node-addresses that satisfy the constraints. We make the following two assumptions about the node-addresses while solving the constraints. (i) There are only two kinds of node-addresses. Either a node-address is 0 (corresponding to $NULL$ pointer) or a node-address is non-zero, which corresponds to a valid address in memory. This is unlike a real system where we may have invalid non-zero addresses. (ii) Two node-addresses X and Y are assumed to be different, unless there is an explicit constraint $X = Y$. Based on these assumptions, for the above set of constraints: $A \neq 0, B = 0, C \neq 0, D = 0, E = 0, F \neq 0, G = 0$, we assign the following values to the node-addresses: $A = 1, B = 0, C = 2, D = 0, E = 0, F = 3, G = 0$.

2.3 Generating the input data structure

Using the whole number values assigned to the node-addresses in the previous step, the statements to allocate the shape of the linked data structure are output. For each distinct non-zero whole number value, the algorithm allocates a node and uses a temporary pointer to reference it (Note that no statements will be output for nodes that have been marked as new such as node-address F in the above example). For node-addresses that are zero, $NULL$ pointer is assigned. For our example the following statements would be output for the solution computed in the previous step:

```
p1 = malloc(node);   //for A
p2 = NULL;           //for B
p3 = malloc(node); // for C
p4 = NULL;         //for D
p5 = NULL;         //for E
p6 = NULL;         //for G
```

Then we set up the $next$ fields of the various nodes based on the entries in the $init$ column for each node-address. Finally, we assign the temporary variables corresponding to the node-addresses of argument pointers to the respective argument pointers. In our example this results in the following statements to be appended to the list of statements above to generate the output function F'.

```
p1→next = p3;
p3→next = p5;
*root = p1;
```

Executing the function F' derived above (shown in Figure 3(b)) generates the data structure shown in Figure 3(c). It has a suitable shape for traversal of the test path P in Figure 3(a). Once a suitable shape of the input data structure is generated, the required values in the data fields can be generated using the techniques in [5, 6].

3 The Shape Generation Algorithm

In this section, we describe our algorithm for generating the shape of the input data structure for execution of a given path in a function. Without loss of generality, we assume that the input is in a C-like source language. The shape generation algorithm given in Figure 4 takes the set of statements S along the test path P in the given function F as input. Its output is the source code of a function F' which upon execution constructs the desired data structure. We first briefly describe the following two subroutines, $FindNA(ptr)$ and $Compact(A, B)$, used in our shape generation algorithm.

- **findNA(ptr)** subroutine: Given a simple or a complex pointer ptr, the $findNA(ptr)$ routine looks up the Node Address Table and returns node-address of ptr in the table. If ptr is a simple pointer, it returns the node-address of row corresponding to ptr. If ptr is a complex pointer such as $q \rightarrow next$, it finds the node-address of q by a recursive call $findNA(q)$ and then returns the node-address of $next(findNA(q)$. If no node-address has been assigned to ptr, $findNA$ creates a new node-address, adds a corresponding row in the table and returns this node-address for ptr. It returns 0 if ptr is $NULL$.

- **Compact(A,B)** subroutine: It combines the rows for node-addresses A and B into a single row in the table. If both A and B are undefined, it does not do anything. If only one of A or B is undefined, say B is undefined, the result of $Compact(A, B)$ is the node-address A. If both A and B are defined and are distinct then the nodes referred by them are recursively compacted. It merges the node-addresses in $ptrVar(B)$ with the node-addresses in $ptrVar(A)$, since both sets of pointer variables now reference the same node-address. All references to B are replaced by A in the table, and the $Constraints(B)$ are added to $Constraints(A)$. Next it recursively compacts the node-addresses in $next1$, $next2$,... fields of A and B by calling $Compact(curr(next1(A)), curr(next1(B)))$, $Compact(curr(next2(A)), curr(next2(B)))$ etc. This is also repeated for the $init$ entry of all the pointer fields. Finally, the row for B is removed from the table.

```
        Input: A set $S$ of statements along a path $P$ in the subroutine $F$.
        Output: A function $F'$ in C language that constructs the required input data structures.
        procedure SHAPEGEN($S$)
step 1: Initialize Node Address Table with entries corresponding to the argument pointers.
step 2: for each statement $s \in S$, do
          switch ($s$)
            case ($s$ has no pointer references): ignore ($s$);
            case ($s$ operates on data dereferenced from a pointer $ptr \rightarrow data$):
              Let $N = findNA(ptr)$; Add constraint $N \neq 0$ to Constraints($N$);
            case ($s$ is pointer assignment $p = q$): Let $N = findNA(q)$;
              if ($p$ is $simple$ pointer) then Let $M = findNA(p)$; Remove $p$ from ptrVar(M); Add $p$ to ptrVar(N);
              else ($p$ is $complex$ pointer $ptr \rightarrow next$)   Let $M = findNA(ptr)$; Set the $curr(next(M)) = N$;  endif
            case ($s$ contains an Inequality comparison $p$ != $q$): Let $M = findNA(p)$;
              Let $N = findNA(q)$; Add $M \neq N$ to $Constraints(M)$ and $Constraints(N)$;
            case ($s$ contains Equality comparison $p == q$):
              $Compact(findNA(p), findNA(q))$;
            case ($s$ contains a pointer allocation $p = malloc(node)$):
              Create a new node-address $N$ and a row for $N$ in the Node Address Table;
              Mark $new(N)$; Add $N \neq 0$ to $Constraints(N)$;
              if ($p$ is a simple pointer) then Add $p$ to ptrVar(N);
              else ($p$ is of the form $ptr \rightarrow next$)   Set $curr(next(findNA(ptr))) = N$;   endif;
            case ($s$ contains a pointer deallocation $free(p)$): Mark $del(findNA(p))$;
          endswitch
        endfor
step 3: Add $N = 0$ to $Constraints(N)$ for each node-address $N$ with no constraints. Collect all the constraints.
step 4: Solve the constraints by assigning whole numbers to node-addresses that satisfy the constraints.
step 5: Use the solution obtained in step 4 to output the Shape-generation function $F'$.
        endprocedure
```

Figure 4. Shape Generation Algorithm

We now give a detailed description of the steps of the algorithm.

Step 1: Initialization. The Node Address Table is initialized by creating node-addresses (and the corresponding rows in the table) for the various argument pointers of the function F. The $ptrVar$ column for each node-address is set to the argument pointer referencing the node. Specifically, for each argument pointer ap_i, a node-address N_i is created and added to Node Address Table. For each N_i, $ptrVar(N_i)$ is set to ap_i. The table now contains the node-address entries for each of the argument pointers. We also keep a list $AP = \{(N_1, ap_1), (N_2, ap_2), ...\}$ to remember which argument pointer was assigned to which node-address. This list is used while generating the statements for the output subroutine F' in the last step.

Step 2: Scan S and build Node Address Table. In this step, the statements in S are scanned. The relevant information is extracted from each statement and entered into the table. Depending on the type of statement s, the following cases are handled.

Case(i) If s does not contain any pointers in it, then the algorithm proceeds to the next statement.

Case(ii) If s contains an operation on non-pointer elements, then for each data element dereferenced through a pointer e.g., *ptr*→*data*, the node-address of *ptr* is computed by calling $findNA(ptr)$ routine. Let $N = findNA(ptr)$. The constraint $N \neq 0$ is added to $Constraints(N)$. This is done because *ptr* can be dereferenced only if it is not $NULL$.

Case(iii) If s is a pointer assignment of the form $p = q$, first the node address corresponding to q is computed. Let $N = findNA(q)$. Next,

1. If p is a simple pointer, the node-address referenced by it must be changed to N. Let $M = findNA(p)$. Remove p from $ptrVar(M)$ and add it to $ptrVar(N)$.

2. If p is a complex pointer as in $r \rightarrow next$, the pointer field $next$ in the node-address for r must be set to N. Let $M = findNA(r)$. Set $curr(next((M)) = N$. When a pointer field is set as a result of an assignment, the $curr$ field alone is set. This is because an assignment changes the initial state of the data structure.

Case (iv) If s contains an inequality comparison of the form p != q, then let $M = findNA(p)$ and $N = findNA(q)$. Add the constraint $M \neq N$ to $Constraints(N)$ and $Constraints(M)$.

Case(v) If s contains an equality comparison of the form p == q, let $N = findNA(p)$ and $M = findNA(q)$. Call $Compact(N, M)$ to merge the rows for N and M in the table into a single row.

Case(vi) If s contains a pointer allocation of the form $p = malloc(node)$, create a new node-address N, allocate a row in the table for N, place a mark in $new(N)$ and add $N \neq 0$ to $Constraints(N)$. If p is a simple pointer then add p to $ptrVar(N)$. If p is a complex pointer as in $r\rightarrow\ next$, compute $M = findNA(r)$. Then the node-address in $curr(next(M))$ is updated to refer to N. When a pointer field in $next$ column is set as a result of a node allocation, the $curr$ field alone is set. This is because this statement is changing the initial state of the data structure.

Case(vii) If s contains a pointer deallocation of the form $free(p)$, the node addres $M = findNA(p)$ is computed and $del(M)$ is marked.

Step 3: Collect Constraints. The table now contains the information about the shape of the input data structure. However, there may be node-addresses in the table which do not have any constraints assigned to them. For each node-address N in the Node Address Table, which does not have any constraints defined, we add the constraint $N = 0$ to $Constraints(N)$. We do this to prevent the final data structure that is generated from having uninitialized pointer fields. We handle the assignment of $NULL$ pointer to each $undefined\ next$ field of non-NULL node-addresses in Step 5 of the algorithm. We now collect all the constraints from the $Constraints$ column of the Node Address Table into a constraint list CL and pass them to the constraint solver described in the next step.

Step 4: Solve Constraints. Our technique for solving the constraints on pointer addresses is based on the following observation. The constraints generated by the shape generation algorithm are simple ones of the form $A = B$ and $A \neq B$ (Either A or B may be 0). Number 0 has a special meaning as it represents the node-address for $NULL$ pointer. Our technique assigns the whole number values to the node-addresses in the constraint list CL so as to satisfy the constraints in CL. Note that a constraint such as $A = 0$ already gives the assignment for A. We first handle such constraints and substitute 0 for every occurrence of A in constraint list CL. Next we handle the constraints such as $B = D$ and use these equations to assign natural numbers to these node addresses so as to satisfy these constraints. For example, if there are three constraints $B = D$, $D = E$ and $F = H$, then assigning $B = 1$, $D = 1$, $E = 1$, $F = 2$ and $H = 2$ will satisfy these constraints. Next we use these assignments to simplify inequalities. For example, an inequality constraint $G \neq H$ will get simplified to $G \neq 2$ using the above solution. Therefore, assigning $G = 3$ will satisfy this constraint. For inequalities such as $I \neq J$ such that neither I nor J has yet been assigned any natural number, we assign an unassigned natural number to I (i.e., $I = 4$) and a different unassigned natural number to J (i.e., $J = 5$) so as to satisfy this inequality constraint. Thus we obtain a whole number assignment to all the node-addresses used in CL. If any inequality constraint of the form $B \neq B$ is obtained during any step of this process, the constraints on the pointer addresses are inconsistent. An error message "infeasible path" is displayed. Otherwise, the solver returns the set of node-address assignments AL.

Step 5: Output the Shape-generation function F'. Based on the assignments AL computed in the previous step, the algorithm now generates the statements in the shape generation function F' as follows. A temporary pointer p_i is created for each node address N_i and p_i is initialized based on the assignment $N_i = v \in AL$ depending on the value of v. However if a node-address is marked as "new" no allocation is performed for it.

1. If $v = 0$ then add the statement "node *p_i = NULL;". Remove $N_i = v$ from AL.

2. If $v \neq 0$ then add the statement "node *p_i = malloc(node);". Remove $N_i = v$ from AL. Scan through AL and if any other node-address N_j has also been assigned the value v, then output the statement node *p_j = p_i and remove $N_j = v$ from AL.

Next the algorithm sets up the pointer fields (i.e.. the $next1$, $next2$,... fields) of the above allocated nodes. This sets up the shape of the required data structure. For each node N_i, scan through the $init$ entries of its $next$ pointer fields in the Node Address Table. If the entry in the $init$ column of the $next$ field of N_i contains N_j then output the statement " p_i→next = p_j". On the other hand, if the entry in the $init$ column of the $next$ field of N_i is $undefined$, then output the statement "$p_i \rightarrow next$ = $NULL$". Finally the algorithm sets up the argument pointers to refer to the appropriate nodes of the data structure using the argument pointer list that was computed in the first step of the algorithm. For each entry (N_i, ap_i) in the argument list, output the statement "ap_i = p_i".

4 Experimental Evaluation

Implementation We implemented our shape-generation algorithm as a C program. It reads input from a text file containing statements along a test path in a given function. The source language in the input files was a subset of C. The language supported a limited set of data types, namely *int*, pointers to *int*, user-defined *struct* types and pointers to these *struct* types. For pointers only the equality comparison, inequality comparison, assignment, allocation and deallocation operations were allowed. Our program first parses each statement in the file and stores it internally as an Abstract Syntax Tree (AST). Next the Node Address Table is set up. The AST's are examined and the table entries are filled. Finally the constraints are collected and solved. Based on the values assigned to node-addresses, the statements in the shape generation function F' are output to a file.

Experiments We conducted two types of experiments with our shape generation algorithm. First we verified that the algorithm was working correctly. We used paths from a Binary Search Tree (BST) insertion function and a Linked List (LL) Search function. For each function we randomly selected different feasible paths of varying lengths. The program successfully generated the required input shape for all these cases. Next we introduced a simple error in each function. We deleted the statement which updated the pointer used for traversing the linked structure from the loops of the functions. Deleting the statement causes the path to become infeasible because the loop termination condition requires the pointer to become *NULL*. When these paths were passed as input to the program, it detected the infeasibility while trying to solve the constraints and reported it.

Next, we conducted experiments to analyze the factors influencing the running time of the algorithm. For these experiments we collected various statistics: (i) number of statements in the path, (ii) number of constraints passed to the solver, (iii) number of node-addresses created in the table, (iv) number of predicates (assertions) in the program, (v) time spent updating the Node Address table and generating and solving the constraints (*Table-Handling time*) and (vi) time spent on parsing the input and outputting the final shape generation statements (*I/O time*). The program was again run with randomly chosen input paths from the BST insertion routine and the linked list routine.

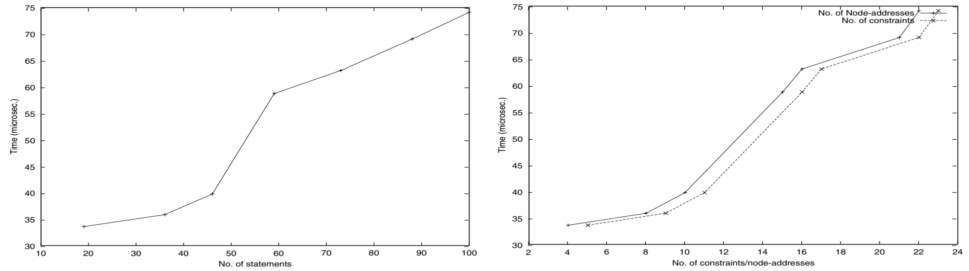
We conducted the experiments on an Intel Pentium 4, 1.7 GHz machine running RedHat Linux 7.2. The execution time taken by each subtask (e.g., I/O) was measured using the *gettimeofday()* system call. During the experiments we noticed that the times measured for a specific task varied over a range of about 5 μsec. To get accurate results, all experiments were run 50 times and the average time over these 50 runs are the ones that have been reported.

Path	# of Stmt.	# of Constr.	# of NA	# of Pred	TH (μsec)	I/O (μsec)
Bst1	17	3	3	8	33.17	2235.42
Bst2	19	4	5	8	33.58	2105.17
LL1	23	6	7	16	32.13	1525.37
LL2	36	8	9	26	35.84	1601.96
Bst3	46	10	11	23	39.74	2995.25
LL3	59	15	16	43	58.74	1729.41
Bst4	73	16	17	38	63.09	2005.84
LL4	88	21	22	65	69.05	2483.49
Bst5	100	22	23	53	74.11	2666.54

Figure 5. Timing results for different paths

Results and Discussion The timing results for the various paths are shown in Figure 5. In Figure 5, the *Table-Handling* (TH) time and I/O time are reported separately since the I/O time was the dominant factor in the total time and was also influenced by the system load. However the TH time was the true time taken for generating and solving the constraints. As can be seen from the table in Figure 5, the TH time steadily increases with the increase in the number of statements along the path.

The Table-Handling time vs. the number of statements is plotted in in 6(a). The Table-Handling time vs. the number of constraints generated and node-addresses generated are plotted in Figures 6(a) and 6(b) respectively. We observe that the time spent on collecting the constraints increases almost linearly with the number of statements, constraints and node-addresses for paths in both the functions. This is true because the statements on the paths in both the functions essentially do similar operation i.e., the pointer traversal operation. In addition, there are very few statements (2-3) along these paths that do not refer to the input data structure. For each new node traversal in these paths, the algorithm creates a new node-address and generates a constraint for the node that is being dereferenced. Hence the Table Handling time is directly proportional to the number of statements, the number of node-addresses cre-



(a) TH time vs. No. of statements

(b) TH time vs. No. of constraints/NA

Figure 6. Factors influencing the time performance of the shape generation algorithm.

ated and the number of constraints generated in these experiments. It is expected that in general, the increase in Table-Handling time with the increase in number of statements will be slower than that observed in plots in Figure 6. This is because in general, there may be many statements along the path that do not refer to the input data structure and hence do not contribute to generation of node-addresses or constraints in the table. Therefore, in general we expect better time performance of the algorithm than that shown in plots in Figure 6.

5 Related Work

Prior work [7] on generating test data for functions with pointer inputs uses a backtracking based approach to simultaneously generate the shape and the data values in the input data structure. This approach is inefficient because in case the method backtracks due to an incorrect decision made earlier about the shape of the data structure, the data values generated after the incorrect decision may become useless. In addition, there can be extensive backtracking if the statements along the paths use pointer aliasing. Since there is no concept of inconsistent constraints in this approach, a lot of time could be spent in backtracking for a path for which no feasible input data structure exists.

In this paper, we have presented a new two phase approach to generate the test data for functions with pointer inputs. We first generate the least restrictive shape satisfying the pointer constraints imposed by the statements the test path. In other words, it contains the maximum number of nodes that can be referred by the statements along the path, and it still satisfies the pointer constraints along the path. The importance of this shape can be seen at the time of generating the data values in the nodes of the generated input data structure. If the pointer constraints allowed *both* a two node data structure and a coalesced single node data structure, the single node data structure cannot be assigned two different data values if the constraints on data values require so. However, a data value in a coalesced single node structure can be copied to multiple nodes in the least restrictive data structure. Therefore, if the required data values can be generated for any other data structure satisfying the pointer constraints, then it is guaranteed that the suitable data values can be generated for for the least restrictive shape.

Our algorithm can easily detect paths for which no feasible shape exists by presence of inconsistent pointer constraints. Therefore, given a test path, it either generates a suitable shape of the input data structure or determines that no feasible shape exists for the path.

It also handles pointer aliasing efficiently. After scanning all the statements along the test path, the Node Address table always contains a single node-address for the pointers that are aliases.

6 Conclusions

In this paper we have developed a two phase approach to generate test data for path testing of functions with pointer inputs. We first generate the least restrictive shape of the input data structure that satisfies the pointer constraints along the path. The data values in this data structure can then be generated using any of the existing test data generation techniques. We have implemented our shape generation technique and our experiments show that our technique is more efficient than existing backtracking based techniques. We further plan to extend our technique by allowing the user to specify additional constraints such as there should be no cycles in the generated input data structure.

Acknowledgments

The authors thank the anonymous referees for carefully reading the manuscript and providing helpful comments.

References

[1] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 215-222, September 1976.

[2] R.A. DeMillo and A.J. Offutt, "Constraint-based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pages 900-910, September 1991.

[3] M.J. Gallagher and V.L. Narsimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, pages 473-484, August 1997.

[4] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," *International Symposium on Software Testing and Analysis*, 1998.

[5] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated Test Data Generation using An Iterative Relaxation Method" *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering (FSE-6)*, pages 231-244, Orlando, Florida, November 1998.

[6] N. Gupta, A. P. Mathur, and M. L. Soffa, "UNA Based Iterative Test Data Generation and its Evaluation," *14th IEEE International Conference on Automated Software Engineering(ASE'99)*, pages 224-232, Cocoa Beach, Florida, October 1999.

[7] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pages 870-879, August 1990.

[8] B. Korel, "A Dynamic Approach of Test Data Generation," In *Conference on Software Maintenance*, pages 311-317, San Diego, CA, November 1990.

Appendix A

Here we show how the Node Address Table is updated for the statements along the test path P' in function `func1` in Figure 3(a). Note that the branch predicates along the path are written as $assertions$ that must be true for the traversal of the path.

(1) Initial Node Address Table:

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root			

(2) Stmt scanned: prev = root

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root, prev			
B						

(3) Stmt scanned: assert(prev→next != NULL)

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root, prev	C		$A \neq 0$
B						
C						$C \neq 0$

(4) Stmt scanned: curr = prev→next

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root, prev	C		$A \neq 0$
B						
C			curr			$C \neq 0$
D						

(5) Stmt scanned: assert(curr→data != d)
No change in Node Address Table.

(6) Stmt scanned: prev = curr

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root	C		$A \neq 0$
B						
C			curr, prev			$C \neq 0$
D						

(7) Stmt scanned: assert(prev→next == NULL)

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root	C		$A \neq 0$
B						
C			curr, prev	E		$C \neq 0$
D						
E						$E = 0$

(8) Stmt scanned: prev→next = malloc(node)

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root	C		$A \neq 0$
B						
C			curr, prev	E	F	$C \neq 0$
D						
E						$E = 0$
F	*					$F \neq 0$

(9) Stmt scanned: prev→next→data = d
No change in Node Address Table.

(10) Final Node Address Table:
Stmt scanned: prev→next→next = NULL

NA	new	del	ptrVar	next		Constr-
				init	*curr*	-aints
A			root	C		$A \neq 0$
B						
C			curr, prev	E	F	$C \neq 0$
D						
E						$E = 0$
F	*			G		$F \neq 0$
G						$G = 0$