Test Shape Generation of Dynamically Linked Structures

Sittisak Sai-ngern Chidchanok Lursinsap Peraphon Sophatsathit
Advanced Virtual and Intelligent Computing (AVIC) Research Center
Department of Mathematics, Faculty of Science
Chulalongkorn University
Email: Sittisak.Sa@Student.chula.ac.th, lchidcha@chula.ac.th, and peraphon.s@chula.ac.th

ABSTRACT

This paper proposes an approach to generate test data specifically for dynamic pointer structures. A pointer is considered and handled as a location in a memory which is modeled as a linear array. Test data can be directly generated from this array. The method handles a variety of dynamic structures and the shape generation can be achieved in linear time in terms of the number of statements related to pointer operations.
Keywords: Software Testing, Unit Testing, Test Coverage of Code, Test Data Generation.

1. INTRODUCTION

Testing is one of the most important tools for ensuring the correctness of the software. One of the most difficult testing tasks is the generation of test data. The goal of the test data generator is to find the input values that will successfully execute each statement along the given path. Many approaches have been investigated for generating numerical test data but very few ([3], [5]) explore more complex structure such as dynamic structure and pointers. Pointers and dynamic data structures are the power of C/C++ programming language. Various applications extensively use them including but not limited to system software and circuit simulations[2]. The process of generating test data for dynamic structures may be separated into two steps, namely, generating linked structures and generating numerical test data. This research will concentrate on generating the linked structure.

This paper presents the following research finding.
1. An automatic technique for generating a practical structure of the input pointer structure which can be further integrated with existing numerical test data generation methods.
2. A technique to handle the pointer aliasing based on the address manipulation.
3. An infeasible path detection caused by invalid pointer operations or constraints.
The rest of the paper is organized as follows. Section 2 briefly summarizes the related works. Section 3 states the overview of the approach. Section 4 shows the algorithm. Section 5 provides the discussion and Section 6 concludes the paper.

2. BACKGROUND AND PREVIOUS WORK

It is assumed that the reader is familiar with the terms used in software test data generation. This paper concerns dynamically linked structures that are accessed through pointer variable. The scope of pointer will be confined to dynamic structure type only. The problem of generating the dynamic linked structure has two folds. First, how many nodes(discrete storages) are needed and how they are linked? We will refer to these issues as a shape generation problem. Second, what is the value for the non-pointer field within a structure? This problem falls into the same problem as generating numerical test data by treating each non-pointer field as a discrete variable.

Korel [3] proposed a goal-oriented approach with dynamic data flow analysis and backtracking to generate data for the dynamic data structure. Viswanathan and Gupta [5] only proposed an algorithm on shape generation based on recursive operation on constraint simplification. Although these two approaches work well, some improvements on more efficient shape generation time and advanced data types handling must be developed.

3. SHAPE GENERATION

3.1 Overview of The Approach

Each statement is examined for a pointer operation. If it is found, the operation is evaluated to manage the memory area where each pointer element is housed. For the actual running environment, the memory space prior to execution of the function contains some data as inputs to the function. When the function is executed, the initial data are modified to produce the output linked structure. However, for testing situation, the initial data are unknown. The approach assumes they exist by initializing each input variable (argument variable) to a given address. Subsequent creation of the address is based on execution of the statements.

3.2 Memory Representation

Our approach models a heap storage as a linear array of consecutive cells. Each cell will be addressed or indexed starting from 1. The address 0 is the special

```
struct Tree { int key; Tree *lt; Tree *rt;}
1. void ex01 (Tree *p, int v){
2. Tree *x,*y;
3.    x = p → lt;
4.    y = x → rt;
5.    x → lt = y → lt;
6.    if (x == y)
7.          y = malloc(); }
8.    if (v ≥ p → key)
9.          p → lt = y;
10.   else
11.         p → rt = y; }
```

Fig.1: An example function

address reserved exclusively for null value. A memory cell is organized into sections and the layout is from left to right according to the declaration of pointer fields of the structure. Any non-pointer field is ignored. A memory cell is represented by the following notation:

$< address, (1^{st} field, 2^{nd} field, ..., n^{th} field) >$

As an example, the cell structure for binary tree may be $< 1, (2, 3) >$ where “1” represents the address in memory, “2”, “3” are addresses assigned to fields. The memory space (MS) is the collection of these cells. We also denote $< variable, address >$ to represent an association between a pointer variable and the memory cell that the variable points to. The environment(VA) is the collection of variables and address pairs. Each memory cell contains properties to provide information about the cell including **stable** for address classification (input/create), **null** for null status, **active** for allocated or deallocated status, **pointto** to collect all nodes pointing to this node, **unequal** to contain all nodes that restricts not to be aliased with this cell, **update** to hold the value that reflects when the cell has been last executed, and **id** to store the structure type that this cell belongs to.

3.3 Pointer Operations

The operations on pointer include dereferencing, assignment, creation, deallocation, alias constraint, and equal/unequal constraints. The structure and function in Figure 3.3 will be used as an example. The selected path is < 1,2,3,4,5,6,7,10,11 >. All coding will be in C-like language.

3.3.1 Dereferencing

There are two types of addresses - the address that is derived from the initial input address (derived address) and the address that is explicitly allocated (explicit address). We summarize the concepts of dereferencing as follows:

- If all nodes in a traversed chain exist, dereferencing of each node will be based on the current value of the node.
- If the next traversed node does not exist and the current node is a derived address, further traversal will cause the creation of the node.
- If the current node is an explicit address, all pointer fields of the node will be null unless they are assigned to some locations.

The approach generates the nodes as needed. The memory affected will be in MS and VA. How the node is created will be explained in the creation section. From Figure 3.3, line 1 will create the derived address 1 in MS and create a link in VA as follows:

MS = {<1,(2,3)>}   VA = {<p,1>}

At line 2, “p→lt” refers to address 2 and it is created as follows:

MS = {<1,(2,3)>,<2,(4,5)>}

3.3.2 Assignment

The assignment may update cell properties, MS, and/or VA as follows.

if (the assigned node is the root node i.e. p, q)
   replace the current address of the node in VA with new address value (i.e. <p, new value >)
else (i.e. p→lt)
   replace the assigned node value (i.e. lt) of the before node (i.e. p) in MS with new value (i.e. < address of p, (new lt, rt) >)

The property **pointto** of assigned address is also updated to add the node that point to it. The +, -, and @ signs are used to add new element, remove the specified element, and replace the element with the new one. From Figure 3.3, line 2 will first dereferencing as in previous section, then the address is assigned to x in VA (VA: + <x,2>). Line 4 and 5 are evaluated in the same manner and the results are shown below.

4: MS: + <5,(6,7)>   VA: +<y,5>
5: MS: + <6,(8,9)>, @<2,(6,5)>

3.3.3 Creation

New address is generated and added to MS. The first created address is the address 1. The **stable** property of the new address is set to true for explicit address and false for derived address. Let x be the last allocated address and y be the address to be created. For the Tree structure, the allocation will be as follows.

MS: +<y,(++x,++x)> for derived address
MS: +<++x,(0,0)> for explicit address

From Figure 3.3, line 1, 3, 4, and 5 generate addresses using the first format. Line 7 will generate the address based on the second format (MS: +<10,(0,0)>).

3.3.4 Deallocation

Deallocation operation will release the specified address. The property **active** of the deallocated cell in MS will be set to FALSE.

3.3.5 Alias constraint

The equality constraint of two nodes (p == q) will cause two alias nodes to share the same address chain.

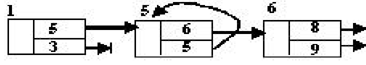


Fig.2: Result of memory operations for function ex01

If the address of the node is explicit, the alias condition(equality condition) is directly evaluated. The implicit alias involves a pointer node to which belongs the derived address. Since two chain of addresses (i.e. p, p→rt; q, q→lt, q→rt) exist before the equality constraint, the constraint will force them to merge into a single address chain. The process is defined as follows:

Repeat until no node to be merged
  if (only one node exists)
    use the existing node
  else
    move node with less property **update** to the higher one
end repeat

The process will be done in MS. From Figure ex01, line 6 will merge node x and y in MS. Address 2 will merge with 5, 6 with 6, and 7 with 5 as follows.

MS: 2→5,6→6,7→5 ⇒ − <2,(6,5)>,@< 5,(6,5)>,@<1,(5,3)>
VA: @<x,5>

3.3.6 Other constraints

All constraints except alias will modify cell properties. For example, the constraint (p == null) will set the **null** property of cell p to true.

The final results for the example in Figure 3.3 are as follows.

MS = $\{< 1, (5, 10) >, < 5, (6, 5) >, < 6, (8, 9) >, < 10, (0, 0) >\}$

VA = $\{< p, 1 >, < x, 2 >, < y, 10 >\}$

For the example, the solution addresses will be based on derived addresses and the generated test shape is given in Figure 3.3.6.

4. ALGORITHM

The algorithm pseudocode is given in Figure 1. The approach consists of 3 tasks.

- **Task 1:** Inspect each statement along the path for a pointer operation and if it exists, then dereference all pointer variables.
- **Task 2:** Classify and evaluate each operation. There are two targets to update- memory area and/or cell properties. All constraints except ALIAS update properties of the related cells. Others may update both.
- **Task 3:** Output the test shape from the input variables with original assigned addresses.

5. DISCUSSION

The goal of test data generation is to generate the shape. The essential information for each structure is the number of fields within the structure. The heterogeneous structure is different from the homogenegeous strucrture in that the number of fields for each storage cell is varied. All other processes are the same. Figure 2 shows the result for our approach on heterogeneous structure.

Our algorithm is implemented in C++. The experimental programs include Linked-List and Binary Search Tree. Each path is selected randomly. The experiment was performed on a Pentium IV 1 GHz running Microsoft Windows 2000 and Dev-C++. The experiments were run 30 times and the average time is reported. Figure 3 shows the results of the generated shape for the selected paths. We select linked list and tree to create the test structures because they are standard programming construct and easy to find reference materials. The data presented in Figure 3 (a) show the relation between the number of statements(St), total generated address (Tot), generated test addresses(Shp), constraint encounter (Con), and processing time(Time). The relationship between the number of statements and processing time, the number of constraints and processing time are shown in Figure 3 (b) and (c), respectively. Each constraint takes different amount of constant time to process. Since the path is selected randomly, different constraints are encountered and hence the graph in the (b)and (c) section are not smooth.

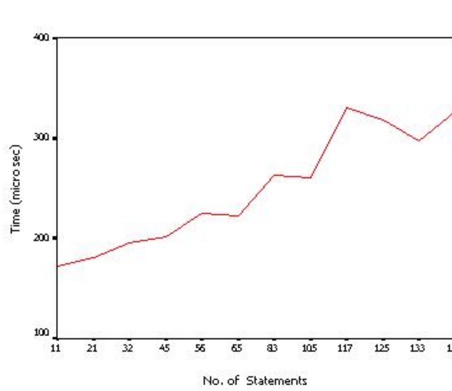
ALGORITHM: GenTestShape
input: A sequence of N statements (S) for a chosen path and data structure definition
output: Linked structures in a format of chain addresses.
i = 0
while i ≤ N
  if ($s_i \in S$ has an operation on a pointer variable)
    Dereference pointer variables
    switch (OPERATOR)
      case Operation involves assignment
        Update memory according to assignment operation
      case Operation involves allocation such as malloc()
        Update memory according to creation and assignment operation
      case Operation involves deallocation such as free()
        Update memory according to deallocation operation
      case Operation involves equality of two pointers
        Update memory according to alias constraint
      case Operation involves other constraints
        Update properties of cell addresses according to constraints
    end switch
  end if
  i = i + 1
end while
Output solution addresses
end procedure

Fig.3: Algorithm Pseudocode

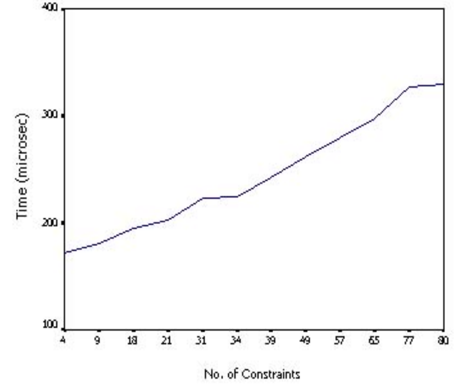
Besides the feasible path results, the program is tested on the infeasible paths by modifying the input statements to have invalid constraints which include forcing equality on nodes previously imposed inequality and vice versa, invalid referencing to a local variable, and forcing null on nodes which are previous as-

St.	Tot.	Shp.	Con.	Time (μs)
11	3	3	4	167
21	6	5	9	181
32	10	10	18	195
45	12	11	21	202
56	18	18	34	225
65	17	16	31	223
83	27	27	52	263
105	26	25	49	261
117	30	29	57	280
125	41	41	80	318
133	34	33	65	297
157	40	39	77	327

(a)



(b)



(c)

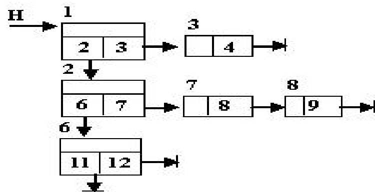
Fig.5: Results for different paths (a), Relationship between statements vs processing time(b), Constraints vs processing time (c).

```
struct infonode { int data; infonode *next; }
struct headnode { headnode *nexth; infonode
*nexti; }
s1: void ex02 (headnode *H) {
s2: headnode p;
s3: infonode q;
s4: p = H;
s5: while (p != NULL) {
s6:       q = p→nexti
s7:       while (q != NULL) {
s8:             display(q→data)
s9:             q = q→next; }
s10:       p = p→nexth; }
s11: }
Path = < s1, s2, s3, s4, s5, s6, s7, s8, s9, s7,
s10, s5, s6, s7, s8, s9, s7, s8, s9, s7, s10, s5, s6,
s7, s10, s5, s11 >
```

(a)

```
MS = {<1,(2,3)>, <3,(4)>, <4,(5)>, <2,(6,7)>,
<8,(9)>, <9,(10)>, <6,(11,12)>, <12,(1)>, <11,(14,15)> }
VA = { <H,1>,<p,11>,<q,12> }
```

(b)



(c)

Fig.4: Example function ex02 (a), Results from memory operations (b), and Generated Test Structure (c).

signed not-null constraint. The results, which are verified manually, show that the program correctly detects all infeasible paths.

6. CONCLUSION

In the area of test data generation, especially for dynamically linked structures test generation, it is difficult to find a benchmark and standard test set to compare the algorithm. A variety of programming languages, programming styles, test concepts, and target applications contribute to different demands on the outcome of the test. Many approaches propose good theoretical concepts, but painful implementation. Our approach is practical. Compare to the existing techniques, our approach is better in time complexity and variety of structures. Both existing techniques have non-linear execution time while ours is linear. The approach presented by [3] may be modified to handle the heterogeneous structure but it will be still inefficient due to backtracking process. The method presented by [5] does suggest how to handle the complex structure. However, the method will be more complex in dealing with system of related tables. Our approach will operate on both homogeneous and heterogeneous with the same process without modification.

References

[1] N. Gupta, A. P. Mathur, and M. L. Soffa. "Generating test data for branch coverage," In 15th IEEE International Conference on Automated Software Engineering (ASE'2000), Sep. 2000.

[2] J. Hummel, L. J. Hendren, and A. Nicolau, "A General Data Dependence Test for Dynamic, Pointer-Based Data Structures," in Proc. ACM-SIGPLAN'94 Conference on Programming Language Design and Implementation, pages218-229, Orlando, Florida, Jun. 20-24,1994.

[3] B. Korel, "Automated Software Test Data Generation," IEEE Transactions on Software Engineering,Vol. 16, No.8, pp. 870-879, Aug. 1990.

[4] R.E. Prather and J.P. Myers, Jr., "The path prefix software testing strategy," IEEE Transaction on Software Engineering. SE-13(7):761-765, July 1987.

[5] S. Viswanathan and N. Gupta, "Generating Test Data for Functions with Pointer Inputs," 17th IEEE International Conference on Automated Software Engineering (ASE'02), pp. 149-160, Edinburgh, UK, Sep. 2002.