Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology 47 (2005) 199–214
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

An address mapping approach for test data generation of dynamic linked structures

Sittisak Sai-ngern*, Chidchanok Lursinsap, Peraphon Sophatsathit

Department of Mathematics, Faculty of Science, Advanced Virtual and Intelligent Computing (AVIC) Research Center, Chulalongkorn University, Bangkok 10330, Thailand

Received 5 April 2004
Available online 2 October 2004

Abstract

Software testing is an important technique to assure the correctness of the software. One of the essential prerequisite tasks of software testing is test data generation. This paper proposes an approach to generate test data specifically for dynamic pointer structures. In our context, a pointer is considered and handled as a location in memory, represented by a dynamic linear array that expands and shrinks during execution. As such, pointer test data can be directly generated from this linear array. The proposed technique can also support any dynamic structures, as well as homogeneous and heterogeneous recursive structures.

Keywords: Test data generation; Linked structure; Software

1. Introduction

1.1. Software testing

A number of approaches have been proposed to ensure the quality of software including technical reviews, walkthroughs, inspection, testing, proof of correctness, simulation, prototyping, and requirement tracings. Among these methods, testing is one of the most important and practical techniques used in industry to assess a software product and reduce the risk of failure [4]. It is an expensive, tedious, and labor-intensive task which accounts for up to 50% of the total cost of software development [1,15]. The testing process involves many subtasks. One difficult aspect is test data generation [2], which requires considerable amount of resources and effort to carry out. As a consequence, the cost incurred inevitably rises. In order to reduce the development costs and improve the quality of software, automatic test data generation is considered desirable. A number of automatic test data generation methods have been proposed. Korel [9] classified these methods into three types: random [5,18], pathwise [9,12], and data specification [5,17]. Random testing and data specification are not appropriate for a program with dynamic structure type since the search space is large. On the other hand, pathwise testing manifests the input data generation to satisfy some kind of adequacy criteria (i.e. branch coverage, statement coverage, etc.). It is primary for unit testing since the method is more effective at finding faults at the unit level, and correcting faults at this level is less expensive when faults are detected during unit testing [11]. This paper, therefore, proposes an approach, which focuses on path wise test data generation.

1.2. Test data generation

Edvardsson [3] models a typical test data generator system, which consists of three parts, namely, program analyzer, path selector, and test data generator. Program analyzer takes source code as an input and produces control-flow or data-flow information such as a data-dependence graph, control flow graph, etc. Based on these data, the path selector selects paths to be executed by the test data generator. The goal of the test data generator is to find the input values that will execute every statement along

* Corresponding author.
E-mail address: Sittisak.sa@student.chula.ac.th (S. Sai-ngern).

0950-5849/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2004.08.004

the path. We will concentrate primarily on this particular aspect of the pathwise test data generator.

Most existing approaches for generating test data deal with basic numerical data types such as integer and real numbers. Although these data types are commonly useful in various applications, they cannot be feasibly used in many applications where symbolic computations are needed and the data size is not fixed. These applications employ advanced data types such as pointer, structure, and file types. Applications written in C/C++ extensively utilize the predominant pointers and dynamic structures, e.g. system software and circuit simulations [8]. Pointer variables range from simple structure (single storage) to recursively defined structure (many discrete storage that form shapes such as linked list, tree, heap, etc.). For simple pointer structure such as 'int *a', generating test data can exploit the algorithms of prior work [6,12]. However, some relevant features such as alias and pointer arithmetic must be considered as well.

Generating test data for a recursively defined structure is more difficult than for a non-recursively defined structure. The problem is two-folds. First, how many nodes are needed as inputs to force a traversal along a given path and how are they linked? These problems are categorized as *dynamic linked structure generation problems*. Second, what should the value for the non-pointer data field within a structure be? The latter problem falls into the same realm as generating test data for basic numerical data types that treat each non-pointer data field as a discrete variable. We will investigate the dynamic structure generation problem.

The overall process of generating input data for a recursively defined structure is divided into two steps. The first step, referred to as address mapping, focuses on solving dynamic structure generation through actual execution of the statements under test. A path is selected and all the statements along that path are laid out linearly. A memory address is assigned to each pointer reference for use in actual execution. However, not all addresses are assigned and participated in the test data generation for the designated path. Only pointers to input addresses (subsequently referred to as class I, see Section 4) are involved. Each statement is evaluated by adjusting the address and content of every pointer during the execution. If any constraint cannot be satisfied during statement evaluation, the algorithm will report an infeasible path. The second step focuses on data generation which rests primarily on existing data generation methods such as [10,12].

In this paper, the following results are expected:

- A technique for automatic input pointer structure generation which can be further integrated with existing numerical test data generation methods. This technique can be subsequently applied to different language paradigms such as object-oriented or assembly language.
- A technique for mapping a pointer reference to linear array structure.
- An algorithm to resolve address for equality and inequality constraints.
- A direct and effective approach to manage pointer alias and heterogeneous pointer structure.
- A detection method for infeasible path caused by invalid pointer constraints and operations.

These results will contribute to the derivation of our proposed address mapping approach for test data generation of dynamic linked structures.

The organization of this paper is as follows: Section 2 presents an overview of the proposed approach by examples. Section 3 clarifies terms used and explains basic concepts of pointer operations involved in this approach. Section 4 elucidates details on dynamic structure generation algorithm. The experiments and discussion are provided in Section 5. Section 6 discusses some related works. Concluding remarks and future work are summarized in Sections 7 and 8, respectively.

2. Overview

In this section, we will describe the proposed approach in general terms. The proposed approach takes the statements along the flow path and relates data structure definitions as inputs to generate the test structure. Each statement is examined for any pointer operation. If one is found, the operation is evaluated to reorganize the memory space where each pointer element is allocated. In an actual running environment, the memory space prior to execution of the function usually contains input parameters. When the function is executed, the initial data are modified to produce an output linked structure. However, in a test environment, the initial data are unknown. To compensate for such an unknown situation, the proposed approach assumes their existence by initializing every input variable (argument) to a designated address. Subsequent address creation is based on demand as required by the statement. Any reference to this input will expand the address chain. Statement execution can further expand, modify, or shrink the chain depending on the operation.

To clarify our concept, function ex01 is used as an example. The function takes two arguments. The first argument is of dynamic structure type that contains two pointer fields. The other argument is of integer type. The pseudo address generation procedure generates the linked structure, which subsequently is reduced to numerical data fields according to the dynamic domain reduction method. The output demonstrates the mapping from pseudo address to real address.

The example declares a node as a dynamic structure containing two pointer fields denoted by $<address, (lt, rt)>$, where 'address' represents the location in memory(ψ), 'lt' and 'rt' are address values assigned to pointer

fields of the node structure which serve as the environment linking pointers to memory locations(γ).

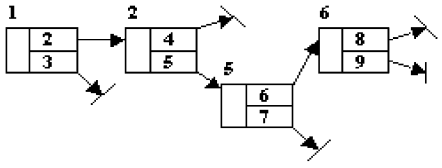
Memory location count begins sequentially from 1. The allocation process will always select the address following the last one allocated. The variable χ has the same structure as ψ, but contains only the address and initial values which are derived from the input variables. Any address that is generated by a function such as malloc( ) will not affect the contents of χ. Fig. 1 demonstrates how the notation is represented. The path selected for function ex01 is $<$1,2,3,4,7,11,12,13,14,15,17,18$>$.

The following describes how the proposed configuration executes function ex01.

Line 1: The input variable p is assigned address 1 in ψ, χ, and.γ The '+' sign represents an add or union operation. The initial value of pointer fields lt and rt are 2 and 3, respectively. These values are reserved addresses in ψ and χ but no actual creation takes place.
Line 2: There is nothing done for local variable declaration.
Line 3: Reference of p will force the address 1 to be not null since p is assigned to point at a data object.
Line 4: $p \rightarrow$lt is dereferenced to $p=1$ and $p \rightarrow \text{lt}=2$. The address 2 is not yet created. Thus, it is created in ψ and χ with its field values initialized to 4 and 5. Address 2 is subsequently assigned to x, which adds to γ.

```
struct node { int key; node *lt; node *rt;}
1. void ex01 (node *p, int v){
2. node *x,*y;
3. if (v < p → key)
4.       x = p → lt;
5. else
6.       x = p → rt;
7. if (v < x → key) {
8.       y = x → lt;
9.       x → lt = y → rt;
10.      y → rt = x;}
11. else {
12.      y = x → rt;
13.      x → rt = y → lt;
14.      y = malloc(); }
15. if (v ≥ p → key)
16.      p → lt = y;
17. else
18.      p → rt = y;
19. }
Path = <1,2,3,4,7,11,12,13,14,15,17,18>
```

(a)



(c)

Line	Memory (ψ)	Environment (γ)	Solution(χ)
1	$+\{<1,(2,3)>\}$	$+\{<p,1>\}$	$+\{<1,(2,3)>\}$
4	$+\{<2,(4,5)>\}$	$+\{<x,2>\}$	$+\{<2,(4,5)>\}$
12	$+<5,(6,7)>,$	$+\{<y,5>\}$	$+\{<5,(6,7)>\}$
13	$+\{<6,(8,9)>\}$		$+\{<6,(8,9)>\}$
	$-\{<2,(4,5)>\}$		
	$+\{<2,(4,6)>\}$		
14	$+\{<10,(0,0)>\}$	$-\{<y,5>\}$	
		$+\{<y,10>\}$	
18	$-\{<1,(2,3)>\}$		
	$+\{<1,(2,10)>\}$		

(b)

Fig. 1. Example function ex01 (a), Memory operations (b), and generated test structure (c).

Line 7: Similar to line 3, address 2 cannot be null from any constraint.
Line 12: y and $x \rightarrow$rt are dereferenced. Address 5 with initial field values 6 and 7 are added to ψ and χ for $x \rightarrow$rt. The assignment will set y to address 5, which is shown in γ.
Line 13: $y \rightarrow$lt is dereferenced. Address 6 with values 8 and 9 is created and added to ψ and χ. x is dereferenced to address 2 and the rt field is reset to 5. This is done by first removing (–) the original address(2) of x and adding it back again with the value of the rt field changing from 5 to 6.
Line 14: Adding to ψ, the malloc( ) allocates address 10 with initial field values (0,0). γ updates y to this new address.
Line 15: There is nothing done, since address 10 is already restricted to non-null.
Line 18: y is dereferenced to address 10. p is dereferenced to address 1 and the rt field of this address is changed the value from 3 to 10 in the same manner as line 14.

After all statements have been processed, addresses 1,2,5,6, and 10 are created. There are two sources of these addresses, namely, input (1,2,5,6) and malloc( ) function (10). The target test structure is obtained from χ, which contains input addresses 1,2,5, and 6. In order to obtain the complete test data set for the function, the numerical test data must be generated. Many methods for the test data generation have been proposed. In this paper, the dynamic domain reduction approach is selected due to its scalability and practical use. Details of the algorithm can be obtained from [12]. In this example, the algorithm will generate numerical data for variables v, $p \rightarrow$key, and $x \rightarrow$key, assuming that the initial domain for each variable lies between -10 and 10. These initial values may be assigned to minimum and maximum possible values of the host machine, or restricted to a reasonable input specification range, or based on the test engineer's knowledge. In our example, the chosen values are based on the test engineer's knowledge. One valid test case obtained from the algorithm are $v = -5$, $p \rightarrow \text{key} = 1$, $x \rightarrow \text{key} = -5$. To combine the non-pointer field values to the pointer structure, a new field is added to the output structure which is created from the generated address representing the 'key' data field. Thus, the modified address structure will be $<$*address*, (*key*, *lt*, *rt*)$>$. The results are depicted in Fig. 2. Note that the numerical data field of the last two nodes at address 5 and 6 are not involved in any constraint conditions or computations; hence they can be any value. In this case, they are set to 0.

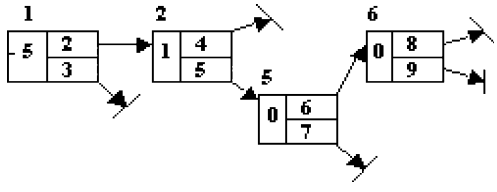


Fig. 2. The integrated output for the algorithm.

The generated pseudo address may be mapped to a designated target language such as C or C++. The mapping process consists of three simple steps, namely, node allocation, numerical data assignment, and link assignment. Depending on the language, node allocation will create an exact number of nodes in accordance with the number of pseudo addresses generated. Each allocated node consists of a numerical data field and a pointer field. The former will be assigned a value obtained from the generated numerical test data whereas the latter will be linked to one of these newly created nodes according to the match between its field value and pseudo address in χ of the designated node. This mapping process is thus illustrated by the following example.

Generated test data:
Address 1: (1,(2,3)), data: -5
Address 2: (2,(4,5)), data: 1
Address 3: (5,(6,7)), data: 0
Address 4: (6,(,8,9), data: 0

Step 1: Node allocation:
node1 = malloc(node);
node2 = malloc(node);
node3 = malloc(node);
node4 = malloc(node);

Step 2: Numerical data assignment:
node1 → key = −5;
node2 → key = 1;
node3 → key = 0;
node4 → key = 0;

Step 3: Link assignment:
node1 → lt = node2;
node2 → rt = node3;
node3 → lt = node4;

All pointer fields of the allocated memory are defaulted to null.

3. Fundamentals of dynamic linked structures

It is assumed that the reader is familiar with the terms used in software test data generation. These include

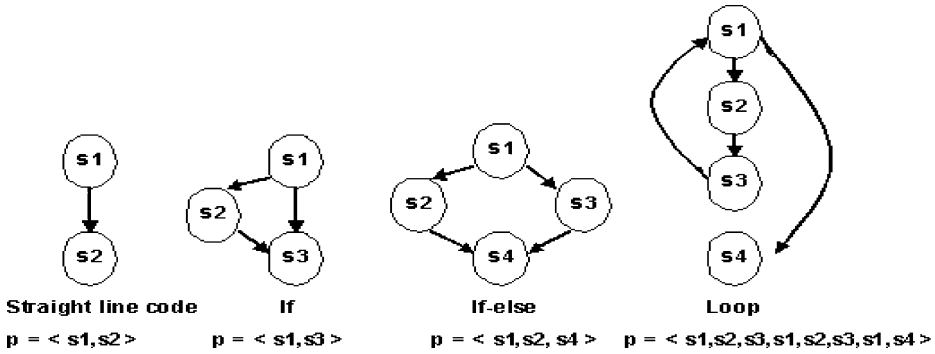


Fig. 3. Examples of program structures along with one possible path selection for each construct.

program (function), input variable of a function, basic block, predicate, constraint, path, flow graph, control path, feasible path, and infeasible path. Details of these terms are given in [3,12].

This paper concerns pointer variables, which contain an address in memory of another variable as their value. The type of the pointer designates the type of the dynamic record or a recursively defined pointer structure container. A recursively defined pointer structure container is divided into two parts. The first part is a record variable, which has at least one field being a pointer of the same type. The second part is an unnamed record of a given type. Both parts can be separately created. The former is allocated first. The latter can subsequently be built and connected to form a linked list or a tree. This pointer structure container is created dynamically by standard memory allocation functions such as malloc( ). Such a configuration suits the proposed recursivelydefined structure. This paper will thus refer to a 'pointer' as the recursively defined pointer structure. The pointer structure is further classified into two types—homogeneous and heterogeneous pointer structures. The homogeneous pointer structure is a pointer structure that contains a pointer field of the same type such as linked lists or binary trees, while the heterogeneous pointer structure is a structure that contains a pointer field of different types, such as union of embedded generic pointers to structures, trees, and file objects.

Based on the above structural framework, path selection for test execution will commence to map every dynamically linked object so generated in the form of 'pointer'. A number of procedural constructs and operations are involved in the execution process which will be elucidated below.

3.1. Execution sequence

The execution sequence is based on the identification of a set of statements that make up a path. The selection of statements aims at satisfying some testing criteria. Many approaches [13] exist to fulfill the task. The path may also be manually provided or randomly generated. This paper opts for the randomly generated path. Regardless of path selection methodologies, a path will contain a sequence of finite statements, which are not necessarily unique. For example, statements within a loop may appear more than once for the path. Fig. 3 shows examples of possible paths for different program constructs.

Finally, a path is a sequence of finite statements $p = <s_1, s_2, \ldots, s_n>$, where s_n is the last statement of path p. The flow of execution is from s_1 through s_n in forward direction. The statement that contains a conditional operation like s_3 in the code segment below may be evaluated to s_3 or s_3', where s_3' denotes the false path of predicate evaluation.

s_3: if $(x == y)$
s_4: $x = a$;
s_5: $y = c$;

If the statement s_4 must be executed, s_3 will evaluate to true, i.e. '$x == y$'. On the contrary, if the statement s4 is omitted, s_3' will be '$x != y$'.

If any $s_i \in p$ is not executed successfully, the path is broken or infeasible. Our approach is interested in subpath $p_s = <s_{p1}, s_{p2}, \ldots, s_{pm}>$ where s_{pi} contains pointer elements, $p1 \geq 1$ and $pm \leq n$. A pointer element may refer to a pointer variable directly (i.e. p) or a traversed field (i.e. $p \rightarrow f1$, $p \rightarrow f1 \rightarrow f2$, etc.). We classify pointer elements into two classes, namely,

- Class I: The value of a pointer element is from a set of input addresses.
- Class II: The value of a pointer element is from a newly allocated memory (e.g. $p =$ malloc( )) or from a transitive assignment of newly allocated memory (e.g. $p = q$, where q is previously executed as $q =$ malloc( )).

Table 1 shows examples of class I and II. The underlying structure declaration for the examples is TREE (struct TREE {int data; TREE *left; TREE *right;}).

Before the execution of a statement, each variable must be dereferenced before use. If the current value of the variable is from an input address, it will belong to class I. The statement on line 4 is in class I before and after execution since the value of the input variable p is unchanged. Variable x is a local variable whose value

Table 1
Examples of class

Line	Statements	Pointers	CLASS before/after	Comments before/after
1	void classI_II (TREE *p, TREE *q)			
2	{			
3	TREE *x, *y;			
4	x=p→right;	x	–/I	L/A
		p→right	I/I	D/D
5	y=q→left;	y	–/I	L/A
		q→left	I/I	D/D
6	p=malloc( );	p	I/II	D/E
7	p→left=x;	p→left	II/I	E/A
		x	I/I	D/D
8	q→right=p→right;	q→right	I/II	D/A
		p→right	II/II	E/E
9	}			

A, assignment; D, dereferenced from input; E, explicitly allocated; L, local variable.

before line 4 is undefined. After execution, x belongs to class I. The statement on line 5 follows the same pattern as line 4. The malloc( ) unction on line 6 explicitly assigns a memory block to the variable p and sets all the pointer fields in p to null, hence belong to class II. The rest of statements are interpreted in the same manner and succinctly annotated in the Comments column.

3.2. Heap representation

The fundamental programming concepts rest on level of abstraction, having rich utilities for convenience. However, data types at high level programming are more complicated to assign the values than those at low level. For this reason, our approach will manage the pointer-based structure type at low level. As mentioned earlier, a pointer in this paper refers to the pointer to a linked structure, which serves as a mechanism for addressing and managing dynamic storage. This dynamically manipulated area of memory is called heap. The actual operations on heap are system-dependent and not of concern here. The focus is on the number of heap cells needed to accommodate the dynamically generated linked structures prior to execution of a given path. Our approach models heap storage as a linear array of consecutive cells. For simplicity, each cell will be addressed or indexed starting from 1. A memory cell is divided into sections where pointer field layout is arranged from left to right according to the declaration of pointer fields of the structure. The leftmost section stores the first declared field, the second leftmost stores the second declared field, and so on. Any non-pointer field is ignored. A pointer field within the structure may be referenced by the number representing the position within the cell as shown below.

struct ex {int data, val; ex *n1, *n2;}

struct ex { int data, val; ex *n1, *n2;}

Cell addresses: 1				4		
...	x	n1	n2	x	n1	n2
Position:	0	1	2	0	1	2

The position 0 is reserved, 1 refers to $n1$, and 2 refers to $n2$, and so on.

To manage this memory address space, we define the memory and the environment as follows:

- The memory is a set of address-value list that specifies cell addresses and their values, i.e.

  $$\psi = \{<a^1, (a^1_1, \ldots, a^1_{n1})>, \ldots, <a^k, (a^k_1, \ldots, a^k_{n1})>\}$$

  where a^i denotes a cell address i, $1 \le i \le k$; a^i_j is the jth pointer field value of a^i, and ni is the number of fields of the structure, $1 \le j \le ni$.
- The environment γ of an execution path is a set of address value pairs that associate a variable with point-to address

  $$\gamma = \{<v_1, a^1>, \ldots, <v_n, a^k>\}$$

  where v_i is a pointer variable, a^i denotes the cell address defined earlier, $1 \le i \le n$.

As an example, let D be a data structure declaration containing two pointer fields; p and q be pointer variables that are declared to be the D type. Suppose p points to location 1 which has addresses 2 and 3 as its pointer field values. The variable q is set to 4 with values 5 and 6 for the pointer fields. Locations 2, 3, 5, and 6 are empty. The configuration of the environment and memory can be expressed as follows:

$$\gamma = \{<p, 1>, <q, 4>\}$$

$$\psi = \{<1, (2, 3)>, <4, (5, 6)>\}$$

To access a particular item from the environment and the memory, we define $\gamma(v_i)$ to denote the address of v_i in the list. The function $\psi(a^i, j)$ will access the value of pointer field in position j at location a^i, $j \ge 1$. From the above example, the address of p and its field values are retrieved as follows:

$\gamma(p) = 1$,
$\psi(1, 1) = 2$, and
$\psi(1, 2) = 3$.

Besides the value of pointer field, each cell also contains the following properties:

- A Boolean flag *stable* to hold the value true if it belongs to class II, and false if it is class I.

- A Boolean flag *active* to indicate that i is the active cell which can be accessed for pointer operations. This flag is false for the cell that is freed.
- A three-value flag *null* to hold three-stage condition, that is, a 0 as the default for non-NULL, 1 for NULL, and 2 for unmodifiable non-NULL value.
- The set *unequal* to contain all nodes that restricts not to be aliased with this cell.
- An integer *id* to contain the id of the structure type to which this cell belongs.

3.3. Pointer representation and operations

3.3.1. Pointer representation

A dynamic data structure consists of a collection of data elements. Each element is called a node. There are two types of nodes, namely, point-to node and reachable node. The point-to node refers to the node that is directly accessed by a pointer variable while the reachable node can be accessed by traversing the point-to node. Based on this configuration, the nodes will be transformed into *traversed vector format*. The traversed vector contains the pointer variable at column 1. The remaining columns represent the rest of traversed nodes by numerical value. Each traversed node corresponds to the field of the structure in the form of its positional value as defined by the heap structure. The value starts from 1 for the first declared field, 2 for the second declared field, and so on. As an example, consider a binary-tree variable p having left and right pointer fields, these two fields are denoted by positions, 1 and 2, respectively. A C-like convention is given below.

$p \equiv [p]$
$p \rightarrow \text{left} \equiv [p,1]$
$p \rightarrow \text{right} \equiv [p,2]$
$p \rightarrow \text{right} \rightarrow \text{left} \equiv [p,2,1]$

The length of $[p]$,$[p,1]$,$[p,2]$,$[p,2,1]$ are 1,2,2, and 3, respectively. From the above example, $p \rightarrow \text{right} \rightarrow \text{left}$ can be represented by $[p,2,1]$. Also, given the traversed vector $v = [p,2,1]$, we can access each element by using the index of the vector, i.e. $v(1) = p$, $v(2) = 2$, and $v(3) = 1$.

3.3.2. Operations

To make it clear, how a pointer is operated, we impose the following restrictions on pointers

- Pointers are allowed to point to dynamically allocated records or null.
- The initial values of all fields of the allocated node from conventional memory allocation operations such as malloc( ) are null.We also make a default assumption that can be overwritten by the statements along the path.
- Every node is stored in a unique storage location.

This assumption will allow a pointer variable to have a unique address on its first reference. The operations on the pointer may be divided into six categories, namely, USE, MODIFY, CREATE, DELETE, ALIAS, CONSTRAINTS.

3.3.2.1. USE operation. The operation will traverse the pointer chain and generate (if it does not exist) proper pointer elements associated with the variables for all valid dereferencing. As mentioned before, there are two classes of pointer elements or nodes. Class I is concerned with the input value while class II involves explicitly allocated memory. However, a typical traversed chain may hold both classes. For example, the point-to node is a node in class I, the second reachable node is a node in class II, and the third node is a node in class I. The following code demonstrates the concepts.

```
1: void mixednode (Tree *y)
2: {
3:     y→left = malloc();
4:     y→left→right = p→right;
5:     if (y→right→left→right == NULL)
6:         free(y→right→left→right);
               ...
7:     if (y→left→left→right == NULL)
8:         y→left→left→right = malloc();   }
```

The function *mixednode* has one input variable y of Tree type. The statement on line 1 will generate a class I node. Connecting to this node, the statement on line 3 generates a class II node. Line 5 creates two more class I nodes for $y \rightarrow$ right and $y \rightarrow \text{right} \rightarrow$ left. The statement on line 7 will not successfully execute because the node $y \rightarrow$ left is of class II whose left field value is by default null and thus prohibiting further traversal. We summarize the concepts of dereferencing as follows:

- If all nodes in a traversed chain already exist, dereferencing of each node will be determined based on the current pointer value of the node.
- If the next traversed node does not exist and the current node is in class I, further traversal will require memory allocation of the next node in class I.
- If the current node is in class II, all pointer fields of the node will be null. No further traversal from those fields is possible unless they are assigned to some location.

The approach generates the nodes as needed. If a node of type I is created, all fields of the node will be assigned an addresses ready for further expansion. However, any field of this node that has not been referred will be set to null at the end of execution. If the node is constrained to null, i.e. if ($y \rightarrow$ right $==$ NULL), the property $i \cdot$null of the corresponding address i for this node will be set to 1 and no further dereferencing is allowed.

Let p be a pointer variable to a dynamic linked structure and v be a traversed vector of p. We denote Use($v(n)$) as the operation that returns the address value of $v(n)$, which is defined as follows:

Use($v(0)$) = $\gamma(v(0))$
Use($v(1)$) = ψ(Use($v(0)$,$v(1)$))
Use($v(2)$) = ψ(Use($v(1)$,v(2)))
…
Use($v(n)$) = ψ(Use($v(n-1)$,$v(n)$))

The definition of Use($v(n)$) is recurrent. It always begins with Use($v(0)$) for the starting address of traversal. The null property of Use($n-1$) is set to 2, otherwise the node Use(n) cannot exist. For $0<x<n$, if the node $x-1$ is in class I and the node x does not exist, the approach will generate the node x under the following conditions:

- Use($v(x-1)$)·active = True
- Use($v(x-1)$)·null → = 1
- Use($v(x-1)$)·stable = False

How the node x is created will be explained in the CREATE operation (Create(d,class)).

3.3.2.2. MODIFY operation. This operation will assign an address value to a node Let p be a pointer variable pointing to a dynamic linked structure, v be a traversed vector of p, x be an address in memory. We denote Modify(v,x) to be a function that assigns the value x to node v. The function is defined as follows:

```
Modify(v,x)
    if (x = =NULL)
        Use(v(n)).null = 1
    else if (n = =0)
        γ(Use(v(n)) = x
    else
        ψ(Use(v(n-1)),v(n)) = x
    end if
end Modify
```

If the address x is null, no memory or environment is updated. Only the null property of the node is set. When the pointer variable itself is assigned a value ($n=0$), the assignment will update the variable in the environment γ to establish a connection between the address and the variable. Otherwise, update will be done at the memory space ψ. As an example, let p, q be input variables of Tree type and have assigned addresses 1 and 4, respectively. Suppose $p \rightarrow$ left is located at address 20 having $<20, (21, 22)>$. The following statement is executed

$p \rightarrow \text{left} \rightarrow \text{right} = q;$

The contents of ψ at location 20 will be changed to $<20, (21, 4)>$.

3.3.2.3. CREATE operation. The operation will allocate one address space for a node. If the node is in class II, all fields will be set to null. If the node is in class I, all fields will be assigned reserved address values that represent specific addresses in memory when the node is referenced. Let d (structure id) be an integer number representing the structure type and k be the number of fields in the structure. We denote the allocation of d by a function that allocates the address in the memory which is defined as follows:

```
Create (d, class, reserved_address)
    k = number_of_fields(d)
    if (class = =I)
        a^i = reserved_address
        a^i_1 = avail_list( ).
        …
        a^i_k = avail_list( )
        ψ = ψ ∪ [{ <a^i,(a^i_1,…,a^i_k)> }
        χ = χ ∪ [{ <a^i,(a^i_1,…,a^i_k)> }
    Else
        a^i = avail_list( )
        ψ = ψ{ <a^i,(0,…,0)> }
    End if
    Property active of a^i is set to class and Property id is
    set to d
end Create
```

The Create( ) operation takes three arguments, namely, d, class, and reserved_address. The argument d denotes the structure id, the argument class represents the created address class, and the argument reserved address holds the reserved_address to be created. The operation returns the allocated location at the reserved address. As an example, let d be the structure id for a Tree structure, the address 19 be the last allocated/reserved address. Execution of Create( ) will create the address 20 ($<20, (21, 22)>$) and reserved addresses 21 and 22. Thus, the next available address will be 23. All properties at the address 20 will be set to default values, except the property *active* and *id*. The newly allocated memory space is then assigned to a node v with MODIFY operation, i.e. Modify(v,allocated address)).

3.3.2.4. DELETE operation. The operation will release the specified address and the property *active* of the deallocated cell will be set to FALSE. Let p be a pointer variable pointing to the dynamic linked structure at the specified address and v be the traversed vector of p. We denote Delete(v) as the function that frees the memory occupied by the node v. The function is defined as follows

```
Delete(v)
    if (Use(v,n)·null != 1)
        Use(n)·active = FALSE
end Delete
```

The operation does not actually release the memory space but merely marks it as deleted. The reason being is to maintain reachability of any node to this address for the infeasible path detection. However, a subsequent dereference passing through this address will cause an error.

3.3.2.5. ALIAS constraint. This constraint will cause two alias nodes to share the same address chain. There are two types of alias, namely, explicit alias and implicit alias. Address sharing of explicit alias from the assignment statement such as $p=q$ is carried out by means of the MODIFY operation. If the node is in class II, the alias condition (equality condition) is directly evaluated explicitly. The implicit alias, on the other hand, is from the equality condition of two pointers which involves pointer nodes in class I. The implicit sharing commences at any point in the program as the two nodes indirectly share the same address through their respective links. As an example, the code segment below demonstrates the implicit address sharing.

```
void falias (Tree *p; Tree *q){
1: Tree *x, *y;
2: x=p→left;
3: y=q→right; ←start implicit sharing of nodes x and y
which is subsequently realized when line 6 is true and no
implicit sharing if line 6 is false
4: x→right=y→left;
5: y→right=q;
6: if (x==y)←sharing of addresses is known when
"x==y" is true
   ··· }
```

Two chain of addresses (i.e. x, $x\rightarrow$right; y, $y\rightarrow$left; $y\rightarrow$right) for the implicit alias nodes exist in a memory space before the equality constraint. When "$x==y$" is evaluated to be true, these two chains of nodes must be resolved to a single chain. The implicit alias constraint is the most operation to handle since there are many factors to consider in coping with the constraint. For example, two chains of nodes may not have the same length; traversed nodes p or q may be combination of class I and class II due to earlier assignments, etc. A sequence of assignments may not be simple to merge the address as shown in the following code segments

```
s1: q=r→left;
s2: q→left→left=a2;
s3: p→left=a3;
s4: q→left→left→right=a4;
s5: if (p==q)
   ···
```

Statement s1 redefines a variable q to $r\rightarrow$left. If s5 is true, statements s2 is canceled by s3 and s4 must be reprocessed since p and q share the same node, which is $r\rightarrow$left. To eliminate this problem, our approach makes the implicit alias explicit by preprocessing all statements along the given path. The preprocessing task is to mark the statement that latest redefines a value of an alias node. During the address generation step, if a marked statement is encountered, the execution of this statement is postponed until the alias( ) is evaluated. The following modification illustrates the concepts.

```
s5′: alias(p,r→left) ←e.g. p=r→left
s1: q=r→left;
s2: q→left→left=a2;
s3: p→left=a3;
s4: q→left→left→right=a4;
s5: if (p==q).
···
```

The sequence of execution is rearranged with s5′ preceding s1. The implicit alias is eliminated since the address mapping process will assign p and q to the same address after s1. No regenerating or combining of addresses needs to be done. It does not matter to set s5′ to be '$r\rightarrow$left$=p$' or '$p=r\rightarrow$left'. The address generation process will decide the choice when the operation is encountered. The alias( ) function operates similarly to the MODIFY operation, except that it must verify the property *unequal* of both nodes before the assignment. To speed up preprocess task, we use two simple structures to identify where the alias statement should be put. The first structure, SymTab, maintains pointer nodes and statements that update their current values. The structure is defined as follows:

SymTab:

variables (var)	fields (fd)
(var_i, st_j)	{(st_{k1},…, st_{kn})}

An order pair (var_i, st_j) represents a pointer variable and a statement which modifies its value while (st_{k1},…, st_{kn}) refers to a series of statements that update the corresponding fields (1,2,…,n) of the pointer variable. The layout of fields is from left to right according to their declaration. An update of a node will override all updates of the descendant nodes. As an example, statements 2–5 from the example function falias( ) generate SymTab as follows:

variables (var)	fields (fd)
(x,2)	{(0,4)}
(p,2)	{(2,0)}
(y,3)	{(4,3)}
(q,3)	{(0,3)}

The line containing (x, 2) and (0, 4) shows that the variable x gets updated at line 2 and its right node ($x\rightarrow$right) is modified at line 4 while its left node is yet referred. The rest are interpreted in the same manner. Note that $y\rightarrow$right is updated at line 5 but the value 3 is kept because its address is defined by q which is created at line 3. If line 7 is added

with an assignment '$x \rightarrow \text{right} \rightarrow \text{left} = y$', the row of the table containing variable x will be $(x, 2)$ and $\{(0, 4),(3, 0)\}$.

The second structure (AliasTab) maintains the position in which the alias function must be inserted. It consists of a statement (st_i) which modifies an alias node together with an alias variable (var_j).

AliasTab:

Statements	Variables
st_i	var_a

From the previous SymTab and the function falias( ) at line 6, AliasTab is updated as follows.

Statements	Variables
3	x

Since y is updated after x, x must be kept in AliasTab as an alias placeholder along with the statement (3) that redefines y ($y = q \rightarrow \text{right}$). When the statement at line 3 is evaluated from the address generation process, it will be verified against AliasTab. Since the line number exists in the AliasTab, the process will execute the function alias( ) with arguments x and $q \rightarrow \text{right}$ before generating target addresses for line 3.

The function preprocess( ) must be done before executing the address generation process which is defined below.

```
preprocess( )
   for s_i = s_1 to s_n
      if referred node does not exist
         add (node,statement) to SymTab
      if Operation is "alias"
         retrieve the corresponding statement from
         SymTab
         add (statement, alias variable) to AliasTab
      else if Operation is "update" (e.g., assignment
      (=))
         update (node, statement) to Symtab
      end if
   end for
end preprocess
```

From the example function falias( ), the preprocess( ) will yield the execution sequence as shown below.

```
2: x = p → left;
3′: alias(x, q → right)
3: y = q → right;
4: x → right = y → left;
5: y → right = q;
6: if (x = = y)
   …
```

3.3.2.6. CONSTRAINTS. All constraints except alias operate on properties of the address cells. For example, the constraint '$p = =$ null' will set the null property of p to true Other constraints work in the same manner.

4. DLS generation algorithm

The goal of the AddressMapper algorithm is to generate the linked structure that can be used to evaluate the path. If there exists a shape to suit the path, the path may or may not be a feasible path. The suitable shape for the path implies that pointer variables do not have any conflicts with pointer operations and constraints. The numerical data generation algorithm is needed to generate data for non-pointer fields and input variables to assure the feasible path.

The AddressMapper algorithm is summarized in Fig. 4 and the pseudo code is given in Fig. 5. The approach consists of three main tasks, namely, classifier, preprocessor and evaluator.

Classifier: Each statement along the path is inspected for any pointer operations before passing through the classifier. The classifier identifies the pointer operations and transforms all the pointer variables involved to corresponding traversed vectors. These vectors, together with operation type, are inputed to the preprocessor before proceeding to the evaluator where the memory operations are simulated. The classifier task is to directly map C-like pointer naming convention to vector format. The reason is to make the algorithm flexible for implementation since the classifier and main module can be written in any language while the evaluator can be generalized due to the well-defined vector format.

Preprocessor: The preprocessor eliminates any implicit alias by asserting an explicit alias statement at the proper place.

Evaluator: The evaluator manages the memory address space through the memory(ψ) and the environment (γ) variables as defined in Section 3. The solution space (χ) is a partial shadow of the. Every time the address in class I is generated in ψ, the same address will also be copied to χ. Once it is there, it will not be updated in χ again.

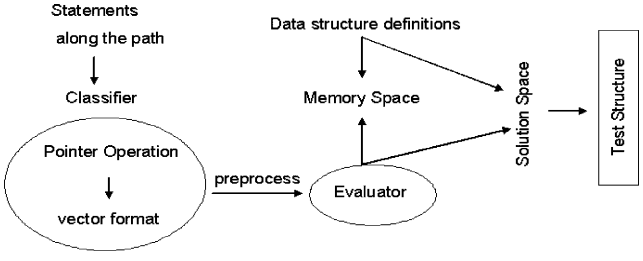


Fig. 4. Concepts overview.

```
ALGORITHM: AddressMapper
input: A sequence of N statements (S) for a chosen path and data structure
definition.
output: Linked structures in a format of chain addresses.
S = classifier()
preprocessor()
for i = 1 to N
        if (s_i = AliasTab.currentline)
           evaluator(AliasTab.var,s_i.v2,ALIAS)
           AliasTab.delete
        else
           evaluator(s_i.v1, s_i.v2, s_i.operator)
        end if
end for
χ.output()
end procedure

procedure evaluator (t_vector v1, t_vector v2, int OPERATOR)
    switch (OPERATOR)
        case USE_ONLY          // An operation on a data field.
            Dereferencing on v1 and/or v2
        case MODIFY
            Modify(v1,v2)
        case CREATE
            x = Create(structure_id)
            Modify(v1,x)
        case DELETE
            Delete(v1,n1)
        case ALIAS
            alias(v1,v2)
        case CONSTRAINT_INEQUALITY
            if (address of v1 is not equal address of v2)
               adjust property unequal of v1 and v2
        case CONSTRAINT_NULL
            if (Use(v1,n1).null != 2)  Use(v1,n1).null = 1
        case CONSTRAINT_NOTNULL
            if (Use(v1,n1).null != 1)  Use(v1,n1).null = 2
    end switch
end procedure
```

Fig. 5. Pseudocode of MemoryTrace algorithm.

On the other hand, class II address is never placed in χ since it is not part of the input address.

The evaluator takes three arguments, which include two pointer nodes in a traversed vector (t-vector) format and one operation type. Each type is evaluated as defined in Section 3. For constraint operations, no memory modification is made, only the property of the node is updated if it does not violate the previous setting. To obtain the proper result, we maintain the original assigned value corresponding to the variable in γ_0, which has the same structure as γ. The target structure is obtained by traversing the address chain starting from γ_0. All reserved addresses for pointer fields, however, are set to null.

5. Discussion

The proposed approach is evaluated against many linked structures such as singly linked list, doubly linked list, circular list, tree, and heterogeneous structures. The following series of functions demonstrate the algorithm. All operations detail how memory (γ, ψ, χ) are updated. The resulting test structures for each example are also included. However, properties updated are omitted for proper length. These updates are straightforward and can be traced without difficulty. Examples on doubly linked list, circular list, and more advanced structures are shown in the Appendix. Tree examples were previously used as examples.

In the area of test data generation, especially for dynamic structures test generation, it is difficult to find a benchmark and a standard test set to measure the feasibility and efficiency of the algorithm under investigation. A variety of programming languages, programming styles, test concepts, and target applications contribute to different demands on outcome of the test. Many approaches propose good theoretical concepts, but difficult implementation. The proposed approach, on the contrary, handles numerical data types, pointer types, and dynamic structure types well by virtue of straightforward concepts as described earlier. As a consequence, recursively generated and backtracking operations are eliminated, thus allowing smooth test execution.

Simple and straightforward as the approach may sound, there are some limitations. The approach does not handle pointer arithmetic, casting, and function pointers. Also, a few assumptions are made for implementation simplification, but has not yet been extended to incorporate them. These assumptions are arrays, union, and recursive functions.

A few technical rationales can be drawn from the application of the proposed approach.

5.1. Well-known linked structures

Generating the test structure for singly liked list, doubly linked list, circular list, and tree structures types using the proposed algorithm is straightforward as shown earlier. Nevertheless, the test structure so obtained and the programmer's desired structure may not be the same since there may exist more than one structure for both successful and unsuccessful execution. The goal is to find one test structure that fulfills the task. This structure is not guaranteed to be at a minimum because our initial assumption assigns a unique storage to each non-existing referenced node. If the referenced node is not set to point to itself or to the previous created node, it will contain a new node. This is a flexible provision to set the node to any value for future use.

Let S be the total pointer-related statements of a given path and K be the maximum number of references for each node in a statement (i.e. $K=3$ for $p\rightarrow\text{next}\rightarrow\text{next}$). Each statement involves at most two variables. The classifier time complexity is O(S) for screening pointer operations. The preprocessor also has the same time complexity. For the evaluator, the statement will create an address for the node if it does not exist. The creation takes constant time in accessing and updating the last available address. Any constraints imposed on the node will also update the property of the node in constant time. The time efficiency of our approach is O($K\times S$). In practice, K is always limited to some numbers. Most programmers do not write a program with too long references such as $p\rightarrow\text{next}\rightarrow\text{next}\rightarrow\text{next}\rightarrow\text{next}\rightarrow\text{next}$. Hence, the time complexity will be O(S) which is relatively linear to S.

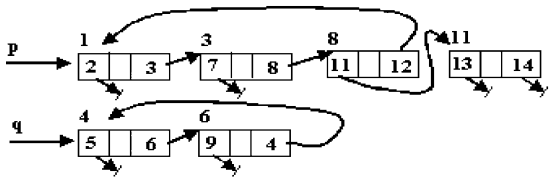
5.2. Heterogeneous structures

In a semantic sense, heterogeneous structures consist of two or more different structures mixing within the same structure declaration. However, for address generation, each pointer field is treated equally since it will occupy one address cell. So, the structure will take $1+k$ spaces, where k is the total number of pointer fields plus the structure itself. Regardless of the type of each field, the value of the pointer is always an address in memory space or null. Care must be taken on the number of fields for each structure to be assigned the correct memory addresses. Our approach handles this by storing the structure id to each housing cell. As illustrated in the examples from Section 4, the algorithm processes heterogeneous structures no differently as the homogeneous ones. Most of the existing work does not consider this type of structure, but some may extend to handle it with additional complexities.

Examples of memory update based on doubly-linked list, circular list, and heterogeneous pointer structures.

```
struct DBLList { int data;
struct DBLList *left,*right; };
s1: void test02 (DBLList *p, DBLList *q) {
s2: DBLList *pptr, *qptr, *l,*d;
s3: pptr = p→right;
s4: qptr = q→right;
s5: while (qptr != q) {
s6:       while (pptr != p)
s7:           if (pptr→data == qptr→data) {
s8:               l = pptr→left;
s9:               d = pptr;
s10:              pptr = pptr→right;
s11:              l→right = pptr;
s12:              pptr→left = l;
s13:              free(d); }
s14:          else
s15:              pptr = pptr→right;
s16:      pptr = p→right;
s17:      qptr = qptr→right; }
s18: }
Path = < s1, s2, s3, s4, s5, s6 ,s7, s14, s15, s6, s7, s8,
s9, s10, s11, s12, s13 ,s6, s16, s17, s5, s18 >
Preprocess:
< s1,...,s10′,s10,s11,...,s17′,s5,s18 >
s10′: alias(p,pptr→right);
s17′: alias(q,qptr→right);
```

(a)



(b)

Fig. A.1. Example function test02 (a) and memory operations (b).

5.3. Infeasible path

Our approach can detect an infeasible path caused by the following invalid pointer operations:

- Traverse beyond the null node
- Traverse beyond the free node.
- Assign to a free node.
- Constraint unsatisfaction.

If the property null of the current node is set to false, any reference beyond this node is invalid. The same holds for the property active. A node can be assigned to any node except the node that has the property active set to false. If the property unequal of node x contains node y, these two nodes cannot be implicit alias. If the node has its null property set to 1, the node cannot be subject to non-null constraint. On the contrary, if the property null is set to 2, the null constraint is illegal for this node. With our approach, these infeasible path conditions are simple to detect.

6. Related work

Research in dynamic structures includes pointer analysis, shape analysis, and test generation of dynamic structure.

Line	Memory (ψ)	Environment (γ)	Solution(χ)
s1	$+\{<1,(2,3)>\}$	$+\{<p,1>\}$	$+\{<1,(2,3)>\}$
	$+\{<4,(5,6)>\}$	$+\{<q,4>\}$	$+\{<4,(5,6)>\}$
s3	$+\{<3,(7,8)>\}$	$+\{<pptr,3>\}$	$+\{<3,(7,8)>\}$
s4	$+\{<6,(9,10)>\}$	$+\{<qptr,6>\}$	$+\{<6,(9,10)>\}$
s15	$+\{<8,(11,12)>\}$	$-\{<pptr,3>\}$	$+\{<8,(11,12)>\}$
		$+\{<pptr,8>\}$	
s8	$+\{<11,(13,14)>\}$	$+\{<l,11>\}$	$+\{<11,(13,14)>\}$
s9		$+\{<d,8>\}$	
s10′	$-\{<8,(11,12)>\}$		$-\{<8,(11,1)>\}$
	$+\{<8,(11,1)>\}$		$+\{<8,(11,1)>\}$
s10		$-\{<pptr,8>\}$	
		$+\{<pptr,1>\}$	
s11	$-\{<11,(13,14)>\}$		
	$+\{<11,(13,1)>\}$		
s12	$-\{<1,(2,3)>\}$		
	$+\{<1,(11,3)>\}$		
s13	8.*inactive*		
s16		$-\{<pptr,1>\}$	
		$+\{<pptr,3>\}$	
s17′	$-\{6,(9,10)>\}$		$-\{6,(9,10)>\}$
	$+\{6,(9,4)>\}$		$+\{6,(9,4)>\}$
s17		$-\{<qptr,6>\}$	
		$+\{<qptr,4>\}$	

Fig. A.2. The generated test structure for example function test02.

Pointer analysis [7,16] collects information about pointers such as the possible memory location a pointer might point to. One of the major issues addressed in this analysis is pointer aliasing. Shape analysis [14,20] analyzes the source program and produces shape graphs for every statement as appropriate. The analysis may be used to find the error that might occur from pointer misused. These methods are related to optimization in compilers. Although they may be applied to program testing, they are computational and resource expensive and do not contain sufficient information for proper testing.

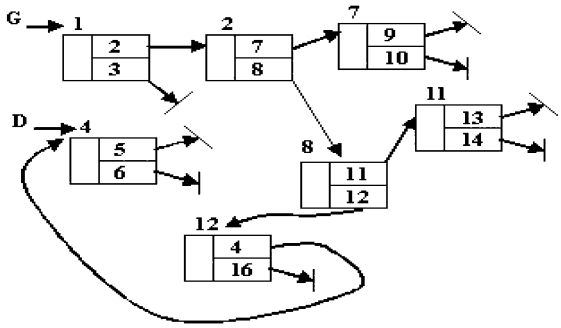
The method proposed by Kore [9] attempts to generate the dynamic data structure and data by using dynamic data flow analysis and backtracking. The method is based on goal-oriented approach by dividing the task into series of subgoals and solving each subgoal to satisfy the path.

```
struct arctype { struct nodetype *anode;
                 struct arctype *next; }

struct nodetype { int node; struct arctype *arc;
                  struct nodetype *node; }

s1: void test03 (nodetype *G, nodetype *D) {
s2: nodetype p;
s3: arctype q,bq;
s4: p = G;
s5: q = p→arc;
s6: while (q != NULL) {
s7:       if (q→anode == D)
s8:             return;
s9:       bq = q;
s10:      q = q→next; }
s11: bq = malloc();
s12: bq→anode = D;
s13: bq→next = NULL;
s14: }
Path = < s1, s2, s3, s4, s5, s6, s7, s9, s10, s6,
s7, s9, s10, s6, s7, s8 >
Preprocess: <s1,...,s7′,s7,s8>
s7′: alias(D,q→anode);
```

(a)



(b)

Fig. A.3. Example function test03 (a) and memory operations (b).

The process begins by initially setting up an arbitrarily input data structure and randomly generating the data for a structure. If the data do not satisfy the constraints, the next data values are obtained by using the proposed searching method. If all attempts result in no data values that satisfies the constraints, backtracking will take place to switch over to new shapes and start solving for different numerical data. The procedure is repeated until a solution is found or no solution to the problem is determined to exist. This approach has introduced a concept for input dynamic structure argument. However, the method has many disadvantages. First, it lacks a systematic method for building a shape. How the current node points to the next node is not well-defined. Arbitrary node connection makes the algorithm unclear. Second, solving the shape and generating data at the same time with this backtracking method is time-consuming as the shape must be rebuilt every time since the data value generated for the shape needs to be regenerated and the previously generated data value becomes unused. Third, the method is not efficient in handling pointer constraint-equality or inequality constraint. Finally, the major problem of pointer, i.e. pointer alias, is not efficiently solved by the backtracking technique.

The approach proposed by Viswanathan and Gupta [19] takes a different approach to generate data for this data type. They only focus on shape (dynamic structure) generation. The technique collects the constraints along a path and puts them into a table with varying number of columns depending on the input data structures. The algorithm simplifies the constraints by expanding or recursively combining the rows. The table is then used to solve the constraints. The approach improves the prior work on pointer alias handling, making it well-suited to simple dynamic type such as linked-list and tree structures. However, it still has to backtrack along the table in solving the alias problem. For more complex structures such as heterogeneous pointer structures, the approach does not explicitly handle them. More works

Line	Memory (ψ)	Environment (γ)	Solution(χ)
s1	$+\{<1,(2,3)>\}$	$+\{<G,1>\}$	$+\{<1,(2,3)>\}$
	$+\{<4,(5,6)>\}$	$+\{<D,4>\}$	$+\{<4,(5,6)>\}$
s4		$+\{<p,1>\}$	
s5	$+\{<2,(7,8)>\}$	$+\{<q,2>\}$	$+\{<2,(7,8)>\}$
s7	$+\{<7,(9,10)>\}$		$+\{<7,(9,10)>\}$
s9		$+\{<bq,2>\}$	
s10	$+\{<8,(11,12)>\}$	$-\{<q,2>\}$	$+\{<8,(11,12)>\}$
		$+\{<q,8>\}$	
s7	$+\{<11,(13,14)>\}$		$+\{<11,(13,14)>\}$
s9		$-\{<bq,2>\}$	
		$+\{<bq,8>\}$	
s10	$+\{<12,(15,16)>\}$	$-\{<q,8>\}$	$+\{<12,(15,16)>\}$
		$+\{<q,12>\}$	
s7′	$-\{<12,(15,16)>\}$		$-\{<12,(15,16)>\}$
	$+\{<12,(4,16)>\}$		$+\{<12,(4,16)>\}$

Fig. A.4. The generated test structure for example function test03 (c).

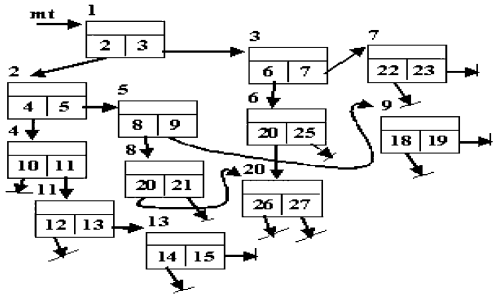
```
struct SMatrix { RowNode *rows; ColNode *cols; }
struct RowNode { DataNode *rdata; RowNode *nextrh; }
struct ColNode { DataNode *cdata; ColNode *nextch; }
struct DataNode { int data; DataNode *nextr, *nextc; }
s1: void test05 (SMatrix *mt, int val) {
s2: RowNode *trows;
s3: ColNode *tcols;
s4: DataNode *r,*c,*p;
s5: trows = mt→rows;
s6: tcols = mt→cols;
s7: while (trows→nextrh != NULL) {
s8:  r = trows→rdata;
s9:  while(r != NULL) {
s10:     if (r→data == val)
s11:       while (tcols→nextch != NULL) {
s12:         c = tcols→cdata;
s13:         if ( r == c)
s14:           if (c→nextr != NULL) {
s15:             c = c→nextr;
s16:             free(r);
s17:             return; }
s18:         else {
s19:           p = c→nextr;
s20:           while (p != NULL)
s21:             if (p == r)
s22:               if (p→nextr != NULL) {
s23:                 c→nextr = p→nextr;
s24:                 free(r);
s25:                 return; }
s26:               else
s27:                 p = p→nextr; }
s28:         tcols = tcols→nextch; }
s29:     r = r→nextc; }
s30:  trows = trows→nextrh; } }
```

Path = < s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s29, s9, s10, s29, s9, s10, s29, s9, s30, s7, s8, s9, s10, s11, s12, s13, s18, s19, s20, s21, s22, s23, s24, s25 >

Preprocess: <s1,...,s19′,s19,s20,...,s25>

s19′: alias(r,c→nextr);

(a)



(b)

Fig. A.5. Example function test05 (a) and memory operations (b).

need to be done on managing cross-table references and operations.

7. Conclusion

Many approaches for software test data generation are limited to basic data types such as integer values. Input test data generation on a primitive data type within a recursive structure is obtained from various existing methods such as [6,12]. Some approaches go the distance to investigate more complex types such as recursive structures and non-linear structures. Unfortunately, the inherent complexity renders the existing methods inefficient. Our proposed address mapping algorithm, on the other hand, treats each pointer in a typical recursive structure as a sequence of operations on addresses. The desired input structure for each pointer variable is chained addresses which permits traversal to any given path. This pseudo-address generation supports generic recursive structures that includes homogeneous and heterogeneous pointer structures. As a consequence, the alias and constraint handling problems are solved efficiently.

Line	Memory (ψ)	Environment (γ)	Solution(χ)
s1	$+\{<1,(2,3)>\}$	$+\{<mt,1>\}$	$+\{<1,(2,3)>\}$
s5	$+\{<2,(4,5)>\}$	$+\{<trows,2>\}$	$+\{<2,(4,5)>\}$
s6	$+\{<3,(6,7)>\}$	$+\{<tcols,3>\}$	$+\{<3,(6,7)>\}$
s7	$+\{<5,(8,9)>\}$		$+\{<5,(8,9)>\}$
s8	$+\{<4,(10,11)>\}$	$+\{<r,4>\}$	$+\{<4,(10,11)>\}$
s29	$+\{<11,(12,13)>\}$	$-\{<r,4>\}$	$+\{<11,(12,13)>\}$
		$+\{<r,11>\}$	
s29	$+\{<13,(14,15)>\}$	$-\{<r,11>\}$	$+\{<13,(14,15)>\}$
		$+\{<r,13>\}$	
s29	$+\{<15,(16,17)>\}$	$-\{<r,13>\}$	$+\{<15,(16,17)>\}$
		$+\{<r,15>\}$	
s9	$15.null$		$15.null$
s30		$-\{<trows,2>\}$	
		$+\{<trows,5>\}$	
s7	$+\{<9,(18,19)>\}$		$+\{<9,(18,19)>\}$
s8	$+\{<8,(20,21)>\}$	$-\{<r,15>\}$	$+\{<8,(20,21)>\}$
		$+\{<r,8>\}$	
s11	$+\{<7,(22,23)>\}$		$+\{<7,(22,23)>\}$
s12	$+\{<6,(24,25)>\}$	$+\{<c,6>\}$	$+\{<6,(24,25)>\}$
s19′	$-\{<6,(24,25)>\}$		$-\{<6,(24,25)>\}$
	$+\{<6,(8,25)>\}$		$+\{<6,(8,25)>\}$
s19		$+\{<p,8>\}$	
s22	$+\{20,(26,27)>\}$		$+\{20,(26,27)>\}$
s23	$-\{6,(8,25)>\}$		$-\{6,(8,25)>\}$
	$+\{6,(20,25)>\}$		$+\{6,(20,25)>\}$
s24	$8.inactive$		

Fig. A.6. The generated test structure for example function test05.

8. Future work

The proposed approach provides a simple, efficient, yet straightforward mapping of complex (heterogeneous) structures to a uniquely identified address space The procedure still requires considerable enhancement to accommodate other language constructs and programming paradigms, i.e. object-oriented language, declarative language, real-time programming (where allocation and deallocation overhead may not be tolerable). Various system aspects such as storage overlay, as well as dynamic address translation primitives, pose a formidable challenge to the stored-program architecture.

Conservatively speaking, we envision that the proposed approach will pave the way and inspire future research endeavors to arrive at more systematic approaches for solving other pointer-related conditions, such as recursive structures, memory leak, pointer to function, and illegal operations (that are not covered in Section 3.3.2), as well as general dynamic storage allocation problems.

Appendix

Examples of memory update based on doubly linked list, circular list, and heterogeneous pointer structures.

Figs. A.1–A.6

References

[1] B. Beizer, Software Testing Techniques, second ed., Van Nostrand Reinhold, New York, 1990.
[2] DRIVE safety—towards a European standard: the development of safe road transport informatic systems. DRIVE Project V1051, 1992. Draft 2.
[3] J. Edvardsson, A survey on automatic test data generation, Proceedings of the Second Conference on Computer Science and Engineering in Linkoping, CCSSE'99, ProNova, Norrkoping, Oct. 21–22, 1999, pp. 21–28.
[4] S. Gardiner (Ed.), Testing Safety-Related Software: A Practical Handbook, Springer, Berlin, 1999.
[5] A. Gotlieb, B. Botella, M. Rueher, Automatic test data generation using constraint solving techniques, International Symposium on Software Testing and Analysis, 1998.
[6] N. Gupta, A.P. Mathur, M.L. Soffa, Automated test data generation using an iterative relaxation method, ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering(FSE-6), Orlando, FL, Nov. 1998, pp. 231–244.
[7] M. Hind, M. Burke, P. Carini, J.-D. Choi, Interprocedural pointer alias analysis, ACM Transactions on Programming Languages and Systems 21 (4) (1999) 848–894.

[8] J. Hummel, L.J. Hendren, A. Nicolau, A general data dependence test for dynamic, pointer-based data structures, in: Proceedings of ACMSIGPLAN'94 Conference on Programming Language Design and Implementation, Orlando, FL, Jun. 20-24, 1994, pp. 218–229.
[9] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.
[10] C. Michel, M. Rueher, Y. Lebbah, Solving constraint over floating-point numbers, in: Seventh International Conference on Principles and Practice of Constraint, Springer, LNCS, Berlin, 2001.
[11] J. Offutt, An integrated automatic test data generation system, Journal of Systems Integration 1 (3) (1996) 391–409.
[12] J. Offutt, Z. Jin, J. Pan, The dynamic domain reduction approach to test data generation, Software-Practice and Experience 29 (2) (1999) 167–193.
[13] R.E. Prather, J.P. Myers Jr., The path prefix software testing strategy, IEEE Transactions on Software Engineering SE-13 (7) (1987) 761–765.
[14] M. Sagiv, T. Reps, R. Wilhelm, Solving shape-analysis problems in languages with destructive updating, ACM Transactions on Programming Languages and Systems 20 (1) (1998) 1–50.
[15] I. Sommerville, Software Engineering, fourth ed., Addison-Wesley, Reading, MA, 1992.
[16] B. Steensgarrd, Points-to analysis in linear time, in: ACM Symposium on Principles of Programming Languages, ACM, New York, 1996.
[17] P. Stocks, D. Carrington, A framework for specification-based testing, IEEE Transactions on Software Engineering 22 (1996) 777–793.
[18] M.Z. Tsoukalas, J.W. Duran, S.C. Ntafos, On some reliability estimation problems in random and partition testing, IEEE Transactions on Software Engineering 19 (7) (1993) 687–697.
[19] S. Viswanathan, N. Gupta, Generating test data for functions with pointer inputs, Seventeenth IEEE International Conference on Automated Software Engineering (ASE'02), Edinburgh, UK, Sep. 2002, pp. 149–160.
[20] R. Wilhelm, M. Sagiv, T. Reps, Shape analysis, in: International Conference on Compiler Construction, Mar. 2000.