Preprint: To appear in the *Journal of Software Testing, Verification and Reliability*

Generating Input Data Structures for Automated Program Testing

Insang Chung
Hansung University
Division of Computer Engineering, Seoul, Korea
Email: insang@hansung.ac.kr

James M. Bieman
Colorado State University
Department of Computer Science, Fort Collins CO 80523, USA
Email: bieman@cs.colostate.edu

October 8, 2007

Abstract

Automatic test data generation usually concerns identifying input values that cause a selected path to execute. If a given path involves pointers, then input values may be represented in terms of 2-dimensional dynamic data structures such as lists or trees. When testing is conducted for programs in the presence of pointers, it is very important to identify a shape of the input data structure describing how many nodes are required and how nodes are connected each other. The approach presented in this paper makes use of the points-to information for each statement in the selected path that will be used to represent the shape of an input data structure. It also converts each statement into static single assignment (SSA) form without pointer dereferences. This allows the approach to consider each statement in the selected path as a constraint involving equality or inequality to make use of current constraint solving systems without significant effort. The SSA form serves as a system of constraints to be solved to yield input values for non-pointer types. An empirical evaluation shows that shape generation can be achieved in linear time in terms of the number of pointer dereference operations.
Keywords: Program Testing, Shape Generation, Automated Test Data Generation.

1 Introduction

Software testing is an essential step for improving software quality, but it consumes large amounts of time and computing resources. The cost of software testing can be reduced significantly by automating the process of test data generation. Test data generation can be treated as a search for

input values to exercise a selected path. Several methods have been proposed to attack the problem [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. However, most of the prior work focuses on search algorithms that find solutions to traverse the selected path in the absence of pointers or heap-allocated structures.

This paper deals with test data generation for programs with pointers. Usually program code involving pointers manipulates dynamic data structures referenced by input pointer variables. A *shape* of the data structure is a configuration of data objects, i.e., how many data objects are needed and how they are linked to each other. Since dynamic data structures may have a variety of shapes including linked lists and binary trees, one needs to determine a suitable shape of the input data structure in order to traverse a given path. Of course, one also needs to uncover the values of the data fields in the input data structure as well as the values of input variables that are not pointers.

The authors' prior work [12] introduced a shape generation technique that handles pointers to stack-allocated objects, i.e., pointers to *int*. This paper extends this prior work to more complicated pointers that reference heap objects such as user-defined *struct*s and to interprocedural interactions to analyze procedures with procedure calls. In addition, a tool supports an empirical evaluation of the approach.

Given a program path, the approach provides:

- a shape of the input data structure and
- a set of constraints describing how to assign values for input variables that are not of pointer type in order to cause the traversal of the selected path.

The approach represents a shape of the input data structure in terms of points-to relations (i.e., what pointer variables are pointing to) [13] for each input pointer variable. The key action is to introduce a new points-to relation whenever necessary. For example, consider a statement of the form "x=*y" for an input pointer variable y. If the variable y is not NULL and it does not yet point to any storage location, it would be necessary to create a storage location pointed to by y in order to execute that statement without any violation.

The new method does not directly generate test data for non-pointer input variables. Instead, it generates constraints for input variables in such a way that existing constraint solving systems can be employed. To support this feature, each statement along a test path is transformed into SSA (Static Single Assignment) form [14] which does not involve pointer dereferences if they exist. One important feature of SSA form is that each variable has at most one definition (meaning that it is assigned at most once). This allows one to regard each statement along the test path as a constraint involving equality or inequality [3]. Thus, one can apply various constraint solving techniques to come up with a solution that describes values for non-pointer input variables. If constraint solving systems that can solve nonlinear constraints are employed, the solution to the constraint system can be computed. That is, getting input values of non-pointer types depends on the ability of the constraint solving algorithm used.

The main contribution of this paper is a *static* approach to automatic program testing for programs in the presence of pointers and heap-allocated structures. The approach does not require any means for controlling the execution of the target program unlike execution-based approaches [2, 7, 10, 11, 15]. In general, execution-based approaches formulate the test data generation problem as a function minimization problem by treating each branch predicate on the given path as a function that becomes minimal when the desired outcome is produced. Thus execution-based approaches must monitor a program's execution and force execution toward the desired direction.

The approach presented in this paper separates test data generation for non-pointer types from the shape analysis problem. Such separation of concerns enables the approach to take advantage of current test data generation techniques for non-pointer variables which have been relatively well studied. Since the proposed method produces a set of constraints for non-pointer types in terms of equalities or inequalities between variables, conventional constraint solving techniques can be employed, reducing development effort.

The rest of the paper is organized as follows. Section 2 explains in detail the SSA form, basic terminologies, and definitions. Section 3 defines transfer functions associated with various types of statements and expressions dealing with statically allocated memory objects and the dereference operator '*'. It also illustrates the method through an example. Section 4 extends the method to heap directed pointers which reference objects dynamically allocated in the heap. Section 5 explains the extension of the approach to deal with procedure calls. Section 6 describes the empirical evaluation and the proof-of-concept tool. Section 7 presents related work. Section 8 gives conclusions and directions for future work.

2 Preliminaries

One straightforward way to generate test data is to extract a number of constraints (equalities or inequalities) from a path under consideration and solve the constraint system. This can be done by transforming the path into SSA form.

A key property of SSA form is that each variable has a unique static definition point [14]. In order to ensure this property, variable renaming is usually done as follows:

- every assignment to a variable v generates a new SSA variable v_i where i is a unique number,
- just after the assignment to v, v_i becomes the current name (the last version or the current instance) of v, and
- every subsequent use of v is replaced by its current name v_i.

The following discussion uses the syntax and semantics of the C programming language.

Assume that the subscript number of each SSA variable starts with 0. For example, the sequence of code `x=10;x=x+3;`is converted into SSA form as follows: $x_1 = 10; x_2 = x_1 + 3$. In this example, two SSA variables x_1 and x_2 can be treated as logical variables rather than program variables. As a result, the first assignment can be treated as the equality to assert that x_1 is equal to 10 and the second assignment as the equality to assert that the value of x_2 is equal to the result of adding 3 to the value of x_1.

However, the presence of pointers complicates the conversion of the selected path into SSA form because aliases can occur (i.e., two or more names exist for the same memory location) and a variable can be defined indirectly via a pointer dereference. This makes it necessary to exploit points-to information on the selected path during conversion to SSA form [16].

At each program location, one collects the points-to information and then replace each pointer dereference with its points-to result. For example, the sequence of assignments given by

$$\mathtt{x} = \&\mathtt{a}; *\mathtt{x} = 10; \mathtt{y} = \mathtt{a}$$

can be converted to the SSA form without the pointer dereference

$$x_1 = \&a_0; a_1 = 10; y_1 = a_1$$

by using the points-to information that x references a after executing the first assignment.

The approach represents points-to relations for each program point with σ mapping variables to memory locations:

$$\sigma \in \text{State} = \text{Var} \rightarrow \text{Loc}$$

Var is the (finite) set of variables occurring in the SSA form on the program path of interest. Loc is a set of locations (addresses) partially ordered as depicted in Fig. 1.

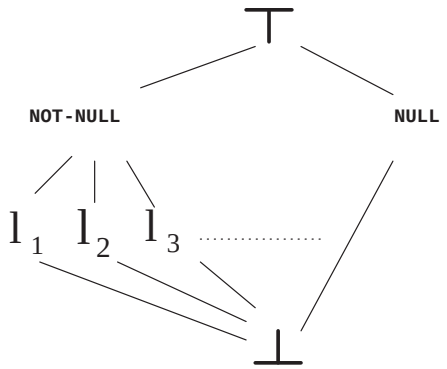

Figure 1: The structure of locations

Then, $\sigma(x)$ will now either be

- $\top$ meaning that x may possibly point to any location (x can be NULL),
- $\bot$ meaning that x is not a pointer variable or its points-to relation is undefined,
- NOT-NULL meaning that x points to a certain memory location, but its exact address is not yet known,
- l_i meaning that x points to a concrete memory location whose address is l_i or
- NULL meaning that x is not currently pointing to any location at all.

States are assumed to be partially-ordered as follows:

$$\sigma_i \sqsubseteq \sigma_j \text{if for all } x, \sigma_i(x) \sqsubseteq \sigma_j(x)$$

The approach also introduces $\bot_\sigma$ such that for all $\sigma \in$ State, $\bot_\sigma \sqsubseteq \sigma$, and uses $\bot_\sigma$ to denote that a selected path is infeasible.

It is often convenient to use a symbolic name to refer to a location instead of its address. The discussion in this paper assumes that the targets of pointers always possess a (symbolic) name. Under this assumption, the fact that variable 'x' points to a location named 'y' can be represented by $\sigma(x) = y$ without any confusion.

However, this assumption does not hold for variables that are not in the scope of a function but might be accessible through indirect reference. For example, consider function `fun` with the formal parameter `x` of type `int **`: `fun(int **x)` $\{\ldots\}$. The problem is that the function `fun` can refer to memory locations through '`*x`' or '`**x`' which are not in the scope of `fun`. In order to capture points-to information accurately, the approach names such locations using *invisible variables* [13]. Invisible variables are names for the variables that are not in the scope of a function but are accessible through indirect reference. For example, the invisible variables for `x` with type `int **` are 1_x with type `int *` and 2_x with type `int **`, respectively.

The function L_σ gives the last version of a variable with respect to σ. For example, suppose that σ is the state after executing the sequence of the assignments "`x=10; y=x+1; x=y`". Then, $L_\sigma(x)$ will give x_2. L_σ can also accept the SSA variable as input instead of the original variable. Thus, $L_\sigma(x)$, $L_\sigma(x_1)$, and $L_\sigma(x_n)$ ($n \leq 2$) will produce the same result x_2. On the other hand, let σ be the state immediately after executing the first assignment. Then, $L_\sigma(x)$ will give x_1.

The pointer variables are partitioned into disjoint collections. A collection is a set of pointer variables which should point to the same memory location. In contrast, two pointer variables belonging to distinct collections can not designate the same location. The approach assumes that (1) each pointer variable initially belongs to a distinct collection, and (2) pointer variables reference different memory locations unless there is evidence that they point to the same location. Hereafter, $[x]_\sigma$ denotes the collection of the pointer variables pointing to the same location which $L_\sigma(x)$ is pointing to.

3 Shape Generation for Program Testing

In this paper, each statement s_i in a path $< s_1, \ldots, s_n >$ is viewed as a transfer function which specifies how the statement acts on the input data structure that will traverse the (sub)path $< s_1, \ldots, s_{i-1} >$ and changes the input data structure to a new input data structure that will traverse the (sub)path $< s_1, \ldots, s_i >$. This section defines transfer functions for boolean expressions and assignments for use in shape generation for stack-allocated pointers, presents the shape generation algorithm, and illustrates the approach through an example.

3.1 Transfer functions for boolean expressions

Fig. 2 shows transfer functions for boolean expressions involving pointers. For a given boolean expression and a given state, one derives the largest solution (state) from the given state that will evaluate the target boolean expression to true.

Consider the transfer function associated with the boolean expression of the form "x <> NULL". The interesting problem occurs when a pointer variable, say $L_\sigma(x)$, should point to a certain memory location, but its exact address is unknown at the moment, i.e., $\sigma(L_\sigma(x)) = \top$. The problem is representing such a points-to relation.

The solution used here is to materialize a concrete location from $\top$. The address of the materialized location is NOT-NULL and not a specific address. The function "new-name(v, σ)" generates a name for the location as follows:

$$\text{new}-\text{name}(v,\sigma) = \begin{cases} k{+}1\_L_\sigma(x), & \text{if } k\_L_\sigma(x) \in [v]_\sigma; \\ 1\_L_\sigma(v), & \text{otherwise.} \end{cases}$$

$$
\begin{array}{lcl}
[\![\mathrm{x}==\mathrm{NULL}]\!]\sigma & = & \sigma\odot\{(\mathrm{k},\mathrm{NULL})|\mathrm{k}\in[\mathrm{x}]_\sigma\} \text{ if } \sigma(\mathrm{L}_\sigma(\mathrm{x}))\sqsupseteq \mathrm{NULL} \\
 & = & \bot_\sigma \text{ otherwise} \\
 & & \\
[\![\mathrm{x}<>\mathrm{NULL}]\!]\sigma & = & \sigma\odot\{(\mathrm{k},\mathrm{new-name}(\mathrm{x},\sigma))|\mathrm{k}\in[\mathrm{x}]_\sigma\} \text{ if } \sigma(\mathrm{L}_\sigma(\mathrm{x}))=\top \\
 & = & \sigma \text{ else if } \sigma(\mathrm{L}_\sigma(\mathrm{x}))\text{=NOT-NULL or } \sigma(\mathrm{L}_\sigma(\mathrm{x}))=l \\
 & = & \bot_\sigma \text{ otherwise} \\
 & & \\
[\![\mathrm{x}==l]\!]\sigma & = & \sigma\odot\{(\mathrm{k},l)|\mathrm{k}\in[\mathrm{x}]_\sigma\} \text{ if } (\sigma(\mathrm{L}_\sigma(\mathrm{x}))\sqcap l)\neq\bot \\
 & = & \bot_\sigma \text{ otherwise} \\
 & & \\
[\![\mathrm{x}==\mathrm{y}]\!]\sigma & = & \sigma\odot\{(\mathrm{k},\sigma(\mathrm{L}_\sigma(\mathrm{x}))\sqcap\sigma(\mathrm{L}_\sigma(\mathrm{y})))|\mathrm{k}\in[\mathrm{x}]_\sigma\} \text{ if } (\sigma(\mathrm{L}_\sigma(\mathrm{x}))\sqcap\sigma(\mathrm{L}_\sigma(\mathrm{y})))\neq\bot \\
 & = & \bot_\sigma \text{ otherwise} \\
 & & \\
[\![\mathrm{x}<>\mathrm{y}]\!]\sigma & = & \bot_\sigma \text{ if } \mathrm{L}_\sigma(\mathrm{x})\in[\mathrm{y}]_\sigma \text{ or } \mathrm{L}_\sigma(\mathrm{y})\in[\mathrm{x}]_\sigma \\
 & = & \sigma \text{ otherwise}
\end{array}
$$

Figure 2: Transfer functions for boolean expressions. Operator $\odot$ is the function overriding operator. The function $\mathrm{f}\odot\mathrm{g}$ is defined on the union of the domains f and g. On the domain of g it agrees with g, and elsewhere on its domain it agrees with f. l denotes the address of a certain location.

The function "new-name(v,σ)" makes use of invisible variables and associates a name with the location pointed to by $\mathrm{L}_\sigma(\mathrm{v})$. It first checks whether an invisible variable is included in the collection $[\mathrm{v}]_\sigma$. If there already exists an invisible variable of the form "k_$\mathrm{L}_\sigma(\mathrm{p})$", then the anonymous location will be named "$\mathrm{k}+1$_$\mathrm{L}_\sigma(\mathrm{p})$". Otherwise, a new invisible variable "1_$\mathrm{L}_\sigma(\mathrm{v})$" names the anonymous location.

Once a name is associated with the materialized location, the approach introduces a new points-to relation by making the pointer variable x point to new-name(x,σ). Note that the address of the materialized location is regarded as NOT-NULL to reflect that it can represent any (concrete) location. This is very important when another pointer variable, say y, points to a concrete location named 'm' and, at a certain point in the given program path, 'm' is shown to refer to the materialized location. That is, (*x, *y) forms an alias pair. Then, the exact address of the materialized location is reduced to the address of 'm'. If one would assign a specific address to the materialized location rather than NOT-NULL, it would not be possible to detect such an alias pair because inconsistency occurs, i.e., $\sigma(\mathrm{x})\sqcap\sigma(\mathrm{y})=\bot_\sigma$.

3.2 Transfer functions for assignments

Basic forms of the assignments considered in this section include "x=y", "x=*y", "*x=y", "x=&y", and "x=NULL". Complex statements can be treated in terms of these basic assignments. For example, the assignment "*x=*y" are broken into "temp=*y; *x=temp".

The common effects of the assignments is to generate new SSA variables since the assignments define variables directly or indirectly. If an assignment defines the variable 'x', then the transfer

```
FUNCTION GP(x,σ:State) returns σ
1:      generate a new SSA variable, N_x, for L_σ(x);
2:      set σ(N_x) to ⊤;
3:      set WorkList to {L_σ(x)};
4:      for each k in WorkList do
5:          delete k from WorkList;
6:          for each p pointing to k w.r.t σ do
7:              generate a new SSA variable, N_p, for p;
8:              set σ(N_p) to N_k
9:              add p to WorkList;
10:         endfor
11:     endfor
```

Figure 3: Function for generating new SSA variables. N_t denotes a newly created SSA variable for the variable $L_\sigma(t)$. For example, if $L_\sigma(t)$ is t_i for $i \geq 0$, then N_t denotes t_{i+1}

function associated with the assignment makes use of the function "GP(x,σ)", shown in Fig. 3, to generate a new SSA variable for the variable 'x' with respect to the state σ. It records the newly created SSA variable as the latest version of 'x' (line 1). The newly created SSA variable N_x for 'x' is initialized to $\top$ (line 2).

GP(x,σ) also generates new SSA variables for all pointer variables that point to $L_\sigma(x)$ on σ(Line 4 through line 11). For example, consider an assignment that defines the variable x when a pointer variable p is pointing to x. Even though p does not appear textually on the left-hand side of the assignment, the assignment is an indirect definition of p. Thus one needs to create a new SSA variable for the pointer variable p. This process is repeated until all pointers that can reach the storage location named x are taken into account.

Fig. 4 defines the transfer functions for the assignments; they are formulated in terms of boolean expressions. The transfer functions associated with the last two assignments play an important role in determining a shape of the input data structure required to traverse the path of interest. The primary effect of the assignments is to introduce new points-to relations whenever necessary, then make a suitable shape of the input data structure. The following discussion illustrates only the transfer function associated with the assignment of the form "$x = *y$" because others can be similarly understood.

The transfer function associated with the assignment of the form "$x = *y$" attempts to remove the pointer dereference operator by using the points-to information for y. The first clause applies to the case where $\sigma(L_\sigma(y)) = \top$. In this case, the approach materializes a location from $\top$ whose name is given by new-name(y, σ). Once a name is associated with the materialized location, the approach introduces a new points-to relation by making the pointer variable y point to new-name(y,σ). Of course, this change should be made for all pointer variables belonging to the collection containing y. The next step is simply to evaluate the transfer function associated with the equality "x==new-name(y,σ)".

The second clause of the transfer function deals with the case where y references a materialized location or a concrete location. The clause simply replaces the right-hand side of the assignment with the location y is pointing to. For example, if y points to a certain location, say v, then the right-hand side of the assignment will be replaced by v and then the transfer function associated

$$
\begin{array}{lcl}
[\![\mathrm{x}=\mathrm{NULL}]\!]\sigma & = & [\![\mathrm{x}==\mathrm{NULL}]\!]\mathrm{GP}(\mathrm{x},\sigma) \\
\\
[\![\mathrm{x}=\&\mathrm{a}]\!]\sigma & = & [\![\mathrm{x}==l_{\mathrm{a}}]\!]\mathrm{GP}(\mathrm{x},\sigma) \\
\\
[\![\mathrm{x}=\mathrm{y}]\!]\sigma & = & [\![\mathrm{x}==\mathrm{y}]\!]\mathrm{GP}(\mathrm{x},\sigma) \\
\\
[\![\mathrm{x}=*\mathrm{y}]\!]\sigma & = & [\![\mathrm{x}==\mathrm{new-name}(\mathrm{y},\sigma)]\!]\mathrm{GP}(\mathrm{x},\sigma_{\mathrm{y}}) \text{ if } \sigma(\mathrm{L}_\sigma(\mathrm{y}))=\top \\
 & = & [\![\mathrm{x}==\mathrm{m}_{\mathrm{y}}]\!]\mathrm{GP}(\mathrm{x},\sigma) \text{ else if } \sigma(\mathrm{L}_\sigma(\mathrm{y}))=\text{NOT-NULL or } \sigma(\mathrm{L}_\sigma(\mathrm{y}))=l_{\mathrm{m_y}} \\
 & = & \bot_\sigma \text{ otherwise} \\
\\
[\![*\mathrm{x}=\mathrm{y}]\!]\sigma & = & [\![\mathrm{new-name}(\mathrm{x},\sigma)==\mathrm{y}]\!]\mathrm{GP}(\mathrm{x},\sigma_{\mathrm{x}}) \text{ if } \sigma(\mathrm{L}_\sigma(\mathrm{x}))=\top \\
 & = & [\![\mathrm{m}_{\mathrm{x}}==\mathrm{y}]\!]\mathrm{GP}(\mathrm{m}_{\mathrm{x}},\sigma) \text{ else if } \sigma(\mathrm{L}_\sigma(\mathrm{x}))=\text{NOT-NULL or } \sigma(\mathrm{L}_\sigma(\mathrm{x}))=l_{\mathrm{m_x}} \\
 & = & \bot_\sigma \text{ otherwise}
\end{array}
$$

Figure 4: The transfer functions for assignments. In the transfer functions, l_{k} denotes the address of k, $\mathrm{m_p}$ denotes the location pointed to by $\mathrm{L}_\sigma(\mathrm{p})$, σ_{p} is the state computed by $\sigma_{\mathrm{p}}=\sigma\odot\{(\mathrm{k},\mathrm{new-name}(\mathrm{p},\sigma)|\mathrm{k}\in[\mathrm{p}]_\sigma\}$.

with the boolean expression "x==v" will be evaluated.

The last clause handles the case where y has the NULL value. Obviously, dereferencing y at the assignment causes a violation. Thus, the result will be $\bot_\sigma$, indicating that the path under consideration cannot be executed.

3.3 The shape generation algorithm

Fig. 5 shows the algorithm for generating a description of the shapes of the input data structure for the traversal of the given path $<\mathrm{s}_1,\ldots,\mathrm{s}_{\mathrm{n}}>$. The view taken by the algorithm is that a program path is a constraint system describing how an input data structure (or input values) should be formed in order to traverse the path. The idea is to extract a number of constraints from the given path by transforming it into SSAform without pointer dereferences. For the sub-path $<\mathrm{s}_1,\ldots,\mathrm{s}_{\mathrm{i}}>(\mathrm{i}\leq\mathrm{n})$, a solution to the constraint system will be a state σ_{i} after evaluating the sub-path. The state σ_{i} describes the shapes of the input data structure required to traverse the sub-path in terms of points-to relations for each pointer variable. Since the constraint system does not necessarily have a unique solution, the largest solution is desired.

The first step is to construct an initial state σ_0. For every variable x, Line 1 concerns the generation of its initial (SSA) version of the variable, x_0. Concerning the points-to relation, every SSA variable generated from input variables[1] is assumed to point to anything. That is, $\sigma(\mathrm{x}_0)=\top$ if x is an input pointer variable. This assumption is reasonable because memory locations pointed to by input variables should not be initially restricted. On the other hand, local variables or nonpointer variables have their points-to relation initially set to undefined. This initialization process is specified in Line 2.

[1]Formal parameters or global variables

FUNCTION get-shape(π:path) returns σ

1: for every variable x, generate its initial SSA version x_0 of the variable x_0;
2: construct σ_0 such that

$$\sigma_0(x_0) = \begin{cases} \top, & \text{if x is an input variable of pointer type;} \\ \bot, & \text{otherwise.} \end{cases}$$

3: set i to 1;
4: for each s_i in π do
5: if (s_i is of the form x<>y) then $\sigma_i = \sigma_{i-1}$;
6: else $\sigma_i = [\![s_i]\!]\sigma_{i-1}$;

7: if $\sigma_i \neq \bot_\sigma$ then
8: transform s_i into SSA form $\bar{s}_i$ without pointer dereferences
9: if ($\bar{s}_i$ is of the form x==NULL) then for all $k \in [x]_{\sigma_{i-1}}$ $[k]_{\sigma_i} = \{k\}$
10: if ($\bar{s}_i$ is of the form x==*l* and $\exists y \cdot \sigma_i(L_{\sigma_i}(y))$=*l*) then $[x]_{\sigma_i} = [y]_{\sigma_i} = [x]_{\sigma_{i-1}} \cup [y]_{\sigma_{i-1}}$;
11: else if ($\bar{s}_i$ is of the form x==y) then $[x]_{\sigma_i} = [y]_{\sigma_i} = [x]_{\sigma_{i-1}} \cup [y]_{\sigma_{i-1}}$;
12: else $[x]_{\sigma_i} = [x]_{\sigma_{i-1}}$ for every variable x;
13: else report that the path is inconsistent and exit;
14: increment i;
15: endfor

16: for each $\bar{s}_i$ of the form x<>y do
17: if ($x \in [y]_{\sigma_n}$ or $y \in [x]_{\sigma_n}$) then report that the path is inconsistent and exit;
18: endfor

Figure 5: Function for computing shape information for the selected path π=$s_1, \ldots, s_n$

Lines 3 through 15 concern evaluation of statements in the path. Line 5 defers evaluation of the boolean expression of the form "x<>y". Just after points-to information for the path is collected (Lines 16 through 18), the boolean expression of the form "x<>y" is evaluated. Lazy evaluation is used, since one can assume that pointer variables reference distinct locations unless there is evidence that they point to the same location.

Line 9 concerns the form "x==NULL". In this case, all the pointer variables in the collection $[x]_{\sigma_{i-1}}$ are supposed to point to no memory location at all. Thus, each pointer pointer variable k in $[x]_{\sigma_{i-1}}$ is separated so that $[k]_{\sigma_i}$ gives $\{k\}$. The next form that affects the points-to information is "x==*l*". The condition at line 10 checks whether an existing pointer variable points to the memory location whose address is *l*. If such a variable, say y, exists, then it is necessary to merge $[x]_{\sigma_{i-1}}$ and $[y]_{\sigma_{i-1}}$ to make the pointer variables belonging to $[x]_{\sigma_i}$ and $[y]_{\sigma_i}$ point to the same memory location with the address *l*.

Line 11 concerns the form "x==y". An interesting case arises when x and y belong to disjoint collections, but they are not in conflict (i.e., $\sigma_i(x) \sqcap \sigma_i(y) \neq \bot$). Then, the collections are merged to indicate that they should point to the same memory location from now on. The other forms do not affect the points-to information.

The time complexity of the shape algorithm is determined as follows. σ_i is computed for each s_i, that is , $|\pi|$ times where $|\pi|$ is the number of the statements plus the expressions in the given path π. The time complexity of the function GP in Fig. 3 is proportional to the square of the number of the variables in the path π, $|v|^2$, because the iteration is traversed $|v|$ times in the worst case and in each iteration, all possible points-to relations have to be considered, which equals $|v|$. Thus the worst case time complexity of the shape algorithm is $O(|\pi| \times |v|^2)$.

The space complexity is proportional to the points-to information that is computed at each program point. Since the points-to information is proportional to the number of variables, the space complexity of the shape algorithm is $O(|\pi| \times |v|)$.

3.4 An example

Suppose that one wants to identify the shape of an input data structure required to traverse the path <1,2,3,4,6,8,9,10,11> of the program in Fig. 6.

For the sake of clarity, each state is represented by a pictorial representation called a shape graph. In a shape graph, square nodes model concrete memory locations. Edges model pointer values. Suppose that $\sigma(x)$ gets y. Then, there is a directed edge from the square node named x to the square node named y. Shape graphs need not explicitly include $\bot$.

The initial state σ_0 is the following:

$$\sigma_0(x_0) = \sigma_0(y_0) = \top, \sigma_0(p_0) = \sigma_0(q_0) = \sigma_0(r_0) = \sigma_0(v_0) = \sigma_0(z_0) = \bot.$$

Initial versions of input variables of pointer type are initialized to $\top$ while local variables or variables of non-pointer type are initialized to $\bot$. Another assumption is that each pointer variable initially belongs to a distinct collection. Fig. 7(a) depicts the shape graph corresponding to σ_0.

The evaluation of assignments 1 and 2 produces the following points-to information:

- x_0 points to 1_x_0 and
- y_0 points to 1_y_0.

```
void Example(int **x, int **y, int v) {
int *p, *q, *r, z;
1:        p=*x;
2:        q=*y;
3:        if (p==q) {
4:           if (p == NULL)
5:           *q = v;
6:           else if (q == NULL)
7:                *p = v;
                  else {
8:                 r=&z;
9:                 *r=10;
10:                if (z==v)
11:                *q=v;
                  }
             }
          else {
12:          *p=v;
13:          *q=v;
          }
}
```

Figure 6: An example program

The effects of assignments 1 and 2 is to introduce new points-to relations by materializing the locations named 1_x_0 and 1_y_0 from $\top$ pointed to by x_0 and y_0, respectively. Since the two assignments define p and q, respectively, their last versions are changed to p_1 and q_1. The result is the shape graph shown in Fig. 7(b).

Consider the expression "p==q". Its effect is to put p_1 and q_1 into the same collection because they can possibly point to the same location. Consequently, the top nodes referenced by p_1 and q_1 are merged, indicating that p_1 and q_1 should reference the same location as shown in Fig. 7(c).

The boolean expression "p==NULL" should evaluate as false. Thus, consider the boolean expression of the form "p<>NULL", which excludes the case where both p_1 and q_1 will be NULL. The top node referenced by both p_1 and q_1 (of course, also referenced by 1_x_0 and 1_y_0) is changed to the node labeled with NOT-NULL which must be named. Name candidates include 1_p_1, 2_x_0, 2_y_0, and 1_q_1. It does not matter which one is used. Fig. 7(d) shows the situation where 2_x_0 is selected as its name. Similarly, one can evaluate "q<>NULL".

Consider the situation where the boolean expression "q==NULL" must evaluate to true. Suppose that one wants to exercise the path <1,2,3,4,6,7>. The analysis will show that statement 7 represents dead code. To cause the traversal of statement 7, the current instance of q, q_1, should be NULL. This will cause a contradiction since the current state as shown in Fig. 7(d) requires that q_1 should not be NULL. The result is $\perp_\sigma$ which indicates the detection of an inconsistent path.

Fig. 7(e) shows the points-to information that r_1 points to z_0 introduced immediately after evaluation of the assignment 8. Consequently, the variable z_0 will be defined at the assignment 9 indirectly. In addition, the function GP generates new versions of r and z: r_2 and z_1. As a result,

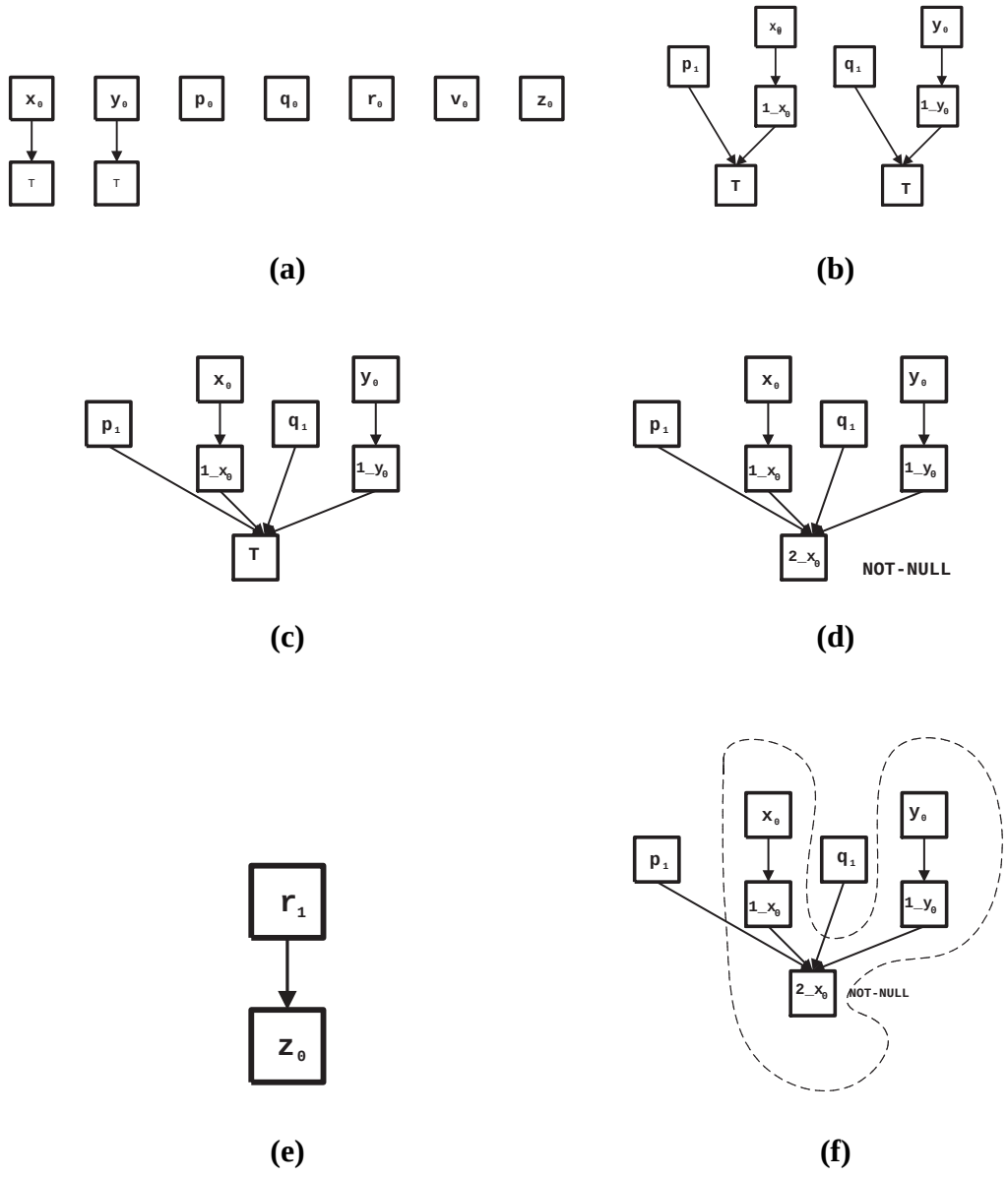

Figure 7: Shape graphs generated by the shape analysis algorithm when applied to the given path of the example program in Fig. 6: (a) depicts the initial state σ_0, (b) depicts the shape graph after evaluating sub-path <1,2>, (c) depicts the shape graph after evaluating sub-path <1,2,3>, (d) depicts the shape graph after evaluating sub-path <1,2,3,4>, and (e) shows the points-to relation arisen after evaluating assignment 8. The part enclosed in the dotted line in (f) shows the shape of the input data structure that can cause the traversal of the target path <1,2,3,4,6,8,9,10,11> in Fig. 6.

the assignment is converted into SSA form without a pointer dereference as follows: z_1==10. Conversion of the boolean expression "z==v" into SSA form is simply "z_1==v_0".

The last statement to consider is "*q=v". Its evaluation is carried out in the same manner as that of assignment 9 by using the points-to information of q. Since q_1 points to 2_x_0, it is converted into the SSA form as follows: 2_x_0==v_0.

Of particular interest is the portion of the final shape graph that is associated with initial input (pointer) variables, because it gives a shape of the input data structure required to traverse the selected path. "Initial input variables" refers to the versions of input variables before any modification; they have 0 as their subscript number. The partial shape graph enclosed by dotted lines in Fig. 7(f) shows the shape of the input data structure required to traverse the selected path.

After finding a shape of the input data structure required to traverse the selected path, one needs to find values for input variables of non-pointer types. Such values are found by solving the constraints generated from the selected path. Having the constraints, one can apply various methods to come up with a solution [3, 9]. The constraints for the example are the following: $z_1 == 10$, $z_1 == v_0$, and 2_$x_0 == v_0$. The solution is z_1:10, v_0:10, 2_x_0:10.

Variable v_0 is key because it represents the input value of v. The resulting form of the input data structure is shown enclosed by the dotted line in Fig. 7(f), and the value of the formal parameter v is 10.

4 Heap-Based Data Structure

So far, the focus has been on pointers that reference statically-allocated memory objects (typically stacks). In this section, the approach is extended to heap-directed pointers, which reference dynamically allocated objects. Heap-directed pointers often involve structures. A structure has multiple fields, each of which is accessed using field identifiers. For the sake of simplicity, assume that each field can either be an integer, a pointer to another structure or NULL.

4.1 Transfer functions for structures and heap-directed pointers

To support structures and heap-directed pointers, transfer functions deal with the statements or expressions of the following forms:

- x.f=expression
- k=x.f
- x.f rel k where rel is one of $\{>, <, <>, ==, <=, >=\}$[2]
- p$\rightarrow$f=k
- k=q$\rightarrow$f
- p=malloc(-)
- free(p)

[2]If k is of pointer type, rel will be one of $\{<>, ==\}$.

- $p \rightarrow f$ rel k

Before defining transfer functions associated with these statements or expressions, one must assign SSA numbers to structures. Since every field in a structure must be treated as a separate variable, SSA numbers are associated with all fields as well as with the structure itself. SSA numbering is adapted from the rules developed by Lapkowski and Hendren [16] as follows:

Rule 1 Assignment to a field increments the SSA number associated with the field. For example, consider the assignment of the form "x.f=..." If the field f has k as its SSA number, then evaluation of the assignment increments the SSA number. Thus, the SSA number of f will be $k+1$ after the assignment. However, the SSA number associated with the structure, x, remains unchanged.

Rule 2 A structure copy assignment changes the SSA number associated with the structure. For example, consider "x=y" where x and y are variables of structure type. If the SSA number associated with x is k, then it will be $k+1$ after evaluation of the assignment. In this case, the SSA numbers associated with its fields remain unchanged.

Fig. 8 shows a program fragment illustrating how to assign SSA numbers to structures. Note that only the structure copy assignment generates a new version of a variable of structure type.

struct foo {	struct foo {
int f;	int f;
int g;	int g;
} x, y;	} x, y;
x.f=10;	$x_0.f_1$=10;
x.g=20;	$x_0.g_1$=20;
y.f=x.f;	$y_0.f_1$=$x_0.f_1$;
x.f=30;	$x_0.f_2$=30;
y=x;	y_1=x_0;
x.g=y.f;	$x_0.g_2$=$y_1.f_1$;
(a)	(b)

Figure 8: A program fragment (a) and its SSA numbering for structures (b)

The transfer functions associated with the first four forms do not differ from those associated with ordinary assignments, since a variable of the form x.f can be regarded as a separate variable. From now on, assume that the function GP is changed to cope with structures accordingly.

Fig. 9 shows the transfer functions for some of the statements involving the pointer dereference operator '$\rightarrow$', and the functions for allocating and releasing memory locations: malloc() and free() function. The first two forms involving the pointer dereference operator '$\rightarrow$' can be transformed into the forms without pointer dereferences using the points-to information, as done for the pointer dereference operator '*'.

If a pointer variable references a node labeled with $\top$, then a memory location is materialized from the top node. For example, suppose that x is a pointer variable such that $\sigma(L_\sigma(x)) = \top$.

$$
\begin{array}{lcl}
[\![y = x \rightarrow f]\!]\sigma & = & [\![y == \text{new}-\text{name}(x,\sigma).f]\!]GP(y,\sigma_x) \text{ if } \sigma(L_\sigma(x)) = \top \\
& = & [\![y == m_x.f]\!]GP(x,\sigma) \text{ else if } \sigma(L_\sigma(x)) \text{=NOT-NULL or } \sigma(L_\sigma(x)) \text{=} l_{m_x} \\
& & \text{if } \sigma(L_\sigma(x)) \sqcap \text{NULL} = \bot \\
& = & \bot_\sigma \text{ otherwise} \\
\\
[\![x \rightarrow f = y]\!]\sigma & = & [\![\text{new}-\text{name}(x,\sigma).f = y]\!]GP(x,\sigma_x) \text{ if } \sigma(L_\sigma(x)) = \top \\
& = & [\![m_x.f == y]\!]GP(m_x,\sigma) \text{ else if } \sigma(L_\sigma(x)) \text{=NOT-NULL or } \sigma(L_\sigma(x)) \text{=} l_{m_x} \\
& = & \bot_\sigma \text{ otherwise} \\
\\
[\![p = \text{malloc}(-)]\!]\sigma & = & [\![p = \&\text{heap\_loc}]\!]\sigma \\
\\
[\![\text{free}(p)]\!]\sigma & = & [\![p = \text{NULL}]\!]\sigma
\end{array}
$$

Figure 9: Transfer functions of statements involving pointer dereference operator $\rightarrow$, free() and the malloc() function.

Also assume that the collection $[x]_\sigma$ is the singleton set $\{L_\sigma(x)\}$. Then, the invisible variable $1\_L_\sigma(x)$ can represent the structure materialized from the top node, and "x$\rightarrow$f" is replaced with "$1\_L_\sigma(x)$.f". This can be viewed as a process of concretizing the shape of a data structure from the 'primordial soup' [17]. The other transfer functions shown in Fig. 9 are similar to the transfer functions associated with the statements "y=*x" and "*x=y".

Fig. 9 also includes the transfer function for the statement 'p=malloc(-)' which creates a new location to be referenced by p. It creates an anonymous object that needs a name. A name is created by using the location in the program, prefixed by the word "heap_".

Consider the following sequence of code:

1: p=malloc(-);

2: q=p;

3: p=malloc(-);

The sequence of code can be converted int SSA form as follows: p1=&heap_1; q1=p1; p2=&heap_3;. An analysis of the code sequence finds that p1 and q1 point to the same heap object named 'heap_1', but p2 points to the heap object named 'heap_3'.

The memory release function "free(p)" returns the memory location referenced by p to the heap. Without loss of semantic information, this is equivalent to saying that p does not point to anything after "free(p)". Thus, the transfer function associated with the assignment "p=NULL" can replace "free(p)".

Finally, consider the boolean expression of the form "p$\rightarrow$f rel k". The transfer functions associated with the boolean expression of the above form can be defined in a manner similar to those associated with the assignments of the form "p$\rightarrow$f=k" (or "k=p$\rightarrow$f"). That is, the points-to information about what the pointer variable p is referencing is used to transform the boolean expression into a boolean expression without the pointer dereference operator $\rightarrow$.

4.2 Example

Fig. 10 shows an example program from Korel [7]. Suppose that one wants to identify a shape of the input data structure that will traverse the path <1,2,3,4,7,8,3,4,7,9,3,4,5,6,3>.

```
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
typedef struct Node *NodePointer;

void Find(NodePointer L, int y, NodePointer q) {
    NodePointer p;
1:  p = L;
2:  q = NULL;
3:  while (p != NULL) {
4:      if (y == p->data) {
5:          q = p;
6:          p = NULL;
        }
        else {
7,8:        if (y < p->data) p = p->left;
9:          else p= p->right;
        }
    }
}
```

Figure 10: An example program from Korel [7]

Fig. 11(a) shows the portion of the shape graph that corresponds to the initial state where the analysis starts. The following discussion concerns only the relevant portion of the state.

Fig. 11(b) shows the shape graph after evaluating the sub-path <1,2>. Pointer variable p points to whatever L is pointing to after evaluating "p=L" and q has the value NULL after evaluating "q=NULL"(this is not explicitly shown in Fig. 11(b)). The subscripts of p and q have been incremented by one because the assignments define them.

The next expression to evaluate is "p<>NULL". The result should be a maximal state that evaluates the expression to true. Such a state can be obtained by materializing a concrete location from the top node pointed to by p_1 and L_0 as depicted in Fig. 11(c). Whenever a location of structure type is created, the approach ensures that all its pointer fields can possibly reference any place by initializing them with $\top$. The newly created location is labeled with 1_L_0 using the notion of invisible variable.

Evaluation of "y<>p→data" does not affect the points-to information, but generates a constraint without the pointer dereference as follows: "y_0<>1_L_0.data". Similarly, "y<p→data" is also transformed into "y_0<1_L_0.data".

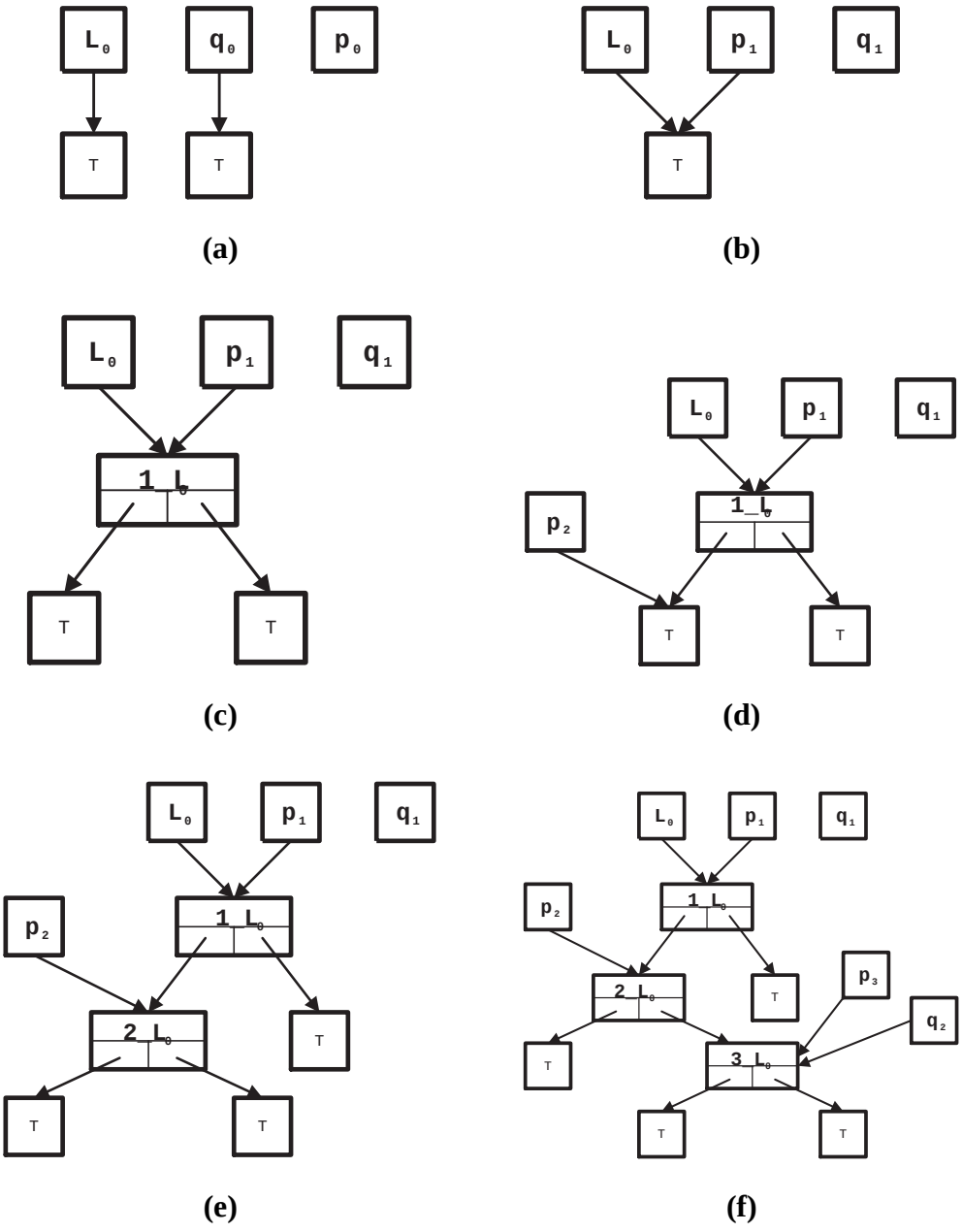

Figure 11: The shape graphs generated by the shape algorithm when it is applied to the given path of the example program in Fig. 10; (a) depicts the initial state σ_0, (b) depicts the shape graph after evaluating the sub-path <1,2>, (c) depicts the shape graph after evaluating the sub-path <1,2,3>, (d) depicts the shape graph after evaluating the sub-path <1,2,3,4,7,8>, (e) depicts the shape graph after evaluating the sub-path <1,2,3,4,7,8,3,4,7>, and (f) depicts the shape graph after evaluating the target path <1,2,3,4,7,8,3,4,7,9,3,4,5,6,3>.

The next statement to consider is "p=p→left". Because p_1 points to a location named 1_L_0 (see Fig. 11(c)), the assignment is transformed into the SSA form "p_2==1_L_0.left" yielding the state depicted in Fig. 11(d).

The next statement is "p<>NULL". Evaluation of the boolean expression ensures that the location pointed to by the current instance of p, i.e., p_2 should not be NULL. Thus, the top node pointed to by p_2 needs to be materialized. The newly created node is named `1_1_L0.left` according to the naming scheme. For simplicity, the name is shortened to 2_L_0 using the mapping table. The result is the shape graph depicted in Fig.11(e).

The evaluation continues with "y<>p→data" and "y≥p→data", which do not affect the points-to information. However, they generate an additional constraint "y_0>2_L_0.$data_0$" to be satisfied by the data field of the structure pointed to by p_2.

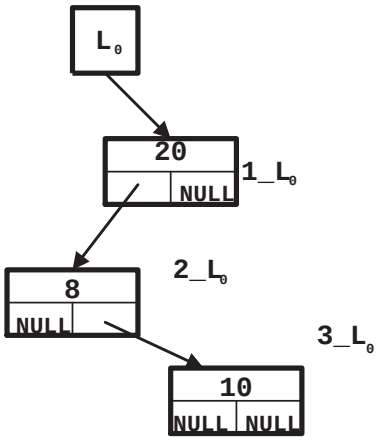

Figure 12: A shape graph of the input data structure

Consider "p=p→right" and "p<>NULL". The net effect is to create a (NOT-NULL) node to be referenced by p_3 and 2_L_0.$right_0$. The newly created node is named `1_1_1_L0.left0.right0`, which is shortened to 3_L_0. Now the sub-path <4,5,6> is evaluated. The boolean expression "y==p→data" is transformed into the equality "y_0==3_L_0.$data_0$". The sequence of the assignments "`q=p;p=NULL;`" can be evaluated without any difficulties. The result is shown in Fig. 11(f).

The pointer fields in the input data structure shown in Fig. 11(f) are set to NULL to keep the input data structure as simple as possible. In addition, the values of each data field of the structures are generated by solving the following constraints:

- y_0<1_L_0.$data_0$
- y_0>2_L_0.$data_0$
- y_0==3_L_0.$data_0$

Assuming that 10, 20, 8, and 10 are the input variable y and the data fields, respectively, this generates the input data structure shown in Fig. 12.

5 Interprocedural Shape Generation

Extending shape generation to programs with procedure calls does not require significant effort. One way to accomplish interprocedural shape generation is to use inline substitution that replaces a procedure call with a copy of the invoked procedure and select a complete path after inlining and apply our shape generation algorithm to it.

This approach requires a few modifications to the technique described so far, but it does not make use of any testing information on the called procedures even when available. To test a procedure P which calls a procedure Q that has already been tested, one should be able to reuse test cases for Q. If Q does not have any test cases which are necessary to test P, additional test cases should be developed for an adequate testing of P.

Describing the approach precisely requires some definitions. Let $S\nabla\sigma$ be the state restricting itself to the points-to relationships in σ which have a member of S as the first element in each pair. Let $\sigma\uparrow_k X$ be a set of locations that are directly or indirectly accessible from each location in X with respect to σ and have k as their SSA number; if k is not specified, the latest SSA versions will be assumed.

This section first describes how to represent test cases developed for a procedure and then gives the process of testing a procedure with procedure calls.

5.1 Modeling parameter binding and procedure call

First, the approach models the effects of executing a procedure call. This paper considers only call-by-value parameter binding as employed in C.

Using call-by-value, the called procedure can alter the contents of a variable in a calling procedure by passing a pointer to that variable as a procedure argument. However, the value of the pointer, (i.e., the address of the variable referenced by the pointer) will not be affected during the procedure call.

For example, consider the procedure in Fig. 13 that will change a variable indirectly. When the procedure is called by "assignVar(p, 4)", it would change the content of the variable pointed to by the pointer `p` to 4. Note that the assignment at program point 3 does not affect the variable referenced by `p`.

```
void assignVar(int *x, int v) {
1:      *x=v;
2:       x=(int *)malloc(sizeof(int));
3:      *x=v+1;
}
```

Figure 13: An example program for modeling call-by-value binding

If call-by-reference semantics are used, the location referenced by `x` should have the value `v+1`. The resulting information is propagated back to the call-site, giving the conclusion that the content of the variable pointed to by `p` should be 5, rather than 4. This is due to the implicit assumption that the actual parameter `p` and the formal parameter `x` are the same location in call-by-reference semantics. In order to support call-by-value semantics, however, it must be explicit that actual parameters and formal parameters represent distinct locations when analyzing a procedure.

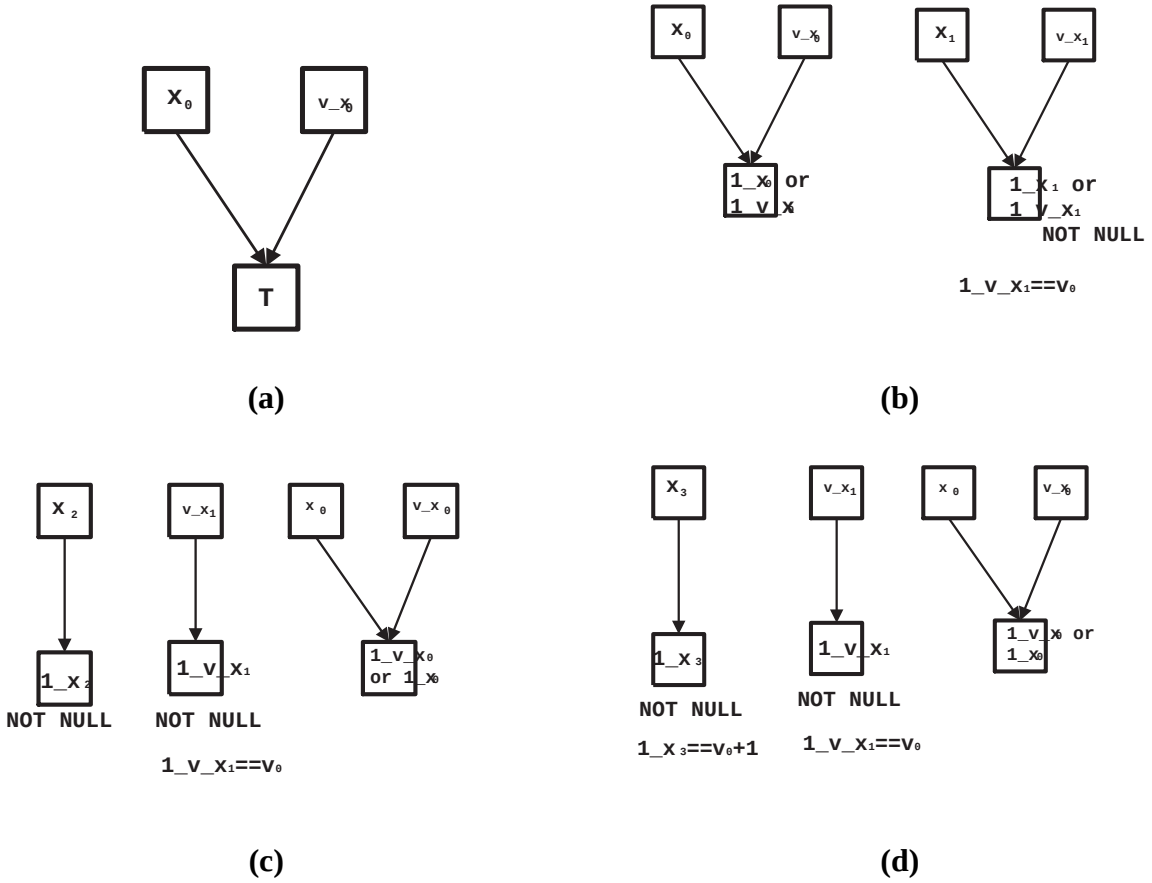

Figure 14: The shape graphs generated by the modified shape generation algorithm when applied to the example program in Fig. 13; (a) depicts the initial state σ_0, (b) depicts the shape graph after evaluating sub-path <1>, (c) depicts the shape graph after evaluating sub-path <1,2>, and (d) depicts the shape graph after evaluating path <1,2,3>.

To address the problem, this paper introduces *virtual parameters*, which represent actual parameters of pointer type. The analysis proceeds in the same manner as the analysis technique without virtual parameters except that it uses an alternative method of building the initial state σ_0. Before analyzing a given path in a procedure, a virtual parameter, denoted by $\texttt{v\_f}_\texttt{i}$ is generated for each formal pointer $\texttt{f}_\texttt{i}$ in such a way that $\texttt{f}_\texttt{i}$ and $\texttt{v\_f}_\texttt{i}$ belongs to the same collection and initially point to $\top$. That is, $\sigma_0(\texttt{f}_\texttt{i}) = \sigma_0(\texttt{v\_f}_\texttt{i}) = \top$ and $[\texttt{f}_\texttt{i}]_{\sigma_0} = [\texttt{v\_f}_\texttt{i}]_{\sigma_0}$. Fig. 14(a) shows the initial state built using the formal pointer $\texttt{x}_0$ and the corresponding virtual parameter $\texttt{v\_x}_0$.

Unlike call-by-value semantics, however, call-by-reference semantics do not require the notion of virtual parameters which has been introduced to explicitly indicate that actual parameters and formal parameters denote distinct locations. Thus, if call-by reference semantics are used, virtual parameters are not needed any more and the analysis can be done in terms of only formal parameters.

Recall that the state information σ_π after evaluating a given path $\pi = < \mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{n} >$ includes all information concerning the shape of the input data structure along the path π in terms of points-to relationships between pointer variables. Since a procedure call is treated as an atomic operation, two concerns must be addressed. One concern is the shape of the input data structure that should be passed as input to the called procedure in order to traverse the path π, which is denoted by $\sigma_{\mathtt{I}_\pi}$. The other concern is the effect of the procedure call on the input data structure, which is denoted by $\sigma_{\mathtt{o}_\pi}$. The (local) effects of all the sub-paths $< \mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{i} > (\mathrm{i} < \mathrm{n})$ of π do not need to be considered for interprocedural shape generation although they are embodied in σ_π. $\sigma_{\mathtt{I}_\pi}$ and $\sigma_{\mathtt{o}_\pi}$ are the entry state and the exit state of σ_π, respectively.

It is simple to extract $\sigma_{\mathtt{I}_\pi}$ and $\sigma_{\mathtt{o}_\pi}$ from σ_π. If the procedure P has the formal pointer parameters $\texttt{f}_1, \ldots, \texttt{f}_\texttt{m}$, then $\sigma_{\mathtt{I}_\pi}$ and $\sigma_{\mathtt{o}_\pi}$ can be computed as follows: $\sigma_{\mathtt{I}_\pi} = (\sigma_\pi \uparrow_0 \{\mathrm{v\_f}_{1_0}, \ldots, \mathrm{v\_f}_{\mathrm{m}_0}\}) \nabla \sigma_\pi$ and $\sigma_{\mathtt{o}_\pi} = (\sigma_\pi \uparrow \{\mathrm{v\_f}_{1_0}, \ldots, \mathrm{v\_f}_{\mathrm{m}_0}\}) \nabla \sigma_\pi$.

For example, the rightmost part of Fig. 14(d) depicts the input data structure required to traverse the path $< 1, 2, 3 >$ of the procedure in Fig. 13. The middle part of Fig. 14(d) shows the effect of the procedure call on the input data structure.

Consider function calls of the form `x=foo(...)` where the function `foo` should have at least one occurrence of the return statement. For each function `foo` returning a pointer type variable, a global variable `ret_foo` is defined with the same type as `foo`. Using this newly defined variable, `return(r)` is expanded into "`ret_foo=r; return`" and the function call `x=foo(...)` is expanded into: "`foo(...); x=ret_foo`". As a result, the formulation of σ_{o_π} is modified as follows:

$$\sigma_{\mathtt{o}_\pi} = (\sigma_\pi \uparrow \{\mathrm{v\_f}_{1_0}, \ldots, \mathrm{v\_f}_{\mathrm{m}_0}, \mathtt{ret\_foo}_1\}) \nabla \sigma_\pi$$

Now consider the constraints on input values that are needed to traverse the path π and the constraints on output values. The constraints on input values and output values are denoted by $\zeta_{\mathtt{I}_\pi}$ and $\zeta_{\mathtt{o}_\pi}$, respectively. Both $\zeta_{\mathtt{I}_\pi}$ and $\zeta_{\mathtt{o}_\pi}$ are strongly related to $\sigma_{\mathtt{I}_\pi}$ and $\sigma_{\mathtt{o}_\pi}$. Rephrased in terms of $\sigma_{\mathtt{I}_\pi}$ and $\sigma_{\mathtt{o}_\pi}$, $\zeta_{\mathtt{I}_\pi}$ is the set of constraints on the data fields of the locations defined in $\sigma_{\mathtt{I}_\pi}$ and formal parameters of non-pointer type while $\zeta_{\mathtt{o}_\pi}$ is the set of constraints on the data fields of the locations defined in $\sigma_{\mathtt{o}_\pi}$ and a return variable of non-pointer type if it exists.

For example, consider the path $\pi = < 1, 2, 3 >$ through the procedure in Fig. 13. The constraints $\zeta_{\mathtt{I}_\pi}$ and $\zeta_{\mathtt{o}_\pi}$ correspond to `true`[3] and $\texttt{1\_v\_x}_0 == \texttt{v}_0$, respectively as shown in Fig. 14(d). With the help of virtual parameters, the (wrong) constraint depicted in the leftmost part of Fig. 14(d) can be ignored.

[3] `It means that there are no constraints to be applied.`

Assume that a test case for a path π is represented by a 4-tuple $(\sigma_{I_\pi}, \zeta_{I_\pi}, \sigma_{o_\pi}, \zeta_{o_\pi})$. Then, if a procedure P is tested according to a certain test criterion that selects a set of paths, say $\{\pi_1, ..., \pi_m\}$, then the test suite for P will be as follows:

$$\bigcup_{k=1..m} \{(\sigma_{I_{\pi_k}}, \zeta_{I_{\pi_k}}, \sigma_{o_{\pi_k}}, \zeta_{o_{\pi_k}})\}$$

When the path π_k is obvious from a context, the discussion will usually omit it.

5.2 Strategy for interprocedural shape generation

This section describes an interprocedural strategy for a procedure which contains a procedure call. Suppose that procedure P contains a procedure call which calls a procedure Q, and Q has been tested using a test suit $TS_Q = \{TC_1, ..., TC_m\}$ where $TC_i = (\sigma_{I_{\pi_i}}, \zeta_{I_{\pi_i}}, \sigma_{o_{\pi_i}}, \zeta_{o_{\pi_i}})$ for i=1,..,m. Let σ_c be a state after evaluating the sub-path which reaches a call-site. σ_c is also a calling context. Then, the input data structure, denoted by $\widehat{\sigma_c}$, that will be passed to the called procedure can be extracted from σ_c as follows:

$$\widehat{\sigma_c} = (\sigma_c \uparrow \{L_{\sigma_c}(a_1), ..., L_{\sigma_c}(a_m)\}) \nabla \sigma_c$$

The interprocedural shape generation is performed in the following steps. The first step is consistency checking — determine if the shape of the input data structure described by $\widehat{\sigma_c}$ can be accepted by Q. Next, if consistency checking fails, the called procedure is reanalyzed to find the path in the called procedure that can be executed with $\widehat{\sigma_c}$. Then, build a shape of the input data structure that performs the procedure call successfully by using $\widehat{\sigma_c}$ and σ_{I_π}. Finally, the effect of the procedure call is incorporated into the calling context. Variables a_1, ..., a_m represent actual parameters and $(v\_)f_1$, ..., $(v\_)f_m$ represent the corresponding (virtual) formal parameters.

5.2.1 Consistency checking

The input data structure $\widehat{\sigma_c}$ that reaches the call-site and an entry state $\sigma_{I_{\pi_i}}$ of a called procedure are consistent if they possibly describe the same data structure. The following formula determines if $\widehat{\sigma_c}$ is consistent with $\sigma_{I_{\pi_i}}$:

$$\widehat{\sigma_c} \sqcap \sigma_{I_{\pi_i}}(L_{\sigma_c}(a_1)/v\_f_{1_0}, ..., L_{\sigma_c}(a_m)/v\_f_{m_0})$$

where $\sigma(x_1/y_1, ..., x_m/y_m)$ represents a state obtained by renaming y_i with x_i in σ and the names of the locations reached through y_i are changed accordingly. Here the assumption is that the actual parameters represent distinct locations, i.e., $a_i \neq a_j$ for $i, j = 1..m$.

The above formula maps the callee's name space to the caller's at each call site because each procedure has its own distinct name space. If the formula evaluates to $\perp_\sigma$, it means that the calling procedure does not pass the input data structure required to traverse the path, i.e., π_i, of the called procedure for the entry state $\sigma_{I_{\pi_i}}$. Consistency checking is repeated for all entry states of test cases until finding an entry state that is consistent with σ_c.

For example, consider the input data structure given in Fig. 15(f) which traverses the path π =<1,2,3,4,7,8,3,4,7,9,3,4,5,6,3> of the procedure in Fig. 10 and call it S. It is easy to see that S2, S3, and S4 given in Fig. 15(b), Fig. 15(c), and Fig. 15(d) are consistent with S whereas S1 given in Fig. 15(a) is not.

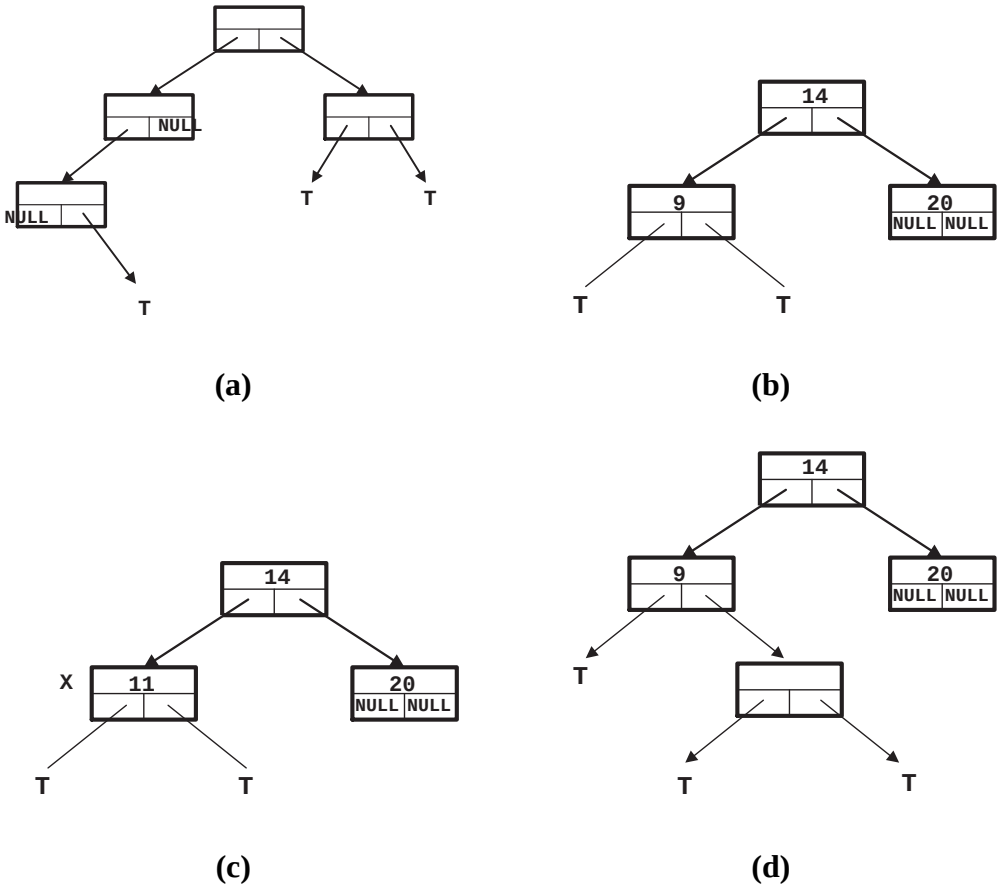

Figure 15: Input data structures passed to the procedure given in Fig. 10: (a) S1 (b) S2 (c) S3 (d) S4

However, this does not necessarily mean that the input data structures S2, S3, and S4 that are consistent with S are able to execute the test path π. Even though S2 is consistent with S, whether or not it executes π depends on the calling context. For example, suppose that the procedure is called by Find(p,x,NULL) where p is a pointer to the data structure shown in Fig. 15(c) and the value of the actual parameter x is assigned 10. Call-by-value semantics assigns the value of the actual parameter x to the corresponding formal parameter y. Recall the constraint $y_0 > 2\_L_0.data_0$ (see Section 4.2) which requires that the value of the data field of the node labeled with X of the data structure in Fig. 15(c) should be less than the value of the formal parameter y. Applying the constraint gives $10 > 11$, which clearly evaluates to false.

This example shows that consistency checking should also be done on the constraints in the data fields of the data structure. Formally stated,

$$\zeta_c \wedge \zeta_{I_{\pi_i}}(L_{\sigma_c}(a_1)/v\_f_{1_0}, ..., L_{\sigma_c}(a_m)/v\_f_{m_0})$$

where ζ_c is the constraint on the data fields of the data structure which will be passed to the called procedure.

5.2.2 Reanalyzing the procedure

What happens if no entry states are consistent with the input data structure given by the calling context, i.e., $\widehat{\sigma_c}$? Such a case does not necessarily mean that the called procedure does not have any path that can be traversed by the input data structure given by $\widehat{\sigma_c}$.

For example, consider a procedure call alias_foo(x,y,z) where x and z point to exactly the same location. A reanalysis of the called procedure alias_foo is needed if there exist no entry states of the called procedure that are consistent with the calling context. Recall that when a procedure is first analyzed, calling contexts are not considered under the assumption that no alias relationships exist among its input pointers. Thus, if no statements in the called procedure force those two pointers to point to the same location, the procedure will not include the entry state matching the calling context, and then one will have to reanalyze the procedure for the calling context. Reanalysis is done by constructing the initial state σ_0 such that σ_0=$\widehat{\sigma_c}$ when the analysis of the procedure is begun.

5.2.3 Building input data structure

An entry state $\sigma_{I_{\pi_i}}$ that is consistent with $\widehat{\sigma_c}$ requires the derivation an input data structure shape with two states in common. This is done by merging the two states as follows:

$$\widehat{\sigma_c} \sqcap \sigma_{I_{\pi_i}}(L_{\sigma_c}(a_1)/v\_f_{1_0}, ..., L_{\sigma_c}(a_m)/v\_f_{m_0})$$

The result is a shape of the input data structure that will perform the procedure call successfully.

For example, Fig. 15(d) can be regarded as the data structure that is derived by merging the data structures S2 in Fig. 15(b) and Fig. 11(f). The resulting input data structure is added to the test suite for the procedure for later use.

5.2.4 Incorporating the effects of the procedure call into the calling context

A procedure call can be characterized as a pair (σ_c, σ'_c) where σ_c is a calling context that reaches the call-site and σ'_c is the state after the procedure call. This step aims at computing σ'_c from σ_c.

The state σ'_c consists of two (sub)states: σ'_{c1} and σ'_{c2}. σ'_{c1} concerns the data structure which will be passed as input to the procedure in order to carry out the procedure call successfully, while σ'_{c2} concerns the effects of the call on the data structure. They are computed as follows:

$$\sigma'_{c1} = \sigma_c \odot (\widehat{\sigma_c} \sqcap \sigma_{I_{\pi_i}}(L_{\sigma_c}(a_1)/v\_f_{1_0}, ..., L_{\sigma_c}(a_m)/v\_f_{m_0}))$$
$$\sigma'_{c2} = \sigma_c \odot \sigma_{0_{\pi_i}}(L_{\sigma'_c}(a_1)/v\_f_{1_0}, ..., L_{\sigma'_c}(a_m)/v\_f_{m_0})$$

where the SSA number of $L_{\sigma'_c}(a_i)$ is $k+1$ if the SSA number of $L_{\sigma_c}(a_i)$ is k. Since the data structures passed to the called procedure might be changed during the call, new versions of SSA variables for the actual parameters are created. The result follows:

$$\sigma'_c = \sigma'_{c1} \odot \sigma'_{c2}$$

The constraints on the data fields of the data structure propagate back to the call-site, i.e., immediately after the procedure call statement as follows:

$$\zeta'_c = \zeta''_c \wedge \zeta_{0_{\pi_i}}(L_{\sigma'_c}(a_1)/v\_f_{1_0}, ..., L_{\sigma'_c}(a_m)/v\_f_{m_0})$$

where $\zeta''_c = \zeta_c \wedge \zeta_{I_{\pi_i}}(L_{\sigma_c}(a_1)/v\_f_{1_0}, ..., L_{\sigma_c}(a_m)/v\_f_{m_0})$.

5.3 Precision, Efficiency, and Applicability of the Approach

The approach must achieve a balance between precision and efficiency to be both practicable and effective. One way to increase the precision of interprocedural analysis is to perform a context sensitive analysis. A context sensitive analysis can distinguish all calling chains and will reanalyze the callee for all distinct calling paths. For example, if procedure P1 calls procedure P2 two times, which in turn calls procedure P3 three times, then procedure P3 will be reanalyzed six times with different calling contexts. Even though this kind of analysis results in precise results due to the use of calling contexts, it might explode as the number of the calling paths increases.

To limit the number of paths that must be analyzed, the new approach computes procedure summary information for each procedure. To build its summary, each procedure is analyzed once for a set of paths, which will be chosen according to a certain testing criterion. A procedure summary is represented as a set of pairs (entry state, exit state), as defined in Section 5.1. Informally, each entry state describes an input data structure which will traverse a path selected from the procedure while each exit state describes the effect of the procedure on the corresponding entry state. When a procedure call to P is encountered, consistency checking determines if the current calling context matches any of the pairs (entry state, exit state) in procedure Pś summary. A matching entry state indicates that a reanalyze of the called procedure P for the calling context is not necessary. Then the summary information can be reused when analyzing the calling procedure without additional effort.

An alternate approach is used when no entry states match the current calling context: the called procedure is reanalyzed for the calling context. The reanalysis result is added to the procedures summary in order to reuse it in other calling contexts, thus avoiding analysis explosion.

An inherent problem with the SSA form is that there can potentially be hundreds or thousands of executions of of a single assignment statement, since the method generates a new SSA variable for every instance. From the perspective of program testing, this situation is not likely to actually occur. Suppose that we have to select program paths which can meet a certain testing criterion

(i.e., statement coverage or branch coverage). There are likely to be very few paths that consist of hundreds or thousands of statements in the selected paths.

The actual effectiveness and practicability of the approach can only be determined by applying it to real programs. The following section describes an initial empirical evaluation.

6 Empirical Evaluation

The empirical evaluation consists of three parts. First, a proof-of-concept intraprocedural shape generation tool (SGEN) demonstrates that the method can be implemented, and supports further evaluation. Second, SGEN is applied to sample programs to show that it can determine if a path is feasible, produce the appropriate input data shapes to traverse feasible paths, and detect program faults. Finally, SGEN is applied to generate input data shapes for ten paths in each of three programs to evaluate execution time performance.

6.1 The SGEN tool

SGEN demonstrates the feasibility and properties of the approach for programs written in a subset of the C language. The subset includes a limited set of data types: int, arrays of int, user-defined struct types, (multi-level) pointers to int and user defined struct types. SGEN supports pointer assignments, (in)comparison, allocation and deallocation operations.

SGEN has three components: a path selector, a shape generator, and a component to visualize shapes. The path selector takes as input a text file containing a C program and guides the user to select a path. The shape generator takes as input a path given by the path selector and generates a shape of the input data structure for the traversal of the selected path if the path is consistent. The shape generator also produces a set of constraints to be solved by classical constraint solving systems that yield the values in the data fields of the input data structure and/or the values of the input parameter of non-pointer type if they exist.

The shape visualizer displays the input data structure graphically so that the user can easily observe the topology of the shape, i.e., nodes and their connections. Fig. 16 shows a snapshot of the data object shape produced by SGEN. A user can readily generate test data that matches the shape displayed by the visualizer.

SGEN generates a suitable shape of the path incrementally. Even though a subpath rather than a complete path is given, the tool can generate a suitable shape for the traversal of the subpath.

6.2 Applying SGEN to evaluate feasibility, find shapes and faults for paths

SGEN is applied to the example C programs given in Section 3.3 and Section 4.2 and from Visvanathan and Gupta [18] to demonstrate the effectiveness of the approach. The objective is to answer two questions: (1) can the approach determine if a given path is feasible? and (2) does the approach simultaneously produce a desirable shape of the input data structure and the constraints on the data values in the input data structure when the path is feasible? The following functions are analyzed: the example programs in Fig 6 and Fig 10, and a linked list deletion function that was based on a program from Visvanathan and Gupta [18] which deletes a node if it has the same data value in the data field as the input parameter. SGEN analyzed a set of paths that cover the branch coverage criterion for each program.

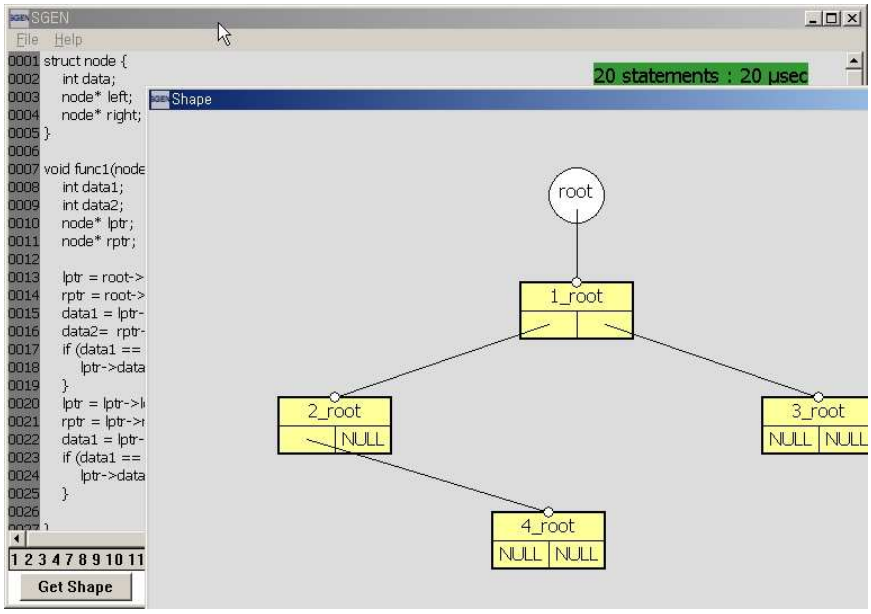

Figure 16: A snapshot of a data input shape produced by SGEN and displayed by its shape visualizer

The program in Fig 6 contains two infeasible paths. SGEN detected the infeasibility of both paths successfully. SGEN also successfully generated the desired shapes of the input data structures for all feasible paths examined and the constraints on the data fields in the input data structure and on the input parameters of non-pointer type.

The input data structure shape generated by SGEN helped the investigators to detect one program fault in the list deletion program. The visual representation of the shape along with the constraints on the input data structure led the investigators to the fault. The list deletion program begins by processing the second node without checking the first node. Fig. 17 shows part of the list deletion program.

A traversal of one of the test paths requires an input linked list consisting of three nodes. Values in the data fields of the first two nodes cannot equal the input parameter value “d”, while the value of the third node must equal “d”. Given the path, SGEN showed the three node linked lists as desired, but generated only the constraints on the data fields of the last two nodes. This indicates a program fault — the first node is actually ignored by the program while traversing the list.

6.3 Shape generation performance

To evaluate performance, SGEN generated shapes for three additional programs: a doubly linked list insertion function (DLL), a binary search tree deletion function (BST) and a polynomial addition function (Poly) using linked lists. These functions represent typical programs of moderate complexity using user defined structures. For each function, ten feasible paths of varying lengths were selected randomly, giving a total of 30 separate paths. Thus, SGEN ran 30 trials, which were executed on a Pentium 4, 2.8 GHz machine running Windows 2000. The average processing time was recorded for each trial along with the number of statements on the path, number of nodes generated to construct the input data structure, and number of assignments involving dereference

```
...
void delete(node* root, int d) {
    node* prev;
    node* curr;
    prev = root;
    while(prev->next != NULL) {
        curr = prev->next;
        data = curr->data
        if(curr->data == d) {
        ...
        }
        else {
        ...
        }
    }
    ...
}
```

Figure 17: A part of the list deletion procedure that was adapted from a similar program in Visvanathan and Gupta [18]

operations in the path.

Table 1 shows the results for the 30 trials. The results are also plotted in Figure 18. The average processing time vs. the number of statements is plotted in Figure 18(a). The average processing time vs. the number of nodes generated and the number of pointer dereference operations in the path are plotted in Figure 18(b) and Figure 18(c) respectively. The relationship between the number of pointer dereference operations in the path and the number of nodes generated to construct input data structures that will traverse the target paths is plotted in Figure 18(d)

The plots in Figure 18(a) through (c) show that the processing times of paths increase almost linearly with three factors: number of statements, number of nodes generated, and number of dereference operations. The three plots in Figure 18(a) — the plots with the number of statements on the x-axis — appear to clearly fit a linear model. However, the plots have different slopes. A linear model does not fit as well for at least one plot in the graphs using the number of nodes and number of deference operations in the x-axis (Figures 18(b), (c), and(d)). The plots demonstrate that processing a fairly long path can be done without an abrupt rise in processing time, and suggests that the approach will scale up to larger programs.

Though nearly linear, the relationships between processing time and (a) the number of statements and (b) the number of nodes are notably different for each of the three programs (DLL, Poly, and BST). In contrast, the relationship between processing time and the number of pointer dereference operations is notably more consistent between the programs. This suggests that the number of dereference operations is the most dominant factor affecting shape generation. Hence SGEN is likely to have better time performance on a comparatively long path with few statements that involve pointer dereference operations, than on a shorter path involving (relatively) many pointer dereference operations.

Consider paths P7 and B7 in Table 1(b) and Table 1(c). The paths are of similar length (P7 has

Table 1: Performance of shape generation trials for paths in three functions

(a) Doubly linked list insertion function

Path	# of Stmt.	# of nodes	# of Deref. Oper.	Time(μsec)
D1	7	1	3	143.2
D2	17	2	8	314.9
D3	27	6	16	599.5
D4	33	8	20	779.9
D5	45	12	28	1153.0
D6	57	16	36	1539.6
D7	66	19	42	1928.2
D8	72	22	48	2264.8
D9	84	26	56	2767.0
D10	93	29	62	3042.8

(b) Polynomial addition function

Path	# of Stmt.	# of nodes	# of Deref. Oper.	Time(μsec)
P1	13	2	2	148.6
P2	28	4	8	406.5
P3	30	4	9	433.6
P4	39	4	13	592.4
P5	46	4	13	685.5
P6	55	4	9	855.1
P7	62	4	14	978.8
P8	71	5	23	1102.5
P9	81	7	32	1342.0
P10	91	8	34	1538.0

(c) Binary search tree deletion function

Path	# of Stmt.	# of nodes	# of Deref. Oper.	Time(μsec)
B1	10	2	3	159.4
B2	20	4	10	438.6
B3	28	5	14	545.7
B4	33	6	17	723.0
B5	43	8	23	946.8
B6	53	10	29	1209.0
B7	63	12	35	1460.0
B8	73	14	41	1768.1
B9	83	16	47	2014.6
B10	93	18	53	2342.2

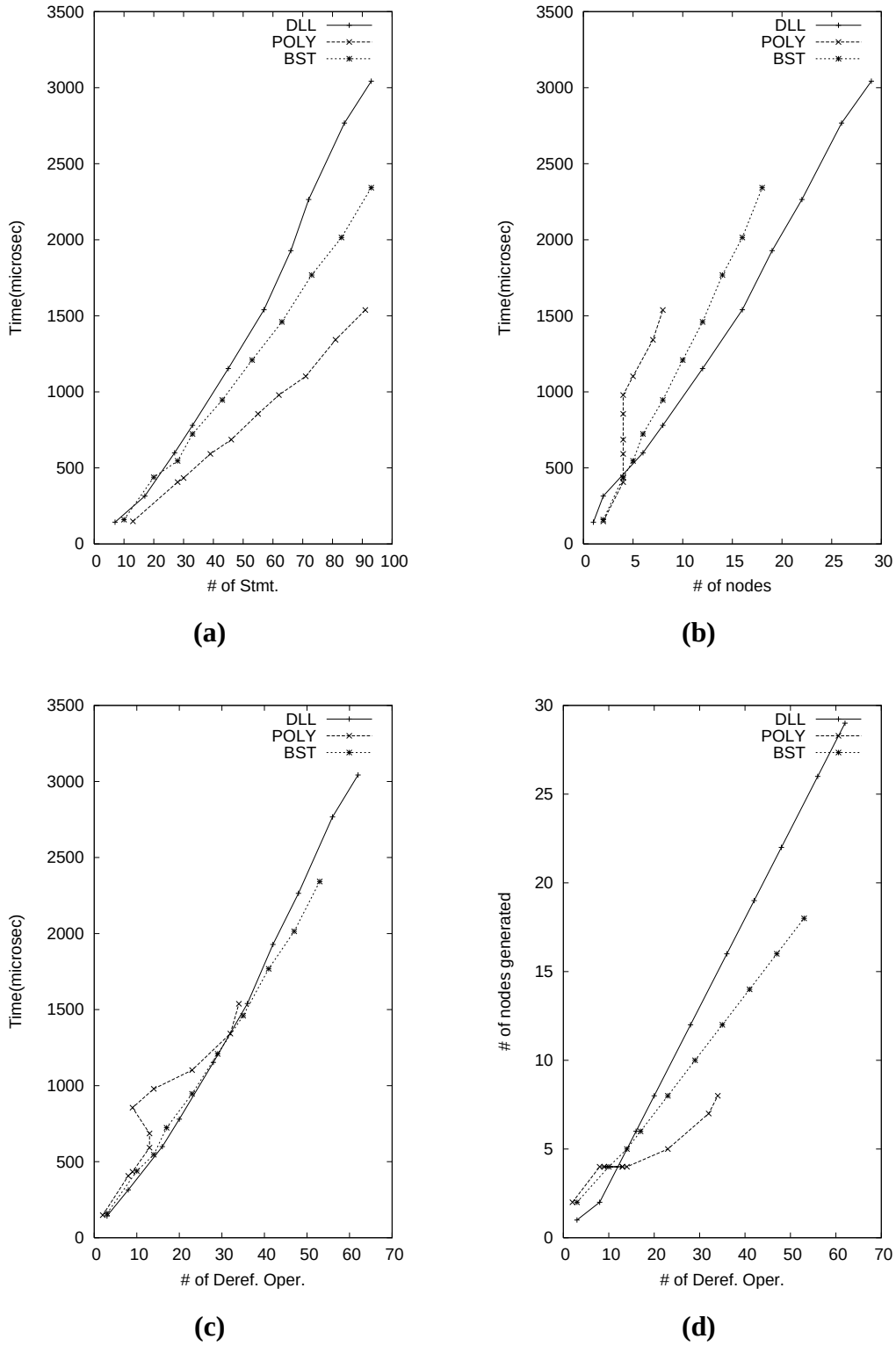

Figure 18: Performance vs. path size for trial programs. Plots (a) - (c) show time on the y-axis and path size measures on the x-axis with # of statements in plot (a), # of nodes generated in plot (b), # of dereference operations in the test path in plot (c). Plot (d) shows the relationship between # of dereference operations and # of nodes of input data structures.

62 statements and B7 has 63 statements), while the ratio of their processing times is about 1.6($\approx 1460.0/978.8$). Clearly, path size does not solely determine processing time, since the proportion of statements involving pointer dereference operations will vary between paths.

The plots in Figure 18(a) show that SGEN gives superior time performance when analyzing POLY than when applied to DLL and BST. SGEN performs better on POLY because the polynomial addition function involves fewer pointer dereference operations than the other two functions. Furthermore POLY requires relatively simple input data structures to traverse the selected paths compared to DLL and BST. For example, consider paths D10, P10, and B10 shown in Table 1. While they are of almost same sizes, there are significant differences in the number of nodes of input data structures that are needed to test the corresponding paths.

The relationship between the size of the required test input data structures (i.e., the number of nodes generated) and processing time provides further insight. In general, an increase in the number of dereference operations will increase the number of nodes generated. However all dereference operations do not involve creating new input data structures nodes, because some dereference operations might manipulate pre-existing nodes. The plots in Figure 18(d) show that the increase in the number of dereference operations in POLY is slower than that in DLL and BST. This is because Poly has very few dereference operations which directly contribute to creation of nodes in the paths. Also, the number of nodes does not necessarily lead to the fast increase of the processing time. For example, consider paths B4, P10, and D5 in Table 1. A traversal of these paths requires the creation of 8 new nodes. However, their processing times vary significantly. This is another indication that shape generation is directly proportional to the number of the dereference operations in the path.

6.4 Limitations

Like all small-scale empirical studies, there are risks to the external validity of the results. One limitation is that the programs evaluated are small, and cannot represent all programs involving dynamic data structures. The feasibility evaluation does not demonstrate that the new method will be effective on all programs. Further work is clearly needed to demonstrate that the new method will be effective in general. Particularly, tests were not conducted on large-scale commercial software systems. Reducing external validity threats will require studies of samples of the "universe" of programs. Such sampling is very difficult, since it would require an accurate characterization of the universe of programs.

Similarly, while the studies demonstrate that the new method can reveal program faults, the studies do not comprehensively evaluate the capability of finding various classes of program faults involving pointers.

The trials that measure the time complexity of the algorithms suggest that the time to generate input data shapes increases linearly with respect to the number of dereferencing operations on a path. However, this represents the performance of a small sample of paths. Clearly there will be cases where generating input shapes will require greater computation times, since the worst case time complexity is of $O(|\pi| \times |v|^2)$, where π is the number of statements and expressions in the path and *v* is the number of variables in the path. However, the results reported here suggests that these cases are rare. Only larger scale empirical studies can determine the true average case complexity and the likelihood of encountering cases that require long computation times. Of course, total computation times could increase greatly when testing complex programs since testing would

involve many more paths than when testing simple programs.

A lack of benchmark data and standard test sets makes it difficult to empirically compare the new method with related work. The two most closely related methods either did not provide any empirical results, or did not provide specific enough information to make direct comparisons. Related work is discussed in the next section.

7 Related Work

Although there are many approaches for test data generation, research on shape generation for programs with pointer inputs and heap-based data objects has received little attention. This section first reviews test data generation for programs without pointers and then reviews results of prior work on generating shapes of the input data structure in the presence of pointers. Finally, a comparison to shape analysis used in compiler optimization and parallelization is given.

7.1 Test data generation without pointers

Classical test data generation for programs without pointers are based on symbolic execution [1, 6]. Symbolic execution derives a system of algebraic constraints from a given path in terms of symbols representing any input values that can traverse the path. Solutions to the system of algebraic constraints become test data that cause the execution of the path. Often symbolic execution requires complex algebraic manipulation to simplify constraints and detect the infeasibility of the path.

Execution-based approaches for automatic test data generation requires the actual execution of the program. Both Gallagher and Narasimhan [2] and Roger and Korel [10] formulate test data generation as a function minimization problem. They treat each branch predicate on the given path as a function that becomes minimal when the desired outcome is produced. Whenever a certain branch predicate on the given path does not produce the desired outcome, the input value is modified in the direction of minimizing the function associated with that predicate. This process is repeated until all the branch predicates along the path evaluate to the required outcomes. These techniques consider just one input variable and one branch predicate at a time. This may be inefficient, in particular for infeasible paths.

Offut et al [11] presented a test generation technique called *dynamic domain reduction* for unit testing. It combines many techniques including symbolic execution, constraint-based testing and execution-based test data generation. While, in general, execution-based approaches take an initial value for each input variable, the dynamic domain reduction procedure (DDR) is initially given a set of values (i.e., domain), and as branches are taken in the selected path, the domains for the variables are reduced to make the predicates true for any assignment of values from the domain. Once the DDR procedure is finished, the domains for the input variables contain test data that will cause execution of the path. If the path is infeasible or the initial domains given for input do not include values that will execute the path, the variables' domains will be empty. Even though the DDR procedure limits itself to non-pointer types, its dynamic nature can handle arrays and loops more efficiently and more accurately than static test data generation techniques that do not take use of run-time information about programs.

Gupta et al [4, 5] presented another interesting approach based on program execution. This approach uses numerical analysis techniques to search for an input that traverses a path and provides the amounts to adjust input value if the current input does not exercise the given path. Unlike the

approaches that use function minimization algorithms, this technique takes into account all branch predicates and all input variables at one time. Thus, the number of program executions required is independent of the path length, but the size of the system of linear constraints to be solved may increase with the number of branch predicates on a path.

7.2 Shape generation in the presence of pointers

There are two prior approaches to shape generation in the presence of pointers: a dynamic approach and a two phase approach. The dynamic approach proposed by Korel [7] requires program execution for automatic test data generation. Backtracking generates the shape of the input data structure as well as the values of the input variables (of non-pointer types) including the values in the fields of the input data structure. This approach monitors the program execution flow to determine if the intended path was taken. If not, it backtracks to a program point where the incorrect decision about the shape of the input data structure and the data values was made and manipulates input (pointer) values so that the intended branch is executed. Thus, many iterations can be required before a suitable shape and data values are found. Furthermore, it can be extremely inefficient when the given path is infeasible. Korel does not describe any empirical results to compare with the new approach. However, the empirical analysis of the new approach uses Korel's example programs as data.

Visvanathan and Gupta [18] independently presented a similar algorithm for generating a shape of the input data structure. Their algorithm is called a two-phase approach because it first generates the shape of the data structure for functions with pointer inputs and then generates the integer and real values in the data fields of the data structure. In the first phase, i.e., the shape identification phase collects constraints on the pointer values along a given path and solves the constraints to find a suitable shape to traverse the path. The approach also collects aliasing information to use in the second phase, the data value generation phase.

Visvanathan and Gupta provide empirical results analyzing two programs — a "linked list search function" and a "binary search function". They found a linear relationship between computation time and the number of statements, number of nodes, and number of constraints. Among a larger data set, the empirical evaluation of the new approach reported in this paper used a function that was adapted from one used by Visvanathan and Gupta. The empirical results for the new method are similar to those reported by Visvanathan and Gupta. Unfortunately, direct comparisons between the empirical results for the two systems is difficult. Visvanathan and Gupta do not provide the complete programs or specify the paths used in the study. Although both empirical studies report linear computation time growth, the new approach offers several advantages.

First, the two phase approach is inefficient when all statements along a given path do not use any pointers and the first phase becomes useless. Even when there are no pointers involved, the approach produces constraints on the data values of non-pointer types, which can be readily solved through classical constraint solving systems. Second, the two phase approach may require additional work to get solutions to the constraints on the data fields in the input data structure through classical constraint solving systems. Unfortunately, most of constraint solving systems are not designed to take aliasing information into account. It implies that one must resolve aliases in the constraints before submitting them to constraint solving systems. The new approach collects the points-to information for each pointer variable during evaluation of each program point along the path and generates the constraint system, where the aliasing problem is already resolved. Third,

in contrast to the two phase approach (where it is only possible to solve the constraints on the data values of non-pointer types only after the first phase which collects the aliasing information is completed), the new approach can solve constraints on the data fields in the input data structure independently of shape generation. It implies that one can detect an infeasible path with inconsistent constraints on the data values more efficiently, even if a suitable shape of the input data structure can be generated for the path. For example, consider the assignments $p_1 \rightarrow f = 10$ and $p_2 \rightarrow f=20$. If p_1 and p_2 appears to point to the same memory location named N, the new approach will transform the assignments to the constraints as follows: N.f==10 and N.f==20. The constraints are obviously inconsistent. Recognizing such inconsistencies prevents wasting time trying to find solutions, even though a shape of the input data structure for the given path is found.

7.3 Shape analysis

Because test data generation for programs with pointers concerns shape identification of data structure, it may be asked whether one can use shape analysis techniques developed for applications such as compiler optimization and parallelization [17]. For each program point, these techniques can determine shapes of data structures that could result from executions of all possible paths that reach the program point. The compiler techniques are not designed to give the information about the shapes of input data structure that will cause the execution of a program path. Furthermore, because all possible execution paths are considered, the compiler techniques inherently generate overly conservative information for programs that can allocate unbounded memory objects. Thus, unbounded data structures need to be *summarized* in some finite way. Due to summarization, it is possible to give shapes of input data structure that can not traverse the selected path. Note that for test case generation, one only needs a shape of an input data structure that can traverse the selected path rather all possible shapes of input data structures. Since a program path is assumed to be finite, one does not have to summarize unbounded data structures.

8 Conclusions

Most work in automated test data generation focuses on finding input values for non-pointer types. Handling pointers is crucial to test data generation for programs written in procedural languages such as C. This paper presents a static approach to determining a shape of the input data structure required to cause the traversal of a selected path in the presence of pointers.

The approach separates the shape generation problem from test data generation for non-pointer types. As a result, determining input values for non-pointer types can be performed independently. The method collects aliasing information along the path used to generate the constraints after the aliasing problem is resolved. This reduces the burden of transforming the constraints to a form that classical constraint solving systems can accept as input.

An empirical evaluation demonstrates the practicability and utility of the new approach. The SGEN tool implements the technique for a subset of the C language. SGEN can determine if a given path is feasible, and will produce input data shapes to traverse a feasible path. It can also identify program faults when generated constraints do not match required shapes. A performance evaluation indicates that the time required to generate input data structure shapes grows linearly with respect to the size of a path, which suggests the method will scale up to large systems. The

number of pointer dereference operations on a path is the best indicator of the time required to generate an input shape.

While the research offers improvements over past work, there are open problems for further study. First, the SGEN tool needs to be extended to handle array elements separately. Currently, an array is regarded as a single object rather than element-by-element basis. This is because the tool is based on classical SSA form which ignores array indices and handles an assignment to an array element as an assignment to the entire array [19]. Thus it is possible to lose a certain degree of precision when input values for arrays are computed. However, because other research has developed methods to treat each array element separately, similar mechanisms can be added to SGEN in near future.

Next, the approach needs to be extended to support pointer arithmetic. Pointers do not have to point to single variables. They can also reference the cells of an array. With a pointer referencing into an array, one can start doing pointer arithmetic. Pointer arithmetic avoids the need to introduce an extra variable for the array indices. However, pointer arithmetic can complicate the identification of a suitable shape of the input data structure because of the presence of overlapping pointers that reference the same array at different offsets. For example, consider the pointer arithmetic `p=q+i` where `p` and `q` are pointer variables. In order to cope with such pointer arithmetic, the new approach needs to be extended to infer that `p` points to the same physical array as `q` but by an offset of `i` bytes. Dor, Rodeh, and Sagiv [20] demonstrate such an analysis to detect overlapping pointers. The new approach can potentially incorporate the work of Dor et al to address pointer arithmetic.

The new test generation method can potentially be extended for use in testing object-oriented programs. Test cases for object-oriented programs are usually represented in terms of object configurations. Since object configurations are affected by message passing, one will need to characterize each method in terms of object configurations as done here for interprocedural analysis. The new method, with extensions, should work for languages with pointers or references such as C++ or Java.

Acknowledgements This research was financially supported by Hansung University in the year of 2007.

References

[1] L. A. Clarke, "A system to generate test data and symbolically execute program," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 215–222, 1976.

[2] M. J. Gallagher and V. L. Narasimhan, "ADTEST: A test data generation suite for Ada software systems," *IEEE Transactions on Software Engineering*, vol. 23, no. 8, pp. 473–484, 1997.

[3] B. B. A. Gotlieb and M. Rueher, "Automatic test data generation using constraint solving techniques," in *Proceedings of ACM ISSTA*, pp. 53–62, 1998.

[4] N. Gupta, A. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method," in *Proceedings of ACM SIGSOFT Foundations of Software Engineering*, pp. 231–244, 1998.

[5] N. Gupta, A. Mathur, and M. L. Soffa, "UNA based iterative test data generation and its evaluation," in *Proceedings of 14th IEEE Int. Conf. on Automated Software Eng.*, pp. 224–232, 1999.

[6] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Transactions on Software Engineering*, vol. 3, no. 4, pp. 266–278, 1977.

[7] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[8] S. Lapierre, E. Merlo, G. Antoniol, R. Fiutem, and P. Tonella, "Automatic unit test data generation using mixed-integer linear programming and execution trees," in *Proceedings of International Conference on Software Maintenance*, pp. 189–198, 1999.

[9] C. Meudec, "ATGen:automatic test data generation using constraint logic programming and symbolic execution," in *Proceedings of IEEE/ACM Int. Workshop on Automated Program Analysis Testing and Verfication*, pp. 22–31, 2000.

[10] F. Roger and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering Methodology*, vol. 5, no. 1, pp. 63–86, 1996.

[11] J. Offutt and J. Pan, "The dynamic domain reduction approach to test data generation," *Software-Practice and Experience*, vol. 29, no. 2, pp. 167–193, 1997.

[12] I. S. Chung, "A static approach to automated test data generation in the presence of pointers," in *Proceedings of the 19th Int. Symp. on Computer and Information Sciences, LNCS 3280*, pp. 887–896, 2004.

[13] M. Emami, R. Ghiya, and L. Hendren, "Context sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN 1994 Conference on Progmming Language Design and Implementation*, pp. 242–256, 1994.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[15] N. Gupta, A. Mathur, and M. L. Soffa, "Generating test data for branch coverage," in *Proceedings of 15th IEEE Int. Conf. on Automated Software Eng.*, pp. 219–228, 2000.

[16] C. Lapkowski and L. J. Hendren, "Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers," tech. rep., School of Computer Science. McGill University Canada, 1996.

[17] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 1, pp. 1–50, 1996.

[18] S. Visvanathan and N. Gupta, "Generating test data for functions with pointer inputs," in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pp. 149–160, 2002.

[19] S. Rus, G. He, C. Alias, and L. Rauchwerger, “Region Array SSA,” in *Proceedings of the 15th ACM International Conference Parallel Architectures and Compilation Techniques*, pp. 43–52, 2006.

[20] R. M. Dor, N. and M. Sagiv, “Cleanness checking of string manipulations in C programs via integer analysis,” in *Static Analysis Symposium*, pp. 194–212, 2001.