2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering

# Test Data Generation for Derived Types in C Program

Zheng Wang†, Xiao Yu†, Tao Sun†, Geguang Pu†, Zuohua Ding‡and JueLiang Hu†§
†*Shanghai Key Laboratory of Trustworthy Computing, Software Engineering Institute, East China Normal University, Shanghai, 200062, China*
‡*Center of Math Computing and Software Engineering, Institute of Science, Zhejiang Sci-Tech University, Hangzhou, Zhejiang, 310018, China*
§*The Institute of Logic and Cognition, Zhongshan University, Guangzhou, 510275, China*
{*wangzheng, lesteryu, suntao, ggpu*}*@sei.ecnu.edu.cn, zouhuading@hotmail.com*

## Abstract

*Test data generation is one of the important tasks during software testing. This paper proposes an approach to generating test cases automatically for the unit test of C programs with derived types including pointers, structures and arrays. Our approach combines symbolic execution and concrete execution. The approach captures operations on variables precisely by concrete execution, and thus it is capable of handling derived types. Benefited from symbolic execution, accessing variables as array index can be solved by a substitution strategy. The substitution strategy also translates a path constraint involving variables of derived type to the one containing only primitive variables. An implementation of this approach is integrated into our test case generation tool called CAUT* [1]*.Experimental results show that our approach is effective to generate test data for derived types.*

## 1. Introduction

Nowadays, the most commonly used techniques for validating the quality of software are testing and verification. Despising verification is rarely used in "large-scale" software development, testing is widely applied, normally in an ad-hoc manner. Testing is a labor intensive process, and occupies about over half the total cost during software development and maintenance. For the time being, some tools such as JUnit [1] and Visual Studio's new team server [2] which focus on test execution or test management and have achieved ideal effects. However these tools pay little attention to automatic test data generation, especially the test data generation for complex data structures. Some other tools from industry can generate test data but failed to fully satisfy the branch-coverage test criterion [3].

In this paper, we describe an approach to generate test data automatically for complex data structures based on our previous test case generation framework [4] which can achieve a high rate of branch coverage for the unit under test. Our approach automates test data generation process by combining the concrete and symbolic execution of the tested program dynamically. The mechanism used by us can be referred to *concolic testing* or *dynamic symbolic execution* [5].

1. freely available at http://caut.lab205.org

### 1.1. Background

The most widely used approach to generating test data, from our point of view, is random testing [6]. In this approach, test cases are generated at random to check the potential errors of programs. However, these approaches hardly guarantee correctness, and the program behaviors being covered take up a very small portion compared to all the behaviors of the program. These weakness above limits the applications of random testing technique.

To overcome the weakness of random testing, several techniques have been proposed. Symbolic execution introduced in 1970's [7] is a well-known program analysis technique to generating test data automatically. It is a way to analyze the behavior of program for all possible inputs. Instead of supplying the concrete inputs to a program under test, symbolic execution supplies symbols that represent arbitrary values. Constraint-based test data generation(CBT) [8] is able to reach a special branch in a program under test by building a constraint system associated with the given branch and solving the system with the help of domain reduction techniques.

Dynamic symbolic execution [9] is proposed to intertwine concrete and symbolic execution together that analyzes program behaviors dynamically. Profiting from concrete execution, our approach managed to overcome some practical obstacles such as alias, and it is the foundation to generate test data for complex data structures such as pointers and arrays. Our approach is based on dynamic symbolic execution.

978-0-7695-3757-3/09 $25.00 © 2009 IEEE
DOI 10.1109/TASE.2009.10

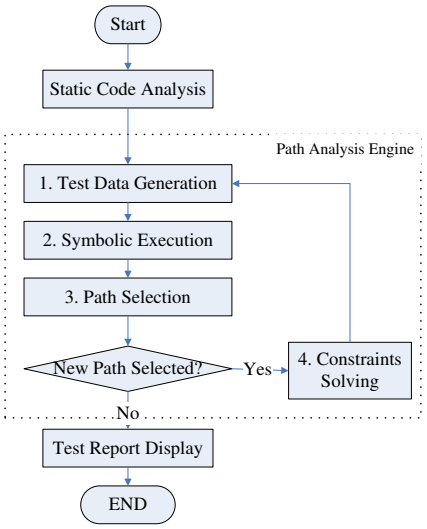


Figure 1. Test Data Generation Framework

### 1.2. Overview

The framework of CAUT is illustrated in Figure 1. The first stage *static code analysis* analyzes and refactors the source codes of program under test. Then refactored program is executed symbolically in the *path analysis engine*, which is the essential part of the framework. The process of this framework can be explained as as follows:

- Static analysis technique is used to collect and record the data information for path analysis engine.
- Based on symbolic execution and constraint solving, the test cases are generated iteratively till a coverage criteria has been reached.
  - Generating test data from the output of solving path constraints collected during the last execution.
  - Using symbolic execution to record path constraints.
  - After an iteration of execution, selecting a new path to be processed. If no more new paths can be selected, the whole process terminates.
  - Solve path constraints.
- Reporting errors found during the execution.

In this paper, we focus on the phases *constraints solving* and *test data generation* shown in Figure 1.

### 1.3. Contributions

The follows are the contributions of this paper:

**Trace Model and Logical Memory Map**: Symbolic execution requires using symbols to replace concrete value of variables so that the behaviors of program under test can be kept tracking. A trace model is introduced to hold the changes on symbolic values of variables. Logical memory map is used to represent derived types in uniform.

**Test data generation for derived types**: There are two tasks to generate test data for pointers intuitively. The first one is to construct a proper memory storage to which the pointer points, and the second is to generate the value stored in that memory location. The difficulty is that several pointers may point to the same variable. This problem is solved by building a relation matrix of pointers on the base of the path constraints collected from dynamic symbolic execution.

Structure and array are both a set of primitive variables in essence. Thus, our approach just divides them from primitive variables and thereafter the test data generated from each part will be combined based on the logical memory. One interesting point in this work is that our approach can deal with the case in which the array index is a variable. A variable as an array index may cause combination explosion. A substitution strategy is proposed to handle this problem. Although this approach cannot thoroughly solve this problem, the experimental result is still exciting.

**Implementation and Evaluation**: The tool CAUT has implemented test data generation for derived types. This implementation handles not only derived types, but also function call and recursion etc. To evaluate the efficiency of our approach, we employ CAUT to test several subjects that contain complex data structures. The experimental results show that our approach achieves a higher branch coverage than CUTE [10], a tool for test case generation in many cases.

### 1.4. Structure of The Paper

The paper extends our test case generation framework to support pointer, array and structure variables. This paper is organized as follows. In the following section, the motivating example is presented to illustrate our brief idea. The framework of our approach is discussed and its implementation is introduced in section 3. To prove the efficiency of our approach, experiments are performed and the results are demonstrated in section 4. Section 5 discusses the related works and Section 6 concludes this paper.

## 2. Motivating Example

We use a simple example to illustrate how our approach is effective on test case generation. Consider a C function $testme$ shown in Figure 2. There is an error location required to cover in the function and the statement can be reached given some specific data of input. The input data of $testme$ consists of the values of the parameters: integer array $a$, integer $x$ and $y$.

Because of indirectly referenced data access referenced, traditional symbolic execution will encounter a practical

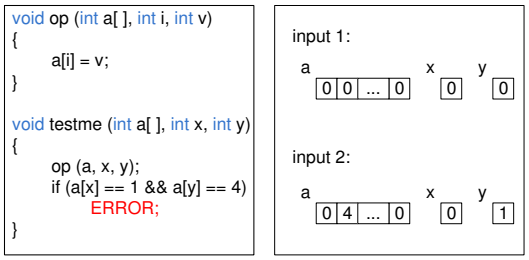


Figure 2. Example C function under test and test data generated by CAUT

obstacle. Here, it is unknown which element of the array $a$ is referenced to by $a[i]$, $a[x]$ and $a[y]$. A similar problem also arises in pointer reference because of the potential aliasing.

To deal with this case for $testme$ function, we firstly initialize the values of all arguments to be 0. Then $testme$ is executed in symbolic and concrete way dynamically. During the execution, the logical memory is established and branch constraints are collected. In the first iteration of execution, the function $testme$ takes the false branch and does not cover the expected statement. Then a path selection algorithm [11] is applied and it tries to generate test case by solving collected constraints. The constraint is $a[x] = 1 \wedge a[y] = 4$. The effect of assignment in function $op$ is recorded in logical memory, and thus the latest definition $y$ of $a[x]$ will be used to replace its occurrence in this constraint. So that the constraint is transformed to $y = 1 \wedge a[y] = 4$. Since $a[y] = 4$ involves unknown array element for the array index variable $y$, our approach replace $y$ with a constant in the range from 0 to the length of array $a$. In this case, the only constant value can be taken is 1, for there is a constraint which makes $y$ satisfy $y = 1$. At last, our approach generates the test data with $y = 1, a[1] = 4$, and the values of other variables are all 0. With this input test data, the next iteration of execution of $testme$ covers the expected branch in the program under test.

## 3. Our Approach

In this section, we present our approach to generate test cases for derived types. Firstly, we design the logical memory to represent data used by symbolic execution. Then we present how our approach solve path constraints from the information collected during symbolic execution. Finally, we describe how test data is generated based on the logical memory and solutions from path constraints.

### 3.1. Trace Model

Our approach keeps tracking the behaviors of the program under test using *trace model*. Here, $v \in \mathbb{V}$, $e \in \mathbb{E}$, and $\mathbb{V} \subseteq \mathbb{E}$. In this definition, $\mathbb{E}$ denotes expressions used in C program and $\mathbb{V}$ is the set of variables.

$$trace ::= \langle\rangle \mid \langle\ (v,e)\ \rangle^\frown trace \mid \langle\ (\epsilon,e)\ \rangle^\frown trace \mid \langle\ (v,\epsilon)\ \rangle^\frown trace$$

A trace keeps the behavior of one execution of a program, and each item in the trace denotes an operation of the program execution. $\langle v, \epsilon\rangle$ means variable $v$ is an input parameter, $\langle v, e\rangle$ indicates variable $v$ is assigned by expression $e$, and $\langle \epsilon, e\rangle$ shows the expression $e$ is used as a branch condition. We take the motivation program as an example, the trace after its first iteration of execution is as follows:

$$\langle (a,\epsilon), (x,\epsilon), (y,\epsilon), (a[0], y), (\epsilon, a[x] = 1 \wedge a[y] = 4)\rangle$$

For the element selection from an item $(v, e)$, the projection functions are defined as follows:

$$\pi_1(v,e) = v, \pi_2(v,e) = e$$

The operation *left restriction* $t \upharpoonright var$ denotes the trace t when restricted to items of which left element is $var$. The law of this operation is shown in table 1.

Table 1. Left Restriction of Trace

| | | |
|---|---|---|
| L1 | $\langle\rangle \upharpoonright var = \langle\rangle$ | |
| L2 | $\langle s^\frown t\rangle \upharpoonright var = \langle s\rangle \upharpoonright var^\frown \langle t\rangle \upharpoonright var$ | |
| L3 | $\langle (v,e)\rangle \upharpoonright var = \langle (v,e)\rangle$ | if $v = var$ |
| L4 | $\langle (v,e)\rangle \upharpoonright var = \langle\rangle$ | if $v \neq var$ |

Similarly, we use $t \upharpoonright expr$ to denote *right restriction* and its law is the same with *left restriction*. These two restriction operations are used in retrieving information from the trace. For example, the path constraints can be gained from the expression $t \upharpoonright \epsilon$ while the input parameter is obtained from $t \upharpoonright \epsilon$. The left restriction can be used in retrieving the latest definition information of a special variable.

Some auxiliary operation on trace is introduced to facilitate illustrating our approach. Operation $last(t)$ returns the last item of a trace $t$. Operation $prefix(t, i)$ returns a prefix of trace $t$ and the returned trace ends as item $i$. The formal definition of $prefix(t, i)$ is as follow:

$$p = prefix(t,i) \Leftrightarrow \exists u \cdot p^\frown u = s \wedge last(p) = i$$

### 3.2. Logical Memory Map

We have to record three relations for variables. The first one is about array and its elements; The second one is about a structure and its fields, and the last one is about a pointer and the memory location to which the pointer points. Here, We use vectors to describe logical addresses. A logical memory map $\mathcal{M}$ is maintained to map logical addresses to their values that are either logical addresses or primitive type values. Assume that $\mathbb{L}$ is the set of logical address and $\mathbb{P}$ is the set of primitive type values, then $\mathcal{M} : \mathbb{L} \rightarrow \mathbb{L} \cup \mathbb{P}$. The map will be updated during symbolic execution when

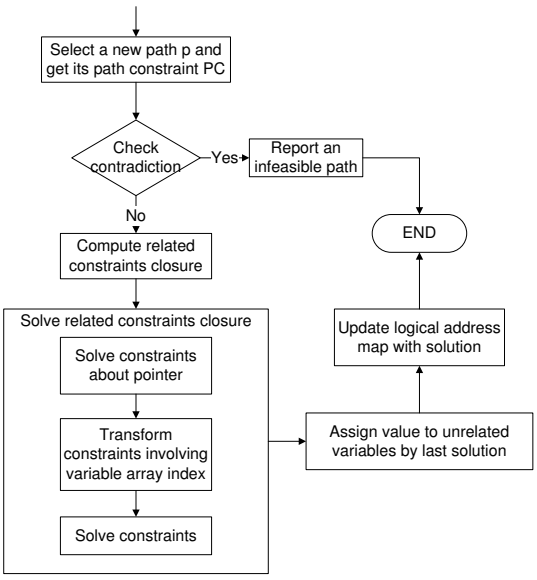


Figure 3. Process of Solving Constraints

there occurs assignment, passes parameter or return value. After path constraints are solved, the solution generated by the solver also updates the value fields of this map.

The reason that our approach applies logical address instead of physical address is that the latter may be varing in different concrete executions because of program dynamic loading and memory dynamic allocation. We use vectors to present logical address instead of physical address because vectors facilitate handling of derived types more naturally.

### 3.3. Constraint Solving

In this subsection, we present how our approach solves the collected path constraints during symbolic execution. In one iteration of symbolic execution, if a path $p$ is performed, the path analysis engine collects its path constraints $PC = \bigwedge_{i=0}^{n-1}(c_i)$, where $c_i$ denotes one constraint (mostly appeared in branch node) and $n$ is the depth of this path. By the path selection algorithm, we will generate a new path to be performed and its path constraint is changed into $PC' = (\bigwedge_{i=0}^{m-1}(c_i)) \wedge \neg c_m, m < n$, where $\neg c_m$ denotes the other branch of node $m$. For the sake of simplicity, we use $PC' = \bigwedge_{i=0}^{m}(c_i)$ to represent the to be solving path constraints. Then, we translate the constraints into a form that a constraint solver can process and check whether $PC'$ is satisfied. If so, a solution is given by the constraint solver and it will be used to generate a real test case for the selected path .

To improve the performance of solving path constraints, some strategies are applied. The general process is shown in Figure 3 and those strategies introduced are discussed below.

```
input:  the path constraints, PC = {c1, c2, ..., cm}
output: mark which constraint is related, marked[1...m],
        related variables set, RVS.

    RVS := vars(cn);
    marked[m] = 1;
    repeat
        RVS' := RVS
        for i from 1 to m
            if (marked[i] ≠ 1) and (vars(ci) ∩ RVS' ≠ Φ )
            then
                marked[i] = 1;
                RVS' := RVS' ∪ vars(ci)
    until RVS = RVS'
```

Figure 4. Compute Related Constraints

Obviously, the new path constraint $PC'$ may be unsatisfied. The contradiction is surely introduced by the constraint $\neg c_m$ for the current path constraint $PC$. If a contradiction is brought by the negation of $c_m$ , we can remove $\neg c_m$ from the candidates of branches. Thus, an optimization procedure *contradiction check* can be taken to check whether $\vee_{i=0}^{m-1}(\neg c_m \wedge c_i)$ is contradictive. As a result, we can judge whether the new path to be performed is an infeasible one or not.

By observing that the solution for the last path constraint $PC$ is also satisfied with path constraint $PC'$ without $\neg c_m$, we do not have to compute the constraints unrelated with $\neg c_m$ again. The solution of $PC$ as the one of unrelated constraints can be applied directly. To implement this optimization, we need to define how to judge two constraints are related or not.

Given a constraint $c$ in path constraints $PC$, we define a function $vars(c)$ to be the set of all variables appeared in $c$. Given two constraints $c_1$ and $c_2$, we define a predicate $related(c_1, c_2)$ to check if $c_1$ and $c_2$ are related:

$$\forall c_1, c_2 \in PC \cdot related(c_1, c_2) \Leftrightarrow (vars(c_1) \cap vars(c_2) \neq \phi) \vee (\exists c_3 \in PC \cdot related(c_1, c_3) \wedge related(c_2, c_3)).$$

The definition of predicate $related$ is recursive and Figure 4 shows an algorithm to compute *related constraints closure*. Every constraint in this closure is related with the constraint $\neg c_m$. The algorithm is certain to terminate because $PC$ is a finite set of constraints.

What we need to do next is to solve the constraints in the closure. Most common constraint solvers do not support constraints involving pointer operation or array access. The following subsections will introduce our approach to solve the constraints of those data structures.

#### Pointer constraints

```
input:   the pointer variables {p[1], p[2],..., p[n]}
         the relationship matrix n * (n + 1)
output:  the values {v[1], v[2], ..., v[n]}

    for i from 1 to n
      if matrix[i][0] = 1 then
        v[i] := 1
    value = 1
    for i from 1 to n
      if v[i] = undef then
        v[i] := value
        value++
      for j from i + 1 to n
        if matrix[j][i] = 1 then
          v[j] := v[i]
```

Figure 5. Solving Constraint about Pointer

In practise, there are mainly two pointer operations which appeared in actual C programs. One is pointer dereference, and the other is pointer comparison.

On the first occasion, instead of handling deference operation directly, a pointer dereference is taken as a whole constraint and the handling is postponed in test data generation phase. Here, we introduce the approach to which deal with pointer comparison in the constraint solving phase.

Unlike relations on variables of integer or real type, there is only one equality relation on pointer variables. In order to accelerate the solving of pointer constraints, a matrix is introduced to hold equality relations among pointer variables, which is shown as follow.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ \dots \\ p_n \end{array} \begin{array}{c} \begin{array}{ccccc} \text{NULL} & p_1 & p_2 & \dots & p_n \end{array} \\ \begin{pmatrix} 0 & \dots & \dots & \dots & \dots \\ 1 & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \end{pmatrix} \end{array}$$

We can solve constraints about pointer variables by assigning value directly through the matrix. Initially, the value of each pointer variable is set as un-defined. Then, each pointer variable which equals the constant $NULL$ is assigned by value 0. Thirdly, for each un-defined pointer variable which will be set a value from a predefined counter, the counter is initialized to be and increases once it encounters an assignment. All the other pointer variables equal to this one are also assigned by the same value. The detailed algorithm is presented in Figure 5.

This adopted technique avoids solving constraints by calling a constraint solver and significantly improves the performance. The time complexity of this algorithm is $O(n^2)$. If we use a straightforward method by calling constraint solvers, the time complexity will be $O(2^n)$.

**Array Constraints**

Operations on array are difficult to tackle with in test data generation as well, especially when there are some variables referred by array indexes. The motivating example in section 2 is such a case.

The basic idea of our approach is to replace the variable index in constraints with constant index, and this idea is demonstrated in the following formula. Assume that $c(a[index])$ is a constraint about an array element $a[index]$ and the $index$ is an expression containing variables, then we transform this constraint to a disjunctive form. Each part in this disjunction is gained by substituting the $index$ with a constant.

$$c(a[index]) \Rightarrow \bigvee_{i=0}^{N-1} (c(a[i]) \wedge index = i)$$

This technique is intuitive but leads to state explosion as well. In practise, there are two problems of this technique. Firstly, the variable $a[i]$ may be defined by other expressions before being used in the conditional expression involving the constraint. The same as the motivating example. The array element $a[x]$ appearing in the conditional expression is defined by variable $y$ before being used. If we solve the constraints directly without considering the definition of $a[x]$, the test data generated from the solution will not lead to the expected execution path. Secondly, the combination of different constraints involving variable array index may be very huge and hardly to be solved. If the path constraints contain $m$ constraints and the length of array is $N$, then in the worst case, we have to solve $N^m$ numbers of constraints to determinate whether the path constraints are satisfied or not.

The first problem affects the correctness of test case generation while the second problem causes this approach impractical. The key to the first problem is to record the definition information about variables. So that it can be solved by our logical memory model, for the define-use chain records the trace of every definition of variables. In the motivating example, the define-use chain shows that the variable $a[x]$ is defined by variable $y$ before it is used in the conditional expression. When solving path constraints, the variable $a[x]$ appeared in the constraint generated from this conditional expression is replaced by variable $y$. Based on the trace model introduced in section 3.1, this strategy can be implemented easily:

$$\pi_2(last(prefix(t, (\epsilon, a[x] = 1 \wedge a[y] = 4)) \restriction a[0]))$$

The operation above returns $y$, which is the latest definition of variable $a[0]$.

For the second problem, it is still hard to solve it precisely with polynomial time complexity. In our approach, we introduce a random algorithm to take a compromise between precision and time cost. For disjunctive constraints transformed from a constraint involving variable array index, some of them are selected to combine with other constraints. If the amount of combination is too huge, parts of them are eliminated randomly. In this strategy, there is a possibility

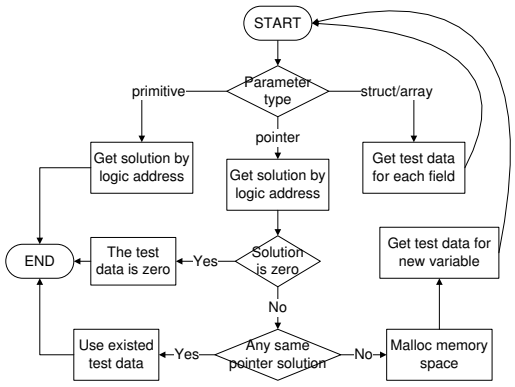


Figure 6. Test Data Generation

that the path constraints are satisfied but the solution is missed. However, as the experimental results shown, the technique works well in practise.

### 3.4. Test Data Generation

After the path constraints are solved, the solution is used to update logical memory map from which the test data is generated. The general process is displayed in Figure 6. The procedure of generating test data is divided into three sub-procedures to deal with primitive type, pointer type, and structure/array type respectively and they are mutual recursion.

#### Primitive Variable

The basic step in the mutual recursion is to generate test data for primitive variable straightforwardly. The value of the variable from logical memory map are obtained directly. If there is no corresponding mapping in the map, test data randomly can be generated. Considering the motivating example in section 2, the input parameter $y$ is a primitive variable and its id is $\langle 2 \rangle$, for it is the third parameter(the number for parameter is zero based). When generating test data for this parameter, we can obtain value from logical memory map, which is 1.

#### Pointer Variable

In the case of generating test data for pointers, the approach is similar to the one of primitive variables, but more complicated. The reason is that we have to handle two parts for a pointer variable: the memory storage it points to and the value stored in that memory storage. Firstly, its solution is obtained from logical memory map. If the value is zero, it indicates that the pointer variable is null. Then the variable is initialized to be null. In the other case, the pointer is not a null pointer. Since the solution is not a real memory address, this value cannot be utilized as test data of the pointer variable. However, we do not allocate memory space

```
struct Node {
    int v;
    struct Node *next;
}
void testme (struct Node *p) {
    while (p != NULL) {
        p = p->next;
    }
}
```

Figure 7. An Example of Constraints about Pointer

for this pointer directly, for different pointers may point to the same memory space. So we find whether there exists any pointer variable of other solutions is the same as the current one. If so, we then check where the pointer variable points to. If it points to a real memory space, there is no need to make memory allocation, and we use the address as test data, otherwise an appropriate memory space is allocated and its address is recorded. Then we recursively generate test data for the new variable which may be complex data structure as well.

We take the program in Figure 7 as an example. Assume that the path constraints are $p \neq NULL \wedge p \rightarrow next \neq NULL$. After solving constraints, the logical memory map is shown in Table 2. At the beginning, the values in third column are $NULL$.

Table 2. Logical Memory Map

| address | value | points |
|---|---|---|
| $< 0 >$ | 1 | 0x448024 |
| $< 0, 1 >$ | 1 | 0x448024 |

In this example, we need to generate test data for parameter $p$, of which logic address is $\langle 0 \rangle$. Given logical memory map, we know its solution is 1 and it does not point to any memory space. Then we find that there is no any other pointer variable with the same value pointing to real memory space. So we allocate memory space for $p$ as test data. Next, we recursively generate test data for the new allocated variable. It is a structure variable whose type is $struct\ Node$. So there are two fields to be filled. The logical address of the first field is $\langle 0, 0 \rangle$, which is not in the logical memory map, so we set its value randomly. The second field is a pointer variable and its logical address is $\langle 0, 1 \rangle$. From logical memory map, we know its solution is 1 and the solution is the same with the variable $p$, so the address of memory space variable $p$ points to is used as the test data for this field.

#### Structure/Array Variable

From the perspective of type system, both structure and array are a set of variables whose memory spaces are continuous. So we apply *divide and conquer* approach to handle such variables. Test data is generated for each field of structure or array separately, and then, we combine all the test data

of fields together into one for the whole structure or array. The case for structure is illustrated in the previous example. We still need to show how to generate test data for array variable with the motivating example.

In this example, the first input parameter $a$ is an array variable and its logical address is $\langle 0 \rangle$. As the previous method shows, we generate test data for $a$ by generating test data for $a[0], ..., a[N]$ separately. Given the logical memory map, test data for $a[1]$, of which id is $\langle 0, 1 \rangle$, is 4 while the others that are not mentioned are generated randomly.

## 4. Experiments

This section presents the experiments by integrating the implementation of our approach into our developed tool CAUT for test case generation. The main part of CAUT is implemented in C language and we use CIL [12] to formalize the source code of the program under test. To solve path constraints, we use lp_solve [13], which is an open source library for linear programming. We compared CAUT with CUTE [10], which is another test data generation tool based on dynamic symbolic execution as well. We have performed the experiments on a Linux machine with a Intel(R) Core(TM)2 Duo CPU @ 2.00GHz using GCC 4.2.3 with 2GB allocated memory.

Table 3. Comparison of CAUT with CUTE

| Program | CAUT | | | CUTE | | |
|---|---|---|---|---|---|---|
| | Error Found | Iters | Branch Coverage | Error Found | Iters | Branch Coverage |
| ERT | Y | 2 | 100% | Y | 2 | 100% |
| HSD | Y | 3 | 50% | Y | 3 | 50% |
| LDT-P10 | Y | 4 | 66.7% | Y | 2 | 33.3% |
| PTA-P4 | Y | 4 | 100% | N | - | - |
| PTA-P6 | Y | 4 | 100% | Y | 2 | 50% |
| PTA-P7 | Y | 4 | 100% | N | 2 | 50% |
| SAM-P57 | N | 1 | 33.3% | Y | 3 | 66.7% |
| SAM-P58 | Y | 1 | 50% | Y | 1 | 50% |
| SAM-P60 | N | 1 | 33.3% | N | 3 | 66.7% |

Table 3 lists nine C programs that has been used in the experiments, and most of them were experimented by other researchers [14]. In the first two case, both CUTE and CAUT can detect errors. In the case three to the case six CUTE either fails to find error or obtains a lower branch coverage than CAUT. It implies that our approach is able to generate efficient test data to uncover potential errors and ensure a reliable branch coverage. In the last three cases, CUTE beats CAUT on the contrary. The reason is that there are operations on strings in the program under test while CAUT does not support such operations well at present.

We have also applied our tool in some real programs and the experimental results are shown in Table 4. The first example is an FFT algorithm. The second program tests the data structure of a binary search tree. The third example is the quicksort algorithm and the last program is an algorithm

Table 4. Comparison of CAUT with CUTE

| Program | CAUT | | CUTE | |
|---|---|---|---|---|
| | Iters | Branch Coverage | Iters | Branch Coverage |
| fft | 10000 | 87.5% | 10000 | 87.5% |
| binary search tree | 32 | 100% | 984 | 100% |
| quicksort | 120 | 100% | 120 | 100% |
| inverse matrix | 37 | 66.7% | 4 | 24% |

to inverse the given matrix. In the first three cases, CAUT and CUTE reach the same branch coverage, however, our tool generates less test data than CUTE. CAUT performs much better than CUTE in the fourth case, because there are many variable array index in the program under test. It shows the our tool is more capable to handle arrays than CUTE.

In many cases, the branch coverage fails to reach 100% for both CAUT and CUTE. There are three main reasons. First, some branch conditional expressions contain non-linear mathematical operations, so that we cannot solve the corresponding path constraints, because most constraint solvers used by us are for linear constraint solving. Second, the programs under test contain some assertions that are never violated. Third, some branches are affected by library function call. Without source codes of these library function, the symbolic execution based approach cannot generate precise test data to be directed to the following branches.

## 5. Related Work

So far, automating unit testing is still a dream for most software vendors. One big obstacle is how to generate appropriate test input. Many methodologies and prototypes have been designed in the past decades. The followings are some examples.

Static Symbolic execution [7] is a traditional approach for test case generation. Unfortunately, due to the limitation of theorem prover, the constraint containing statements such as function call, pointer access, are unable to be solved. In other words, the exclusive use of symbolic execution cannot process pointer or complex data structures. For example, the tool Symstra [15] cannot handle such codes that contain array indexing with variables.

Dynamic test case generation [9] is a more practicable approach. It can be treated as a combined technique to some extent by applying static analysis, symbolic execution and constraint solving. It executes the target program repeatedly, usually starts with random inputs and then collects symbolic constraints along the real executing path. By using a constraint solver, these collected constraints are solved to infer the alternative set of inputs which directs the next execution path.

DART [16], abbreviation of Directed Automated Random Testing, proposed by Patrice Godefroid, et al., is a typical

tool using dynamic test-case generation technique. The goal of DART is systematically executing all feasible program paths to detect latent runtime errors. It adopts an improved random testing technique to achieve better coverage. Systematic Modular Automated Random Testing (SMART) [17] is an extension to DART. SMART extends DART by testing functions in isolation, encoding test results as function summaries expressed using input preconditions and output post-conditions, and then re-using those summaries when testing higher-level functions.

CUTE [5] is another tool on dynamic test-case generation. It mixes dynamic concrete and symbolic execution, *concolic execution*, which is good for dealing with pointers and complex data types and has some optimized constraint solving algorithms. It supports bounded depth-first search to avoid search space explosion. Hybrid Concolic Testing [18] extends the CUTE work, where random search and bounded depth-first search are combined. The branch coverage by using this method has notable boost.

## 6. Conclusion and Future Work

Testing is an important phase in software life cycle, and test data generation is really a challenge in real life programs in industry. An approach to generate test data for complex data structures of C program is proposed in this paper, and we also implement our idea by integrating it into the tool CAUT. This tool can generate test data for pointer, array and structure variables. Experiments show that our tool is effective in handling some non-trivial programs. We plan to improve the tool performance for constraints involving variable as array index and extend the tool further to support more mechanisms of C language, such as bit operation, function pointer. By collaborating the efficient path selection strategy proposed in our previous work, we believe that this tool will be scalable to large programs of real world.

However, full support of accessing variable as array index is still needed to be explored in the future work, because the selection of a large set of index combination is a heavy burden. We will consider to introduce heuristic methods based on data flow analysis to select constraints from transformed conjunctive constraints instead of the random selection method currently used.

## Acknowledgment

Geguang Pu is partially supported by 973 Project No. 2005CB321904, 863 Project No.2007AA 010302, NFSC No. 90818024 and Qimingxing Project No. 07QA14020.

Zuohua Ding is supported by NFSC No.90818013.

## References

[1] "Junit," 2003, http://www.junit.org.

[2] "Micorsoft visual studio developer center," 2004, http://msdn.micorsoft.com/vstudio.

[3] B. Beizer, *Software Testing Techniques*. New York, NY, USA: John Wiley & Sons, Inc., 1990. [Online]. Available: http://portal.acm.org/citation.cfm?id=533100

[4] G. Pu, Z. Wang, X. Yu, T. Sun, and J. He, "Caut: Automated test case generation with program analysis and scalable path search," http://caut.lab205.org/publication/publication.htm.

[5] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 263–272.

[6] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM Syst. J.*, vol. 22, no. 3, pp. 229–245, 1983.

[7] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[8] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, 1991.

[9] B. Korel, "A dynamic approach of test data generation," *Software Maintenance*, pp. 311–317, November 1990.

[10] K. Sen, "Cute: A concolic unit testing engine for c and java," http://osl.cs.uiuc.edu/ ksen/cute/.

[11] Z. Wang, G. Pu, X. Yu, T. Sun, and J. He, "Scalable path search for automated test case generation," http://caut.lab205.org/publication/publication.htm.

[12] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *In International Conference on Compiler Construction*, 2002, pp. 213–228.

[13] "lp solve," http://lpsolve.sourceforge.net/.

[14] "Software-artifact infrastructure repository," http://sir.unl.edu/portal/index.html.

[15] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *TACAS*. Springer, 2005, pp. 365–381.

[16] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[17] P. Godefroid, "Compositional dynamic test generation," in *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2007, pp. 47–54.

[18] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 416–426.