A Test Data Generation Tool for Unit Testing of C Programs [*]

Zhongxing Xu
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Graduate University
Chinese Academy of Sciences
xzx@ios.ac.cn

Jian Zhang
State Key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
zj@ios.ac.cn

Abstract

This paper describes a prototype tool, called SimC, which automatically generates test data for unit testing of C programs. The tool symbolically simulates the execution of the given program. It simulates pointer operations precisely. This makes it capable of generating test data for programs involving pointer and structure operations. Experiments on real-world programs including the GNU coreutils are presented. Some trade-offs of simulation schemes are also discussed.

Keywords: Test data generation, symbolic execution, pointer operation, unit testing.

1 Introduction

Testing is a common activity in software development, and it is crucial to software quality. However, for the time being, few tools are available that support testing adequately. In particular, automatic test data generation is desirable but hard to achieve.

In this paper, we describe an approach to test data generation for unit testing of C programs, which is highly automatic and efficient. It generates program paths and symbolically executes the paths to collect path condition restricting the input data that can drive the program to execute the path. Then the path condition is solved by a constraint solver to get the input data. In the most general cases, solving the path condition is an undecidable problem. But under reasonable assumptions (such as excluding nonlinear integer arithmetic), the path condition can be solved.

Symbolic execution was proposed in 1970's [11] [2]. But some fundamental difficulties limit the application of symbolic execution to real-world programs. For example, variable array indexing and symbolic pointer dereferencing are troublesome in symbolic execution.

Our approach addresses these difficulties, and it has the following features:

1. Simulating memory with auxiliary arrays. The address space of a process can be seen as a large array. Ideally we can represent all the pointer operations by array references and index operations.

2. Differentiating between symbolic and concrete pointers. A symbolic pointer is treated as an unknown variable, while a concrete pointer represents a memory region and is replaced by its simulated address in the path, see Section 3.2.

3. Solving hybrid constraints which involve both boolean variables and numerical variables.

We implemented a tool called SimC which accepts programs written in a significant subset of ANSI C. Many experiments are done on code from the research literature and GNU utilities (such as `ls`, `make`). The results are encouraging. Test data are generated that cover all the branches of the flow graph, for rather complex functions.

In the next section, we recall some basic concepts and previous work. Then in Section 3, we describe our test data generation method in detail. We report some experience on applying our prototype tool to GNU programs in Section 4. Limitations of the method are discussed in Section 5. We review related work in Section 6 and conclude in Section 7.

2 Path Feasibility Analysis

A program can be described by a flow graph which has many paths. A path analyzer called PAT is described in [18].

[*] Supported in part by the National Science Foundation of China (under grant No. 60125207 and 60421001).

Proceedings of the Sixth International Conference on Quality Software (QSIC'06)
0-7695-2718-3/06 $20.00 © 2006 IEEE

PAT accepts a path as input, which is a sequence of assignment statements and conditional expressions, preceded by a set of variable declarations. The variables handled by PAT include integer variables, real variables, Boolean variables and arrays.

PAT can decide the feasibility of a program path under reasonable assumptions. The tool tries to find input data such that the program will be executed along a given path. If successful, the input data (mainly the initial values of the variables) will be shown to the user, and the path is said to be feasible. Otherwise, the path is said to be infeasible.

Given a path, PAT determines if it is feasible by first symbolically executing it to get a set of constraints (i.e. the path condition) and then solving the path condition.

In this paper we use EPAT, which is an extended version of PAT [18].

3 Automated Test Data Generation

3.1 Overview

This section gives an overview of SimC. The test data generation process consists of the following steps:

1. Parsing the C code into an intermediate representation and building up control flow graphs (CFG).
2. Generating program paths and translating them into a form that can be processed by our path solver EPAT.
3. Symbolically executing each path and solving the path condition to decide its feasibility.
4. Selecting a group of feasible paths according to certain path coverage criteria.

In the end, a set of test data that can drive the program to execute different paths is generated.

SimC generates paths for the user specified function. For simplicity, we assume that each function has only one exit. Each path starts from the entry point of the function, and ends at the exit of the function. A path consists of two kinds of statements: assignments and conditional expressions. An assignment assigns a storage place or constant to a storage place. A conditional expression is a mathematical constraint on storage places which must be satisfied at the execution point of the path. Storage places are represented by variables or array references.

We use arrays to simulate the underlying memory. All storage places in the program, including scalar variables, structures, arrays, and dynamically allocated memory, are mapped onto the array that models the memory. We record the simulated addresses of variables and pointed-to addresses of pointers in the symbol table. With this simulation, all references to storage places can be represented by array references. All pointer operations can be converted to array index operations. With the help of a powerful constraint solver, we can solve the generated path condition and get input data for the program.

```
void f(char *s)
{
  char *p;
  p = s;
  while (*p != ’a’)
    p++;
}
```

Figure 1. A simple example

Let us look at a simple example in Figure 1. One of the program paths can be translated into:

```
p = 1;
@ mem[p] != ’a’;
p++;
@ mem[p] == ’a’;
```

In this example, statement `p = s` is translated into `p = 1`, where 1 is the simulated starting address of array `s`. The reason that the simulated address starts from 1 instead of 0 will be discussed in Section 3.2.1. Then pointer dereference `*p` is translated into array reference `mem[p]`. Pointer increment `p++` is translated into array index incrementation `p++`. The heading `@` indicates a conditional expression. This path is then checked with EPAT. We get `s[1] = ’b’` and `s[2] = ’a’` which satisfies the path condition.

In this section, we focus on path generation, which is a translation process. SimC tries to translate various C program constructs into a simpler path form that is acceptable by EPAT. Below we discuss how we handle various problems in the translation process.

3.2 Arrays and Pointers

The C programming language allows the programmer to manipulate pointers almost arbitrarily. To simulate pointer operations precisely, we use arrays to simulate the underlying memory. If a variable `p` is allocated on simulated memory `mem` with simulated address i, then it can be represented by `mem[i]` in the path.

Pointer operations can be roughly classified into two classes: pointer arithmetic and pointer dereference. Pointer arithmetic can be simulated by array index arithmetic. Pointer dereferences are translated into array references.

3.2.1 Simulation Schemes

There are two design choices we currently face. First we have to decide whether to use a single array to allocate all the variables or use multiple arrays for different objects. It is essentially a matter of renaming.

If we use a single array, then there will be no different variable names in the path. Every variable in the path is replaced by an array reference with its simulated address as index. If we use multiple arrays, each memory object (variable, array, or structure) is allocated on simulated memory with different names.

Using a single large array simplifies path representation. Each variable, including a pointer variable, is mapped onto the large array and has its own simulated address. The pointed-to address of a pointer is saved in its associated array unit. With this scheme, all the variables are replaced by their corresponding array references in the path.

The disadvantage of using a single array is that it increases the computation cost of solving the path for input data. An unknown pointer can point to an arbitrary address. When we solve the path condition, we have to try each possible value for the unknown pointers. Large array increases the search space, although this can be addressed by adding constraints for each array reference with unknown index.

On the other hand, we can use separate arrays for different objects in the program. We call such an array a memory region. We have to record in the symbol table both the point-to address of a pointer and which memory region it points to. Some programs that involve list and tree operations can not be simulated in this way. For example, in code

```
struct node {
  int i;
  struct node *next;
}
struct node *p;
p = malloc(sizeof(struct node));
```

we can represent the i field of the malloced structure by p[0]. But we can not represent the object pointed by p->next, because we have no place in the symbol table to record the name of the memory region p->next points to.

Second we have to decide whether to allocate pointer variables themselves on the simulated memory. If we do not allocate simulated memory unit for a pointer, we must record the simulated address it points to in its meta-data in the symbol table. If a pointer is allocated a simulated memory unit, its point-to address can be recorded in the simulated memory unit.

We think the most powerful scheme is to use a single array to simulate the entire memory and allocate pointers on the simulated memory. The power of this scheme is illustrated by the example in Figure 2.

```
void f(void)
{
    int a, *p, **q;
    a = 1000;
    p = &a;
    q = &p;
    **q = 1000;
    p--;
}
Translated into path:
  mem[1] = 1000;
  mem[2] = 1;
  mem[3] = 2;
  mem[mem[mem[3]]] = 1000;
  mem[2] = mem[2] - 1;
```

Figure 2. Using a single array and allocating pointers on the simulated memory can simulate complex pointer operations.

Suppose we use array mem to simulate the memory. The simulated addresses of variables a, p, q are 1, 2, 3 respectively. Simulated address 0 is reserved for the NULL pointer. The program can be translated into the path at the bottom of Figure 2. Everything is translated naturally.

Simulated address starts from 1 instead of 0. We explain the reason below. Pointers are often compared with NULL. They are replaced by their point-to addresses in the generated path. If a simulated address starts from 0, then 0 would be a legal address that can be pointed to. This is inconsistent with the real machine. Thus simulated address 0 is reserved to represent a NULL pointer.

3.2.2 Symbolic and Concrete Pointers

To simplify exposition, from now on we use the term *the address of a pointer* to refer to the address that the pointer points to.

SimC is designed as a unit testing tool. When doing unit testing, we usually do not have the full runtime information of the program. This lack of information causes no difficulty for treating local pointers, since they are always assigned values before being used (otherwise it is a potential bug). But pointers as function parameters are hard to treat uniformly.

We observed that there are three cases when a pointer is passed as a function parameter.

1. The pointer is a concrete pointer, which means that it points to the beginning of an existing memory region,

```
void foo(char *s)
{
        int i;
        for (i = 0; i < 10; i++)
                s[i] = i;
}
```

Figure 3. Function parameter pointer p is used as a concrete pointer.

```
void mem_free(char *buf, char *p)
{
        ...
        position = (p - buf) / 16;
        ...
}
```

Figure 4. Function parameter pointer p is used as a symbolic pointer.

such as a malloced buffer. This is the most prevalent case. Most function parameter pointers are used to pass a memory region from the caller to the callee, such as the example in Figure 3.

In such cases, we should assign a simulated address to the parameter pointer and let it point to a chunk of memory. The value of a concrete pointer is not changed in the called function.

2. The pointer is a symbolic pointer. The symbolic pointers do not point to the beginning of a memory region. They represent some arbitrary addresses. For our purpose of test data generation, the addresses they point to should be solved according to different program paths.

   For example, in the code in Figure 4, `buf` is a concrete pointer which points to a memory region. We should allocate a chunk of memory and assign the beginning address to `buf`. But `p` does not point to another chunk of memory. We infer from the code that `p` should point to somewhere in the memory region pointed by `buf`. Its value should be solved according to different paths.

3. The pointer is both a concrete pointer and a symbolic pointer. This is possible. Some pointers are first used to pass a memory region to the callee, then used as a symbolic pointer. The value of such pointers may be changed during path execution.

   Let us look at the code in Figure 5. `s` is first used to pass a memory region to function `bar`. Then it is assigned a new address.

```
void bar(char *s)
{
        ...
        putchar(s[0]);
        ...
        s = strchr(s, ’a’);
        ...
}
```

Figure 5. Function parameter pointer that is used both as a concrete and as a symbolic pointer

In general, there is no easy way for a testing tool to know whether a pointer is concrete or symbolic. We let the user specify whether the pointer is concrete or symbolic when it appears as a function parameter. In addition, the user may specify the size of the memory region which the concrete pointer points to.

A concrete pointer represents a memory region. Its simulated address is known during the execution of the program path. It is replaced by its simulated address when it appears in the path.

A symbolic pointer represents a simulated address. Its address is unknown. We must solve it for test data generation. It appears in the path literally as an unknown variable. So the statement in Figure 4 is translated into:

```
position = (p - 1) / 16;
```

Note that `buf` is replaced by its simulated address 1.

Operations on symbolic pointers appear in the path as usual variable operations. Because their addresses are unknown, we can not operate on their meta-data in the symbol table directly (as for concrete pointers). Take statement `p++` for example, if `p` is a concrete pointer, we increase its point-to address field in the symbol table. If `p` is a symbolic pointer, we generate `p = p+1` in the path. Dereferencing them introduces array references with unknown indices. Our path solver EPAT can check such paths.

3.3 Function Calls

There are two common approaches to dealing with function calls in programs. One is inlining the called function. The other is modeling the function.

Inlining the called function is straightforward. We make the argument passing process explicit as assignments in the generated path. Then the called function is expanded. The value return process is translated into assignments again.

To model a function we use several conditional expressions to describe the behavior of the called function. For example, when we encounter an input function, we often don't care about its internal mechanism, say how it interacts with the disk or network. We only need to know that it returns an `int` value. It is suitable to describe the behavior of the function by a conditional expression characterizing its return value. Furthermore, some library functions' source code is not available and can not be executed symbolically.

In SimC, we use both methods to handle function calls. We model library functions in a simple constraint form. For example, for function `c = getchar();`, we translate it into:

```
c = INPUT;
@ 0 <= c <= 255;
```

As another example, function call `p=strchr(s,c);` can be modeled as :

```
(p==NULL) ||
(*p==c && s<=p && p<=s+lengthOf(s))
```

These function models work well in our experiments.

3.4 Path Generation Strategy

The strategy for generating paths from CFG is crucial to the effectiveness of our method for generating test data.

Depth-first search of CFG is not effective when the CFG has nested loops. Some loops are executed many times while others are never executed.

Currently we use breadth-first search with a maximum path length limit to generate paths. The CFG of the program is processed with the standard BFS method. Additionally, we compute the partial path length for each node before it is entered into the queue. If the partial path length exceeds the specified maximum path length, the node is discarded. When the exit node of the CFG leaves the queue, the complete path is recorded.

Even if the path length is limited, we may still generate hundreds of feasible paths for some functions (see Section 4). We use an optimal path-selection mothod described in [15]. The main idea is to take edge covering of the CFG as path selection criteria. To make sure every edge of the CFG be covered at least once in some path, the problem can be formulated as a zero-one integer programming problem.

We developed a separate tool to select paths. We use `lp_solve` [12] as the integer linear programming back end. It works well in our experiments: usually fewer than ten paths can fulfill the edge covering.

4 Experiments

Many previous test generation papers experimented on small, custom programs written in restricted languages [6] [13]. We experimented our method on programs including `coreutils` and `make`. The code fragments from `coretuils` are listed in Table 1.

Function	File
remove_suffix()	basename.c
cat()	cat.c
cut_bytes()	cut.c
parse_line()	dircolors.c
set_prefix()	fmt.c
attach()	ls.c
bsd_split_3()	md5sum.c
hex_digit()	md5sum.c
isint()	test.c
make_printable_char()	tr.c
strtol()	strtol.c

Table 1. Code experimented on from GNU coreutils 5.2.1 package

The criteria for selecting functions are: (1) that the function should have intensive pointer operations and several loop and branch statements; (2) that the function should not call other non-library functions. The second restriction is due to our tool's inability to generate interprocedure paths. We will extend the tool in the future. Detailed experimentation is described in the following subsections.

Since our current implementation can only accept ANSI C code, the programs were slightly modified to eliminate GCC extensions before being processed by SimC. Some macros were expanded by hand to keep things simple. But none of the essential functionalities were altered.

Experiments were done on a PC with a Pentium 4 3.4GHz CPU and 1G memory. The computation time of each experiment is less than two minutes.

4.1 Example: remove_suffix() in GNU coreutils

This is a typical piece of code that SimC is good at generating test data for. The code has extensive pointer arithmetic and dereference operations. Pointer aliases also exist. SimC maps the storage of the program onto the simulated memory. All program operations are done naturally on the simulated memory.

```
void remove_suffix(char *name,
             const char *suffix)
{ ...
  np = name + strlen(name);
  sp = suffix + strlen(suffix);
  while (np > name && sp > suffix)
    if (*--np != *--sp) return;
  if (np > name) *np = ’\0’;
}
```

The code is part of the implementation of the `basename` command. Although it is small, it is sufficiently complex to frustrate all of the test generation tools that we know. SimC generates 5 sets of test data covering all branches of the function.

4.2 Example: strtol() in GNU coreutils

This section presents a more complex example from GNU coreutils package. The code related to pointer operations is listed below.

```
int strtol(const STRING_TYPE *nptr,
    STRING_TYPE **endptr,
    int base,
    int group LOCALE_PARAM_PROTO)
{
  ...
  save = s = nptr;
  while (ISSPACE (*s))
    ++s;
  if (*s == L_(’-’)) {
      negative = 1;
      ++s;
    }
  ...
}
```

We limited the path length to 20, and 10 sets of test data are generated.

4.3 Example: getop()

This example is from K & R’s C programming language book [10] and it is used by Bertolino and Marré [1] as an example.

This program has complex control structures, and is difficult to test manually. We limited the path length to 20. SimC generates 178 sets of test data from 297 paths. After applying the optimal path selection method described in Section 3.4, we got 4 paths that can cover all of the edges in the CFG of the function. In [1], the authors described how to generate a set of paths which are *likely* to be feasible. They generated 7 paths and have no automatic tool support.

4.4 Example: InsertionSort()

This is a common implementation of the insertion sort algorithm.

```
void sort(int *a, int n)
{
  int cur, j, low_ind, temp;
  for (cur = 0; cur < n-1; cur++) {
    low_ind = cur;
    for (j = cur + 1; j < n; j++) {
      if (a[j] < a[low_ind])
        low_ind = j;
    }
  ...
  return;
}
```

In this example, there are nested loops. Generating paths which need different input data are not trivial. Depth-first search performs poorly in this case.

We limited the maximum path length to 20. EPAT generated 9 sets of test data: {0}, {0,1} {1,0} {0,1,2} {1,2,0} {0,2,1} {1,0,2} {2,1,0} {2,0,1}. The program is executed through different paths for these inputs.

4.5 Example: dosify() in GNU make 3.80

This section applies SimC to the `dosify()` function of GNU’s `make 3.80`.

```
static char *
dosify (char *filename)
{
  ...
  df = dos_filename;
  ...
  if (*filename != ’\0’) {
      *df++ = *filename++;
      for (i = 0; *filename != ’\0’ &&
            i < 3 && *filename != ’.’; ++i)
      *df++ = ...;
    }
  while (*filename != ’\0’ &&
        *filename != ’.’)
    ++filename;
}
```

This example has intensive pointer operations. Parameter `filename` is passed in as a pointer to a character array. It is compared to 0 and dereferenced. We let it be a concrete pointer.

Among the 420 paths whose lengths were less than 20, EPAT found 50 feasible paths, and generated 50 sets of test data in less than one minute. After applying the optimal path selection model, 3 paths were selected covering all of the edges of the CFG.

5 Discussion

We made the following observations after experimenting our method.

1. Not all of the functions need automatic testing. Many functions have simple control structures and data structures. Their correctness is easy to ensure. Only a few 'core' functions need to be tested thoroughly.

2. Our method can generate test data for functions with complex control structures and pointer arithmetic. The test data is effective with respect to path coverage metrics.

3. Many library function calls are easy to model. And some can be ignored safely.

4. Some library functions are hard to model precisely, especially functions that deal with strings. We are improving our modeling system for them.

Although the above simulation scheme is effective for a large number of programs with pointers, there are a few limitations of the scheme.

1. Function pointers can not be simulated under current simulation scheme. There is no way to symbolically execute a function call when we do not know which function to execute.

2. The current scheme can not generate test data for programs with structurally complex inputs, such as linked lists and binary trees.

3. The path generation strategy must be refined when dealing with recursive function calls.

4. Our current path analyzer EPAT can not handle nonlinear and bitwise path condition.

6 Related Works

The initial idea of this paper appeared in [18]. An extension to a custom language was described in [19]. Structures and pointers are treated in a following paper [17]. This work extends the method in these papers significantly. First, pointers and structures are treated more thoroughly. Second, path generation strategy is considered. Third, a complete prototype tool is implemented. Last and the most important, the method was experimented on real-world programs.

6.1 Symbolic Execution

Symbolic execution was proposed in 1970's [2, 11]. The basic idea is generating logical condition for each program path, and solving this condition to provide input data which will drive the program through the path. But few systems implement the idea fully. And even fewer ones can process programs with the complexity as ours. For example, Denaro *et al.* [5] describe a system which can deal with a small subset of C.

Recently there are some renascent research on program testing and bug finding based on symbolic execution [8] [14]. These works are all dynamic testing.

The DART project [8] and CUTE project [14] combined concrete and symbolic execution. DART runs the unit being tested on a concrete random input. It symbolically collects path constraints during the execution. Then the last constraint is negated to generate the next test input. DART only handles constraints on integers and resolves to random testing when symbolic pointers are encountered.

CUTE extends DART to handle simple symbolic pointer constraints of the form: `p == NULL`, `p != NULL`. But it fails to deal with symbolic array index expressions. In contrast to both DART and CUTE, SimC models symbolic pointers and array indexes precisely, and thus it can generate more accurate test data.

CBMC is a bounded model checker for ANSI-C programs [3]. It is designed to check program properties including pointer safety, array bounds, and user-provided assertions. Like SimC, CBMC runs code entirely symbolically. It translates the program to boolean formula that is satisfiable if there exists an error trace. The formula is then checked by using a SAT procedure. CBMC unwinds all loops and recursive calls. This sometimes makes the checking procedure run forever unwinding loops. Also the lack of support for library functions makes it hard to apply CBMC to real-world code.

Xie *et al.* [16] described how to generate test data for object-oriented programs using symbolic execution. But array expressions are not considered.

6.2 Test Generation

Previous test generation work mostly experimented on small, custom programs written in a subset of real programming languages [13, 6]. The programs do not involve pointers and structures. Programming languages like C are much more complex and versatile. This makes test data generation more difficult. Our method simulates the language operations precisely and can generate test data for more complex programs.

Gotlieb et al. [9] developed an automatic test data generation technique based on SSA and constraint solving. Their system accepts a subset of the C language, which does not include pointers and dynamically allocated structures. The largest program they experimented on was less than 20 lines of code. In contrast, our system can process programs with pointers and dynamically allocated structures.

An attempt at using symbolic execution for test data generation for fault based criteria is described in [4]. In this work, a test data generation system based on a collection of heuristics for solving a system of constraints is developed. The constraint solving method is incomplete, resulting in an approximate solution on which the path may not be traversed. Again, EPAT can guarantee the selected path is traversed on the generated test data.

An approach to automatic generation of test data for a given path using the actual execution of the program is presented in [6]. Another program execution based approach that uses program instrumentation for test data generation for a given path has been reported in [7]. These approaches consider only one branch predicate and one input variable at a time and use backtracking. Therefore, they may require a large number of iterations even if all the branch conditionals along the path are linear. If several conditionals on the selected path depend on common input variables, a lot of effort can be wasted in backtracking. They cannot consider all the branch predicates on the path simultaneously because the path may not be traversed on an intermediate input.

7 Conclusion

Testing is an important step in software development, and test data generation is quite challenging for non-trivial programs. In this paper, we have presented an automatic test data generation tool SimC. It uses symbolic execution and constraint solving techniques. A new method to simulate pointer operations precisely is proposed.

Experimental results show that our tool is effective in handling real-world programs. It can generate test data for programs involving complex pointer operations. This should be very helpful to programmers and testers.

Although we did not experiment our system on large benchmarks, we believe that a good implementation and increasing computational power can make the techniques scalable to large programs.

In the future, we plan to extend our tool further, so that more code fragments can be handled. For example, we intend to study bit operations. We are also going to write error checkers which can check for some common bugs (such as array index out of bound errors) automatically.

References

[1] A. Bertolino and M. Marré. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, 1994.

[2] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT — a formal system for testing and debugging programs by symbolic execution. In *Proc. of the Int'l Conf. on Reliable Software*, pages 234–245, 1975.

[3] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the ASP-DAC 2003*, pages 308–311, 2003.

[4] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[5] G. Denaro et al. A symbolic execution based approach for verifying safety critical software. 2004.

[6] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. on Software Engineering and Methodology*, 5(1):63–86, 1996.

[7] M. J. Gallagher and V. L. Narasimhan. ADTEST: A test data generation suite for ada software systems. *IEEE Trans. on Software Engineering*, 23(8), 1997.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[9] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–62, 1998.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall PTR, 1978.

[11] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[12] lp_solve. http://groups.yahoo.com/group/lp_solve/.

[13] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9:263–282, 1999.

[14] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.

[15] H. S. Wang, S. R. Hsu, and J. C. Lin. A generalized optimal path-selection model for structural program testing. *Journal of Systems and Software*, 10:55–63, 1989.

[16] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. TACAS 2005*, LNCS 3440, pages 365–380.

[17] J. Zhang. Symbolic execution of program paths involving pointers and structure variables. In *Proceedings of the Fourth International Conference on Quality Software*, pages 87–92, 2004.

[18] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.

[19] J. Zhang, C. Xu, and X. Wang. Path-oriented test data generation using symbolic execution and constraint solving techniques. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, 2004.