http://dx.doi.org/10.7236/JIWIT.2012.12.1.123
JIWIT 2012-1-16

목적 지향 콘콜릭 테스팅을 이용한 플래그 변수가 있는 프로그램에 대한 테스트 데이터 생성

Generating Test Data for Programs with Flag Variables using Goal-oriented Concolic Testing

정인상

Insang Chung

요 약 이 논문은 콘콜릭 테스팅을 특화한 목적 지향 테스트 데이터 생성 방법을 제안한다. GCT(Goal-oriented Concolic Testing)라 불리는 제안된한 특정 목표를 실행하는 테스트 입력을 생성한다. 콘콜릭 테스팅은 요구되는 테스트 입력을 발견할 때 까지 모든 가능한 경로 공간을 탐색하는 브루트 포스 방식으로 간주할 수 있다. 대조적으로 GCT는 자료 흐름 정보를 활용하여 목표가 실행되기 위해 먼저 수행되어야 하는 문장들을 식별하여 탐색되는 프로그램 경로의 수를 제한한다. 플래그 변수가 있는 실험을 통해 GCT의 효과성을 보인다.

Abstract This paper presents a goal-oriented test data generation technique that specializes concolic testing. The proposed technique, referred to as GCT (Goal-oriented Concolic Testing) produces test inputs which execute a specific target. Concolic testing can be seen as the brute force approach to search the space of all possible paths until a required test input is found. In contrast, GCT restricts the number of program paths that are explored by using data flow information to identify statements that should be executed beforehand in order for the target to be executed. We conducted experiments to evaluate the performance of GCT with programs with flag variables to show its effectiveness.

Key Words : 목적 지향 테스팅, 콘콜릭 테스팅, 플래그 문제

Ⅰ. Introduction

Concolic testin[1,2] is receiving attention because of its ability to explore the path space exhaustively to achieve full path coverage. It combines symbolic and concrete execution to generate test data to explore all feasible execution paths. A concrete execution is performed on a program with random inputs and the path constraints are then collected along the executed path. These constraints are systematically negated to generate new test inputs that drive the program along alternative paths.

However, concolic testing can be ineffective when testing a particular targeted section of a program. For example, consider one key regression testing activity, referred to as test suite augmentation to test whether

*정회원, 한성대학교 컴퓨터공학과
접수일자 2011.11.7, 수정완료 2011.12.20.
게재확정일자 2012.2.10

Received: 7 November 2011 / Revised: 20 December 2011 / Accepted: 10 February 2012
*Corresponding Author: insang@hansung.ac.kr
Dept. of Computer Engineering, Hansung University, Seoul, Korea

or not modifications are made as intended[3]. Test suite augmentation refers to the process that creates additional test data required to test new or modified program behaviors that are not addressed by existing test suites. It is efficient to generate test data that exercise only changed or affected portions of a program rather than all program components.

Static analysis techniques can be used in conjunction with program testing to determine if warnings reported by static analysis toolsreally exist in the actual program. If program testing is performed on only program locations relevant to reported errors, then the total time to detect errors can be dramatically decreased[4].

This paper presents a goal-oriented approach for automated test data generation based on concolic testing, referred to as GCT (Goal-oriented Concolic Testing) that generates test inputs which execute a specific target. From the viewpoint of goal-oriented testing, concolic testing can be seen as the brute force approach to search the space of all possible feasible paths until a required test input is found. Even though searching the path space exhaustively enables all statements vital to reaching the target to be executed, the execution of statements not required to reach the target can be performed needlessly. This can make testing intractable when the search space is large. This paper revises the GCT procedure in[5] and adds some experimental results.

GCT is applied to programs with flag variables to show its effectiveness for test data generation. Most of the goal-oriented techniques rely on a distance function to discriminate between candidate test inputs in terms of the cost required to achieve the test goal. However, the distance function becomes a near constant function that returns a constant value for a wide range of inputs when Boolean-valued (flag) variables are used in branch predicates. Thus, it is very difficult to direct the search of the required test data for those programs with flag variables[6][7].

Many automated test data generation techniques involve solving a system of equations to generate test data. Solving a system of equations is in general undecidable. As a consequence, most of test data generation techniques place a certain termination condition including a maximum number of iterations in order not to end up in an infinite loop. In this paper, we show that the GCT procedure performs only a few iterations even when infeasible targets are encountered. This is of great benefit to saving the cost of testing because it is impossible to know whether a given target is infeasible in advance.

The paper isorganized as follows. Section 2 gives a brief description about concolic testing. Section 3 gives the GCT procedure with some basic definitions. Section 4 gives experimental results to show that the GCT approach is promising for a certain class of programs and is also efficient even when encountering infeasible targets. Section 5 gives related works and Section 6 gives the conclusion and future works.

II. Concolic Testing

Concolic testing is a dynamic symbolic execution where the program under test P is executed both on concrete and symbolic values, generating symbolic path constraints Φ_π along the path π. In certain cases, Φ_π is simplified using the corresponding concrete values. In addition to Φ_π, concolic testing maps program variables to symbolic expressions using the symbolic memory map S. Initially each input parameter x_i is assigned a fresh symbolic value δi. That is, $S[x_i]=\delta i$. The mapping of a program variable, say t, is updated to the symbolic expression (e) at every assignment statement of the form 't=e' where by evaluating e in the current symbolic memory map.

Φ_π is initially assigned the Boolean value True. Φ_π is updated whenever a conditional statement of the form 'if c then … else …' is encountered during the symbolic execution along π using the symbolic memory map. Of course, if the concrete execution takes the then branch,

the symbolic expression ϐ(e) is conjoined with the current symbolic path constraint Φ_π as follows: '$\Phi_\pi \leftarrow \Phi_\pi \wedge$ ϐ(c)'. Similarly, if the else branch is taken, Φ_π is changed to '$\Phi_\pi \wedge$not ϐ(c)'.At the end of the concolic execution, the last constituent of Φ_π is selected and negated to produce a new symbolic path constraint Φ_π° when the depth first search strategy is used. A new test input satisfying Φ_π° directs the program along a different execution path. This process is repeated until all feasible execution paths have been explored. Indeed, concolic testing explores program state space exhaustively, and may even require great effort when a test input is only required to reach a particular program point. The situation can get worse if there are a huge number of program paths to be explored before reaching the target point.

Ⅲ. Goal-oriented concolic testing

1. Preliminary

An *execution path* is a sequence $\langle s_1, s_2, \cdots, s_n \rangle$ of program statements executed on a certain input. A *branch* is defined as a pair consisting of a conditional statement and a statement immediately following the conditional statement. For each branch b_i, *paired*(b_i) denotes the paired branch, i.e., the alternative branch of b_i . For each conditional statement c_k, b_i is executed when c_k evaluates to true (or false) iff *paired*(b_i) is executed when c_k evaluates to false (or true). Executions of a program are represented using a *computational tree*. Each node of the computational tree corresponds to an execution of a conditional statement while each edge corresponds to a sequence of non-conditional statements executed between two successive conditional statements. Fig. 2 shows the partial computational trees of the program in Fig. 1, which is adapted from Ferguson and Korel[8]. We call a branch b_i a *guiding branch* with respect to an execution path and a target point t if for b_i, b_i exists on and there exists a path leading to t through *paired*(b_i). In other words, a guiding branch is a candidate branch leading to a target by negating it. $GB_t(\pi)$ denotes the set of the guiding branches for the execution path . For example, consider the program path of Fig. 2(a) that is indicated by the shadowed area and let the path be π. Then, $GB_{target}(\pi)$ is the set of the branches executed when each predicate of the set {a[i]!=5 for 0≤i≤9, fa!=1, fb!=1} evaluates to true.

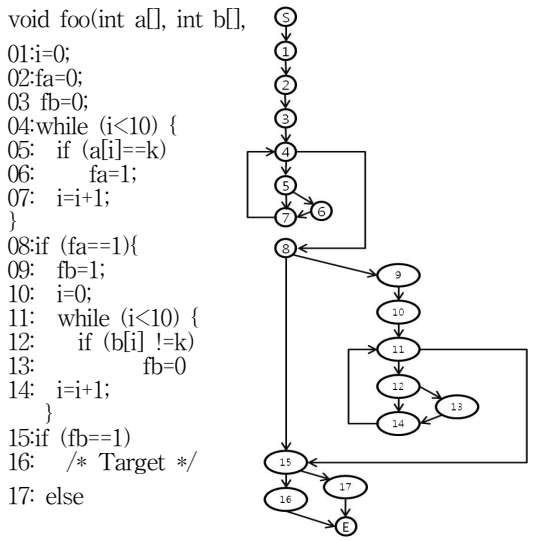
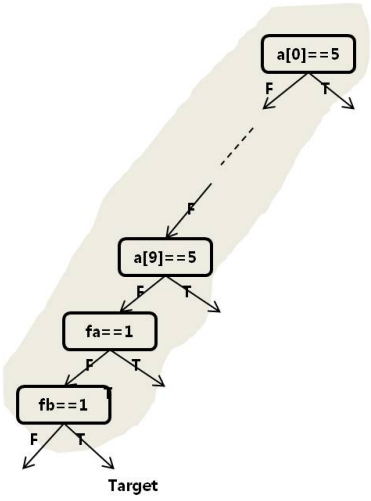
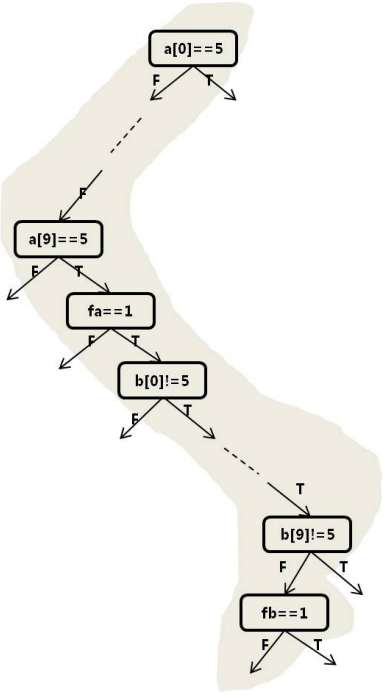


그림 1. 예제 프로그램과 제어 흐름 그래프
Fig. 1. An example program and the corresponding control flow graph

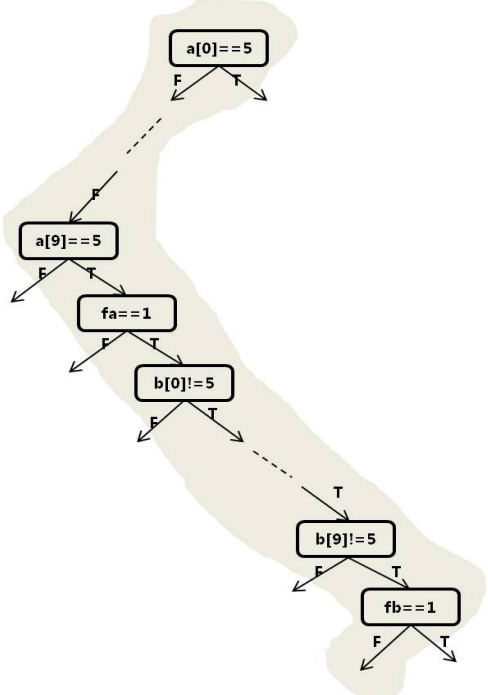
Now we introduce some basic notions about dataflow between program statements. $D(s_i)$ denotes a set of program variables defined at statement s_i and $U(s_i)$ a set of program variables used or referenced at s_i. For a branch b_i, $\triangle b_i$ denotes a set of statements that can be executed if b_i is executed. More formally, $\triangle b_i$ includes the statements *dominated* by b_i[9]. We say that statement s_i *affects* statement s_j iff there exists a path $\langle s_1, s_2, \cdots, s_n \rangle$ such that for some v, $v \in D(s_i) \cap U(s_j)$ and for all k, $i \le k \le j$, $v \notin D(s_k)$.



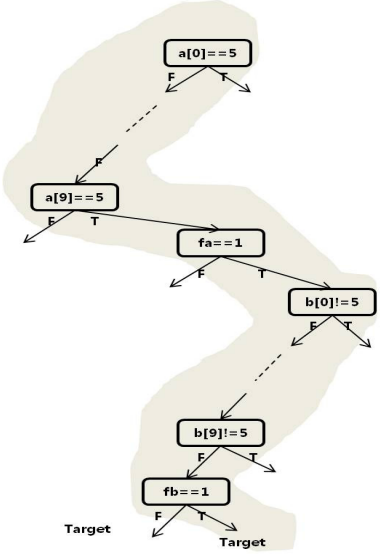
(a)



(c)



(b)



(d)

그림 2. GCT에 의해 생성된 계산 트리
Fig. 2. Computational trees produced during GCT

2. GCT Procedure

Fig. 3 shows the GCT procedure. The GCT procedure starts running the program on a randomly generated input. If the target is hit luckily, the procedure is over (line 1). When the target is not executed, the procedure seeks an alternative path to direct the search towards the target by selecting a guiding branch along the execution path is selected (line 2). The prefix of the alternative path is identified by conjoining the symbolic constraint just before executing the guiding branch with the constraint generated by its paired branch of the guiding branch. This is done by the MakePC procedure. The loop (line 3-line 23) runs until the target is traversed or no guiding branches to be investigated exist.

```
Procedure GCT(Program P, Target t) /* Let a randomly generated input be I
and π be the path of the program P executed on I. */
1:   if the execution hits the target
     then found=true else found=false
     end if /* I is the required test data */
2:   if (GB_π(t) ≠ ∅)
     then pick a branch bt from GB_π(t) E=MakePC(π,bt)
     else termin_cond=true
     end if
3:  while not (found or termin_cond)) do
4:     if (there exists a satisfying assignment I for E) then
5:           run P with I and let π° be the path executed on I
6:        if (π° hits the target t) then found=true
            /* I is the required test data */
7:        else if (π=π°) then /* the same path is executed again */
/* Let w(bt) be the set of statements affecting the branch predicate of bt and
b be a guiding branch before w(bt) on  where paired(b) has not yet been
executed */
8:           if b∈ GB(t) such that (△(paired(b))∩w(bt)≠∅ or
               (△b∩w(bt )≠∅)
             then
9:              E=MakePC(π , b)
10:             bt = b
11:          else termin_cond=true;
12:          end if
13:       else
14:          if (GB(t)≠∅) then
15:             pick a branch bt from GB(t);
16:             E=MakePC(,bt)
17:             π =π°
18:          else termin_cond=true;
19:          end if
20:          end if
21:       else termin_cond=true;
22:       end if
23:   end while

24:   if found then print I; /* I is a required test data */
25:   else print "Not Found"
end procedure

PC MakePC(Path π, Branch b) /* PC: Path Constraint */ /* Let Φ_t be the
symbolic constraint just before executing b and Φ_t be the constraint generated
by b with respect to the path π */
        return  E←(∧_{φ∈Φ_t} φ)∧not Φ_t
end MakePC
```

그림 3. GCT 프로시듀어
Fig. 3. The GCT Procedure

If solutions satisfying the resultant constraint are supplied as inputs, the new execution will follow the previous execution up to the selected guiding branch, but then take the paired branch of the guiding branch. This ensures that the new execution will take the other branch of the one taken by the previous execution. For a selection criterion of guiding branches, we select a branch with the minimum distance to the target in terms of the branches between each guiding branch and the target.

The code fragment (line 7-line 12)addresses the flag variable problem. The flag variable problem arises when Boolean-valued variables are used in branch predicates and then makes it difficult to direct the search of the target[6][7]. It is notnecessarily restricted to flag variables. The same problem arises if branch predicates use internal variables which are not represented in terms of input variables. Such branch predicates will not produce any symbolic constraints. Consequently, the execution path taken by the new execution will be the same as the one taken by the previous execution.

In cases where the new execution follows the previous execution (line 7), we seek a path that is different from previously executed path by selecting a new guiding branch, say b, such that execution of the paired branch of b i.e., $paired(b)$, forces the statements

affecting the previously selected branch, say b_t, to be executed. The condition '$(\triangle(paired(b))\cap(b_t)\neq\varnothing$' at line 8 allows us to identify such a guiding branch. Alternatively, if certain statements affecting b_t have been already executed through theguiding branch b (the condition '$(\triangle b\cap w(b_t)\neq\varnothing$' at line 8), then we mask their influence on b_t by executing the paired branch of b. If the new execution path is different from the previous path, then we select a new guiding branch with respect to the new path with the hope of traversing the target in the next iteration (line 14-line 19).

3. An example

We now revisit the C program and its flow graph shown in Fig. 1 along with their associated derived computational trees shown in Fig. 2 to illustrate the GCT strategy. Suppose that the target is the branch (15, 16). In order to execute the target, at least one of the elements of array a must be a constant 'k' while all elements of array b equal the value of 'k'.

Assuming that the initial execution is on inputs, $0\leq i\leq 9$, a[i]=0, b[i]=0,k=5 the target is not executed, but its paired branch (15, 17) is executed (Fig. 2(a)). This execution passes through the conditional statement at line 5 ten times, yielding the constraints E=(a[0]≠5)∧(a[1]≠5)∧…∧(a[9]≠5). Note that the constraints generated by the branches (8, 15) and (15, 17), i.e., (fa≠1)∧(fb≠1) are not appended to E because both the variables 'fa' and 'fb' are internal variables which are not represented in terms of input variables.

The branch (15, 17) is chosen as the guiding branch because it has the minimum distance to the target, i.e., *paired*((15,17))=(15,16)=target. The search negates the guiding branch (15, 17) to force execution through the branch (15, 16) by solving E, undoubtedly yielding test inputs which follow the initial execution because the constraint (fa==1) generated by negating the guiding branch (15, 17) leaves E unchanged.

The GCT procedure checks what statements affect the branch predicate, i.e., fb==1. The assignment at line 13 affects the branch predicate and thus the branch (8, 9) is selected for the next execution. However, negating the guiding branch (8, 15) also leaves the symbolic constraints E intact, thus yielding the previous execution path.

We are now in a position to identify and execute the statements affecting the branch predicate, i.e., fa==1. The assignment at line 6, i.e., 'fa=1' affects the branch predicate and then the execution is forced through the branch (5, 6) by solving the constraint (a[0]≠5)∧(a[1]≠5)∧…∧(a[8]≠5)∧(a[9]=5), perhaps yielding a[0]=0, a[1]=0, …, a[8]=0, a[9]=5. These inputs give execution which reaches the branch (5, 6), yielding the execution path depicted in Fig. 2(b). Although the new execution path does not traversethe target, it allows us to pick up the branch 'b[9]!=5' as the guiding branch and to negate it, leading to the new constraints E=(a[0]≠5)∧(a[1]≠5)∧…∧(a[9]=5)∧(b[0]≠5)∧(b[1]≠5)∧…∧(b[9]=5). This process is repeated until all elements of the array b equal to 5, givingthe desired test inputs (Fig. 2(c) and Fig. 2(d)). Note that a branch can come up with several instances during execution due to loops. When we are required to choose between branch instances, the last instance is firstly taken into account in this paper. Of course, you can pick it randomly. This paper does not address what branch instances produce best results.

Ⅳ. Experimental results

GCT has been implemented in CREST[2] which is open source software for generating test data for C with concolic testing. In order to extract data flow information from source code, we have also extended CIL which CREST uses to perform the code instrumentation for symbolic execution. We have conducted the experiments on a Linux machine with a 2.8 GHz Pentium 4 processor.

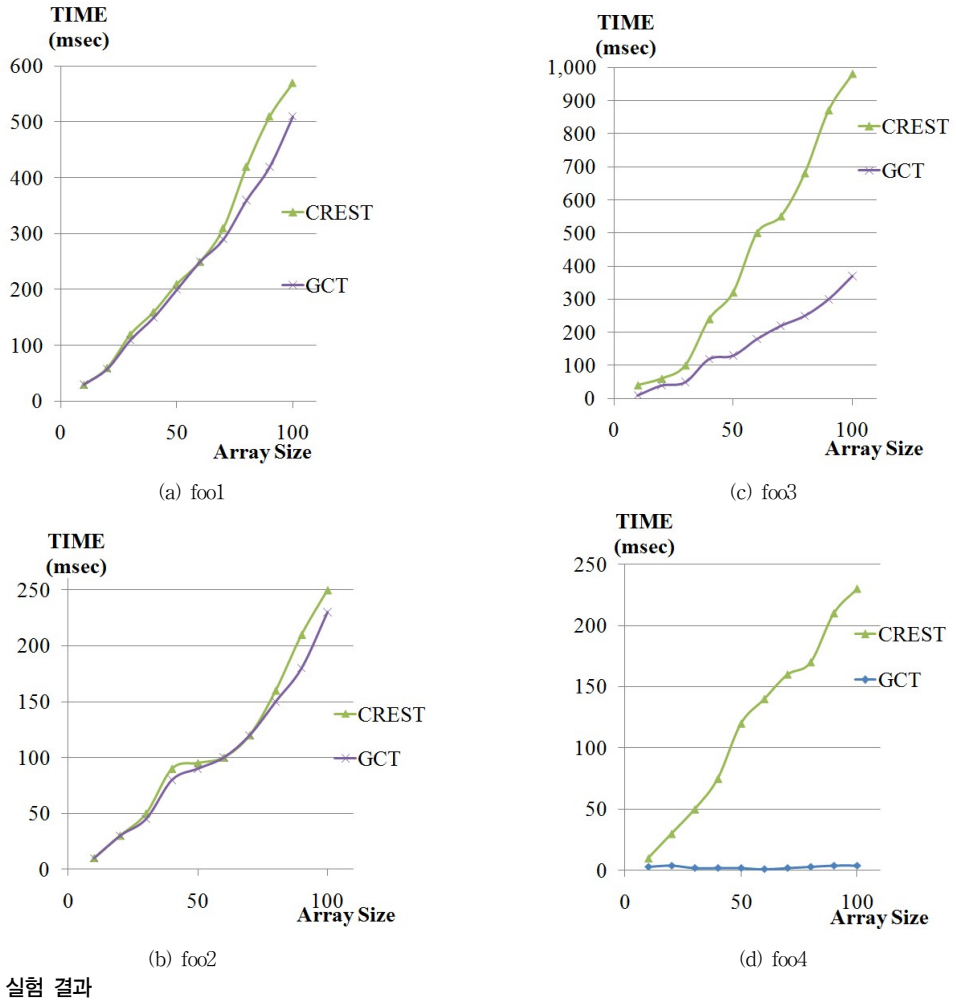


그림 4. 실험 결과
Fig. 4. Experimental results

We used CREST to evaluate concolic testing on several programs from the literature[6][7]. These programs introduce particular technical difficulties for tackling the flag variable problem even though they are relatively small programs. CREST was able to generate required test inputs for all programs used in the experimental study in less than one hundred milliseconds. In addition, the time required to search targets is nearly identical for both CREST and GCT. The results show that concolic testing can be successfully applicable to test data generation for programs with flag variables.

We took a closer look at programs included in the experiments. Most of these programs require that all of input values be the same so that flag variables will yield desired values, i.e., all array elements are set to a special value. Meeting this requirement necessitates exhaustively exploring the entire path space preceding the target. To explore this issue, we have conducted further experiments on four versions of the foo function in Fig. 1. Targeting the assignment at line 16 requires the following test inputs:

- foo1: The foo function is modified so that, when all of the array values of a and b are five, the target is executed.
- foo2: The foo function is modified so that, when

all of the array values of a are five, and at least one of the array values of b is five, the target is executed.

- foo3: It is the same as foo where the target is executed on at least one of the array values of a is five and all of the array values of b are five.
- foo4: The foo function is modified so that when, for each of the arrays a and b, at least one of the array values is five, the target is executed.

The comparison result of CREST with GCT for those programs is plotted in Fig. 4. The x-axis gives the array size and the y-axis gives the search time elapsed by CREST and GCT to find the target. The two plots in Fig. 4(a) almost overlap, indicating no difference between CREST and GCT. In contrast, the plots in Fig. 4(c) show a significant difference in the search time. This can be illustrated by the size of path space that must be explored before the target is reached. The foo1 function requires that all array elements including both a and b should be set to five. In order to generate such test inputs, GCT has to explore all execution paths caused by the conditional, i.e., 'if (a[0]==5) …', 'if (a[1]==5)',…, 'if (a[k]==5)…' for the array size k, as CREST does. Similarly, the plots given in Fig. 4(b) can be illustrated even though CREST continues to explore the other paths instead of stopping even after the target is executed.

In case of the foo3 function, however, GCT explores just one path out of those execution paths which sets a[i] to five to reach the target. In general, only a small fraction, $1/2^k$, of the path space explored by CREST is required by GCT when the array size of a is k. The search time increases as the size of the array b increases because both GCT and CREST require that all of the array values of b should be set to five.

The differences between GCT and CREST are manifest in Fig. 4(d) where for the foo4 function, the search time taken by GCT is almost a constant value even though the array size increases. In this case, GCT does notdepend on the number of execution paths prior to the target. GCT needs to explore just one execution path setting one array value of a and one array value of b to set to five whereas CREST explores all possible execution paths. The size ratio of the search spaces to be explored by GCT and CREST is $1/2^{m+n}$ when the array size of a is m and the array size of b is n.

In addition, it is worth observing how GCT and CREST work when a target point is infeasible. One important goal of automated test data generation is not to search test data needlessly when they could not be found. We conducted another experiments to demonstrate how GCT and CREST address an infeasible target point with the program in Fig. 5.

```
void  infeasible(char c[], char s[]){
01:  state = 0;
02:  if(c[0] == 'A')
03:       if(state == 0)              state = 1;
04:       else          printf("state:!0\n");   //target
05:  if(c[1] == 'B')
06:        if(state == 1)             state = 2;
07:        else                       printf("state:!1\n");
08:  if(c[2] == 'C')
09:        if(state == 2)             state = 3;
10:         else                      printf("state:!2\n");
11:  if(c[3] == 'D')
12:        if(state == 3)             state = 4;
13:         else                      printf("state:!3\n");
14:  if(c[4] == 'E')
15:        if(state == 4)             state = 5;
16:         else                      printf("state:!4\n");
17:  if(c[5] == 'F')
18:        if(state == 5)             state = 6;
19:         else                      printf("state:!5\n");
20:  if(c[6] == 'G')
21:        if(state == 6)             state = 7;
22:         else                      printf("state:!6\n");
23:  if(c[7] == 'H')
24:        if(state == 7)             state = 8;
25:         else                      printf("state:!7\n");
26:  if(c[8] == 'I')
27:         if(state == 8)            state = 9;
28:         else                     printf("state:!8\n");
29:  if ( s[0] == 'r' && s[1] == 'e' &&
              s[2] == 's' && s[3] == 'e' &&
              s[4] == 't' && s[5] == '!' && state==9)
30:         printf("ERROR!");
}
```

그림 5. 실행 불가능한 목표를 지닌 예제 프로그램
Fig. 5. An example program with an infeasible target

The comparison result of CREST with GCT for those programs is plotted in Fig. 6 for the targets at line 4, line 7, line 10, line 13, line 16, line 19, line 22, line

25, line 28, and line 30, respectively.

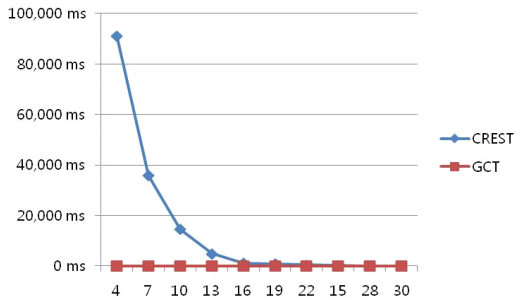


그림 6. 실행 불가능한 목표에 대한 실험 결과
Fig. 6. Experimental results for the infeasible results

The *x*-axis gives the target points of the program and the y-axis gives the search time elapsed by CREST and GCT to find them. The plot for GCT is almost constant indicating no difference between the target locations. This means that GCT is almost independent of where the targets are located. In contrast, the plot for CREST shows a significant difference in the search time depending on the locations of the targets. As the target is located nearer at the program beginning, the search time is abruptly increased. This is strongly related to the way of how CREST works. In the experiments, we consider the DFS strategy that CREST employs as one of its search strategies. As the name implies, the DFS strategy searches firstly the deepest part of the search space whenever possible. Thus, the 'printf' statement at line 30 is the first one explored because it is located at the deepest part among the targets.

The most notable difference between GCT and CREST in terms of the search time can be observed in line 4. The 'else' branch' at line 4 is infeasible because the variable 'state'is initialized with 0 at line 1 and is not altered afterwards even though it needs to set to a value other than zero to be executed. In this case, CREST endlessly explores the search space unless certain termination conditions are given. On the other hand, GCT stops searching in finite iterations. Assuming that the initial execution is on inputs such that c[0] 'A'. Then, the target, line 4, is not executed. The search negates the guiding branch (2, 5) by setting c[0] to 'A'. However, the execution passes through the branch (2, 3). Because the branch (2, 3) involves the internal variable 'state', GCT procedure tries to find what statements defines the variable 'state'. GCT does not explore the search further because only the assignment at line 1 defines the variable and has been already executed.

V. Concluding remarks

The proposed GCT seeks the solution with far fewer trials by restricting the number of program paths that are explored. GCT uses data flow information to identify statements that should be executed beforehand in order for the target to be executed. We conducted experiments to evaluate the performance of GCT. Results demonstrate that GCT can be effective for certain classes of programs. Unlike concolic testing, GCT is *directed* in that its process for test data generation is biased towards a particular goal. Further work is needed to demonstrate that GCT will be effective on large-scale programs.

References

[1] P. Godefroid, N. Klarlund, K. Sen, "DART: Directed automated random testing", Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, Illinois, pp. 213-223, 2005.

[2] J. Burnim, K. Sen, "Heuristics for dynamic test generation", Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering , pp. 443-446, 2008.

[3] R. Santelices, P. K. Chittimalli, T. Apiwattanapong,

A. Orso, M. J. Harrold, “Test suite augmentation for evolving software”, Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 218-227, 2008.

[4] N. Rungta, E. G. Mercer, W. Visser “Efficient testing of concurrent programs with abstraction -guided symbolic execution”, Proceedings of SPIN Workshop on Model Checking of Software, Grenoble, France, pp. 218-227, 2009.

[5] I.S. Chung. J. Park, "Goal-oriented concolic testing", Journal of KIISE: Software and Applications, vol. 37, n. 10, pp. 768-772, 2010.

[6] L. Bottaci, “Instrumenting programs with flag variables for test data search by genetic algorithm”, Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1337-1342, 2002.

[7] P. McMinn, M. Holcombe, Evolutionary testing using an extended chaining approach, Evolutionary Computation, vol. 14, no. 1, pp. 41-64, 2006.

[8] R. Ferguson, B. Korel, The chaining approach to software test data generation, ACM Transactions on Software Engineering and Methodology, vol. 5, no. 1, pp. 63-86, 1996.

[9] S. Muchnick, N. Johnes, Program Flow Analysis, Theory and Applications, Englewood Cliffs, NJ, Prentice-Hall International, 1981.

※ 본 연구는 한성대학교 교내연구장려금 지원과제임.

저자 소개

정 인 상(Insang Chung)(정회원)

- 1987년 서울대학교 컴퓨터공학과 졸업(학사)
- 1989년 한국과학기술원(KAIST) 전산학과 졸업(석사)
- 1993년 한국과학기술원(KAIST) 전산학과 졸업(박사)
- 1999 ~ 현재 : 한성대학교 컴퓨터공학과 교수

<관심분야 : 소프트웨어 공학, 소프트웨어 테스팅>