테스트 데이터 자동 생성을 위한 입력 변수 슬라이싱과 진화 알고리즘 적용 방법

최효린, 이병정
서울시립대학교 컴퓨터과학부
e-mail : {yoinoichr2015, bjlee}@uos.ac.kr

Applying Evolutionary Algorithms with Slicing Input Variables to Support Automation of Generating Test Data

Hyorin Choi, Byungjeong Lee
Dept. of Computer Science, University of Seoul

요 약

소프트웨어 테스트는 시스템의 신뢰도를 판단하는 중요한 작업이지만, 많은 노력과 비용을 필요로 한다. 모델 기반 테스트는 이러한 비용을 줄이기 위한 방안으로써 제안되었다. 정형적 모델로부터 시스템의 실행 가능한 경로를 파악하고, 각 경로마다 입력 값을 생성하여 테스트를 수행한다. 이 때, 적절한 입력 값을 찾기 위해 메타-휴리스틱 기법을 사용하는데, 기존의 알고리즘은 목적 경로와 관련이 없는 변수까지 구분없이 고려하기 때문에 시스템이 복잡할수록 불필요한 연산이 많아지는 문제가 있다. 본 논문은 슬라이싱 기법과 우선순위 정책을 적용한 테스트 데이터 자동 생성 기법을 제안하며, 실험을 통해 기존의 방법보다 효과적으로 테스트 데이터를 생성함을 보인다.

1. 서론

개발된 소프트웨어가 요구사항에 맞게 동작하는지 확인하는 작업은 매우 중요하다. 그러나 개발된 시스템에 대한 적절한 테스트를 수행하는 것은 전체 유지보수 비용의 40%를 차지할 정도로 막대한 노력과 비용이 필요한 작업이다[1].

모델 기반 테스트(Model-based Testing, MBT)[2]는 이러한 비용을 줄이기 위해 테스트 설계 및 테스트 데이터 생성을 자동화하는 방안으로써 제안되었다. 시스템 요구사항을 UML, OCL 등 표준 언어로 작성한 모델을 분석하여 실행 가능한 경로(Executable Path, EP)를 찾으면, 임의의 입력 값을 수행시킨 경로와 비교하는 과정을 통해 테스트를 수행한다. 각 실행 경로에 해당하는 입력 값을 생성하는 방법으로는 메타-휴리스틱(Meta-Heuristic)[3] 기법을 적용하여 동적으로 생성하는데, 대표적인 알고리즘으로 유전 알고리즘(Genetic Algorithm, GA)[1] 이 많이 사용된다.

그러나 테스트하려는 시스템이 복잡할수록 기본적인 GA 로는 많은 연산을 필요로 하며, 지역 해 문제에 빠지게 되는 문제가 있다. 이러한 문제를 극복하기 위해 메타-휴리스틱 기법을 개선하는 많은 연구들[4]이 제안되었지만, 이는 알고리즘의 복잡도를 증가시켜 테스트 대상 시스템(System Under Test, SUT)에 대한 최적화 노력이 필요하다는 문제를 야기한다[5, 9].

본 논문은 앞서 언급된 문제들을 회피하면서, 동시에 성능 향상을 위한 새로운 방안을 제시한다. 제안하는 방법은 시스템의 실행 경로의 각 분기마다 관련된 변수들을 슬라이싱 기법[5]을 통해 추출하여, 이를 GA 의 교차 및 변이 연산에 사용함으로써 불필요한 연산을 줄여 알고리즘의 성능 향상을 꾀한다. 또한 지역 해 문제를 해결하기 위해 3 단계 우선순위 정책을 적용하였으며, 실험을 통해 제안하는 방법이 기존 알고리즘보다 효과적으로 테스트 데이터를 생성할 수 있음을 보여준다. 본 논문의 기여도는 다음과 같다.

- 본 논문의 방법은 테스트 데이터 자동 생성을 지원한다.
- 본 논문에서 제안하는 방법이 기존의 방법보다 효과적임을 실험을 통해 보인다.

2 장에서는 관련 연구에 대해 기술하고, 3 장에서는 제안하는 방법에 대해 설명한다. 4 장은 실험의 결과를 통해 제안하는 방법의 효과성을 확인한다. 5 장에선 본 논문에 대한 결론과 토의를 수행한다.

2. 관련 연구

Korel[6]은 경로를 기반한 휴리스틱 알고리즘을 적용한 테스트 데이터 자동 생성방법을 제안하였다.

* 이 논문은 2016 년도 정부(미래창조과학부)의 재원으로 한국연구재단-차세대정보·컴퓨팅기술개발사업(NRF2014M3C4A7030504)과 서울시의 재원으로 수행한 서울시 창조전문인력 양성사업(No.CAC1510)의 지원을 받아 수행된 연구임.

목적 경로를 해결하기 위해 현재 테스트 데이터가 수행한 경로와의 차이를 분기 거리(Branch distance)와 근사 수준(Approximation level)으로 계산하고, 입력 값에 변화를 주어 보다 적절한 테스트 데이터를 선택하여 모든 경로에 대한 입력 값을 찾아내는 방법이다. 이는 모델 기반 테스트의 테스트 데이터를 자동으로 생성함에 있어 가장 일반적으로 사용되는 방법이다.

S. Back[7]은 Korel 의 방법을 기반으로, 동시성을 포함하는 프로그램의 경로 탐색에 대한 새로운 방법을 제안함으로써, 테스트 데이터를 효과적으로 생성하는 문제에 대해 알고리즘 자체에 대한 개선 이외의 접근 방법을 제안하였다는 점에서 본 논문의 방향성과 유사하다. 그러나 동시성을 포함하는 프로그램의 경우에만 효과를 볼 수 있는 한계가 있다.

M. Harman[8]은 분기 커버리지(Branch coverage)를 달성하기 위해 코드 정적 분석을 기반으로 입력 값의 범위를 줄이는 방법을 제안하였다. 관련성이 적은 변수에 대해선 알고리즘을 수행하지 않는다는 점에서 본 논문과 유사하지만, 관련되는 변수를 사용자가 판단해야 하며, 시스템의 수행 경로에 대해서는 고려하지 않고, 소스 코드에 의존적인 문제가 있다.

3. 테스트 데이터 생성 방법

이 장에서는 본 논문이 제안하는 방법을 기술한다. 이해를 돕기 위해 간단한 구조를 가지는 함수를 예시로서 사용하도록 한다.

3.1 입력 변수 슬라이싱

그림 1 은 함수 foo 의 구조를 보여준다. int 형태의 값 3 개를 매개 변수로 받아 각 변수 i, j, k 에 초기화하고, 변수 i, j 값이 양수이면 변수 k 에 추가한다. 결과가 0 이상이면 k, 아닐 경우 0 을 return 한다. 소스 코드는 Java 로 작성되었으며, 모델은 UML 의 활동 다이어그램(Activity Diagram)이다. 각 조건을 지나는 식을 참(T), 거짓(F), 그리고 고려하지 않음(X)이라고 표현할 때, 주어진 모델을 기반으로 생성할 수 있는 모든 경로는 표 1 과 같다.

```
1  int foo(int input1, int
2  input2, int input3){
3     int i = input1;
4     int j = input2;
5     int k = input3;
6     if( i > 0 ){ //조건 1
7        k = k + i;
8        if ( j > 0 ){//조건 2
9          k = k + j;
10       }
11    }
12    if( k > 0 ){//조건 3
13       return k;
14    }
15    return 0;
16 }
```

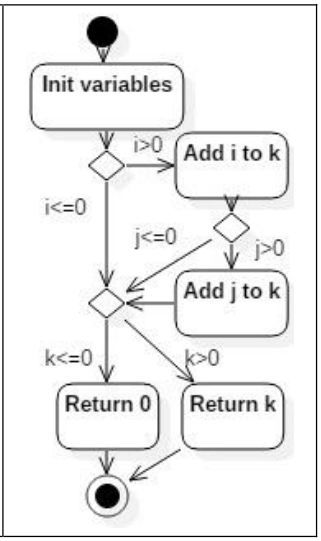

그림 1 사용 예시 함수 foo

<표 1> 함수 foo 모든 경로

Label	Path	Label	Path
P1	{ F, X, F }	P4	{ T, T, F }
P2	{ F, X, T }	P5	{ T, T, T }
P3	{ T, F, T }	P6	{ T, F, F }

만약, 분기 커버리지가 목표이면 목적 경로로써 {P1, P3, P5} 이나 {P2, P3, P4} 등을 생성하게 된다[2].

적합한 테스트 데이터를 찾기 위해선 우선 초기 입력 값 집합을 생성하여 이를 SUT 에 수행시켜 실행 경로(Execution path)를 생성하고, 각 실행 경로가 목적 경로와 비교하여 얼마나 적합한지 평가한 후 입력 값의 변화를 통해 상이한 분기를 맞춰 나가야 한다. 예를 들면, 목적 경로가 'P3', 현재 실행 경로가 'P6'일 때, 상이한 분기는 1 개며, 현재의 입력 값을 변화시켜 조건 3 식에 참인 입력 값을 찾는다.

일반적인 유전 알고리즘에서는 입력으로 받은 변수 3 개를 모두 고려하여, 'P3'에 맞는 입력 값을 찾는다. 그러나 조건에서 활용되는 변수를 '조건 변수'라 했을 때, 실제로 'P3'에서 조건 3 의 조건 변수 'k'에 영향을 줄 수 있는 입력 값은 Line 5 에서 사용된 'input3'과, Line 7 에서 사용된 'input1'이며, 'input2'는 해당 분기를 해결하는 데에 영향을 주지 않는다. 따라서, 위와 같은 상황에서는 {input1, input3}을 연관된 '입력 값 집합(Input Value Set, IVS)'으로 보고, 이를 알고리즘의 휴리스틱 연산에 활용한다면 테스트 데이터를 보다 빠르게 찾아낼 수 있게 된다.

3.2 우선순위 정책

실제 시스템에서 조건 변수와 입력 값과의 의존성을 슬라이싱 기법 만으로 판단하는 것은 IVS 가 전체 입력 값과 동일해 지거나, 지역 해 문제를 극복하지 못하게 되는 어려움이 있다. 이를 극복하기 위한 방법으로 3 단계의 우선순위를 두어, 일정 반복 횟수 이상 더 좋은 해를 찾지 못한다면 사용되는 IVS 는 목적 경로를 해결하기에 적절하지 않다고 보고, 더 다양한 변수를 고려하기 위해 IVS 에 포함되는 범위를 확장하여 휴리스틱 연산에 고려한다. 제안하는 단계는 표 2 에 정리하였다.

<표 2> 우선순위 단계

Step	Description	Example
Level 1	조건 변수와 직접 연관된 IVS	{ input3 }
Level 2	현재 실행 경로를 Slicing 하여, k 값의 변화에 영향을 주는 IVS	{ input1, input3 }
Level 3	전체 IVS	All

우선순위 단계는 기본적으로 1 단계부터 시작한다. 위와 같은 상황에서, 1 단계에 속하는 IVS 는 조건 변수 k 에 직접적인 영향을 주는 input3 을 가진다.

다음 단계는 목적 경로를 슬라이싱하여, 조건 변수에 변화를 주는 변수를 포함한 IVS 를 생성한다. 위와 같은 상황에서는 input3 에 input1 을 같이 고려한다.

<표 3> 테스트 대상 시스템(SUT)

Label	Function name	Branches	Target Paths	Parameter	Approximate domain size(10^x)
S1	validDate[7]	14	11	int: 4	7
S2	triangle2[10]	20	8	int: 3	14
S3	triangle3[5,11]	20	8	int: 3	14
S4	triangle4[3,12]	26	14	int: 3	14

마지막 3 단계는 2 단계의 IVS 으로도 적절한 테스트 데이터를 찾을 수 없을 때, 모든 입력 변수들을 고려한다. 이는 기존 GA 방식과 동일하며, 가능한 모든 경우를 다 수행함으로써 지역 해 문제를 벗어나는 데에 사용된다. 해당 목적 경로의 해를 찾기 위해 알고리즘을 반복한 횟수를 I_{TARGET}, 더 나은 해를 찾지 못하는 상태로 반복한 횟수를 I_{FROZEN} 이라 할 때, 우선순위 단계를 판단하기 위한 조건은 다음 식으로 정리된다.

$$\mathrm{T} = \frac{I_{TARGET} - I_{FROZEN}}{I_{TARGET}}, \qquad IVS_{level} = \begin{cases} 1, & I_{FROZEN} < N \\ 2, & \theta < \mathrm{T} \\ 3, & \theta \geq \mathrm{T} \end{cases}$$

N 은 GA 의 초기 인구 수이며 현재의 실행 경로로는 더 나은 해를 찾을 수 없을 때, 다음 단계로 진행한다. θ는 임의의 값으로 계산식 T 가 임계치를 넘지 않을 땐 2 단계, 넘으면 3 단계를 적용한다.

3.3 진화 알고리즘 적용 방법

3.1 절과 3.2 절에서 기술한 내용을 GA 에 적용하여 그림 2 와 같이 의사 코드로 정리하였다.

```
1  Generate all target paths TP_ALL = {TP_1,…,TP_n}
2  Generate an initial population X = {x_1, …, x_n}
3  Loop while (not found all solution S)
4     Execute X to find solution for TP_i
5     Find a best solution x_j from X
6     If x_j is fit for TP_i Then
7        Add to Solution S_TP = x_j
8        Continue;
9     End If
10    Set input variable set with priority IVS_PL
11    For k = 1 to n DO
12       If crossover probability CP Then
13          Select the worst solution x_z from X
14          x_z = Crossover(x_k, x_z, IVS_PL)
15       Else
16          Mutate(x_k, IVS_PL)
17       End If
18    End Do
19 End while
   Terminate Algorithm
```

그림 2 제안하는 알고리즘

Line 1~2 에서는 시스템의 목적 경로를 파악하고, 유전 알고리즘의 초기 인구(population)을 생성한다.

Line 3 부터 반복문을 시작하는데, Line 4~9 에선 생성된 데이터(X)를 SUT 에 실행시켜, 도출된 실행 경로 중에서 가장 목적 경로(TP_i)에 적합한 데이터(x_i)를 찾는다. 이 때 적합도 계산에 사용되는 함수는 Korel [6]이 제안한 방법을 적용한다. 찾아낸 데이터가 목적 경로의 해(S)를 만족하면, 해당 데이터를 추가하고, 아닌 경우 다음으로 진행한다. Line 10 에서 해당 목적 경로의 입력 변수 집합(IVS_{PL})을 우선순위를 고려하여 생성하면, Line 11~18 에서는 초기 데이터 집합(X)의 모든 요소들에 대해, IVS 를 고려한 교차 및 변이 연산을 수행한다. Line 12~15 는 GA 의 교차 연산으로, 목적 경로와 가장 동떨어진 데이터(x_z)와 순차적으로 선택된 데이터(x_k)를 2 진수화하여 임의로 교차선택을 수행한다. Line 16 은 변이 연산으로, 순차 선택된 데이터(x_k)에 일정 범위 내 임의의 값을 부여한다. 이러한 과정을 충분히 반복하여, 모든 경로에 대한 최적 해를 찾으면 알고리즘을 종료한다.

4. 실험

4.1 실험 환경

본 실험은 랜덤(Random) 알고리즘과 유전 알고리즘(GA), 유전 담금질 기법(GSA)[5], 그리고 본 논문이 제안하는 기법(GA/SV)을 4 개의 테스트 대상 시스템(SUT)에서 수행하였다. 사용되는 프로그램은 본 실험의 동일한 실험 환경을 조성하기 위해 Java 언어로 작성하였고, 도구는 Java IDE 인 Eclipse 의 Neon.2 버전을 사용하였다.

<표 4> 실험 매개변수

Algorithm	Parameter	Value
GA, GSA, GA/SV	Population size	N=50
	Crossover probability	0.99
GSA	Temperature	T=1000
	Neighborhood size	b=50
GA/SV	Priority level threshold	θ=0.85

표 3 은 SUT 를 요약하여 나타낸 표다. 각 SUT 의 분기 수와 매개변수의 형태, 근사 도메인 크기를 보여준다. 매개변수인 int 는 도메인 범위를 기본적으로 '-32768'부터 '32767'까지를 지정하였으며, validDate 에서 사용되는 int 는 음수를 제외하고, 각기 상이한 도메인 범위를 지정하였다. 실험에 주어진 알고리즘의 매개변수 값은 표 4 에서 정리하였다. 모든 실험은 Intel Xeon CPU X5550(x2), 2.67GHz, 12GB RAM 에서 수행하였다.

4.2 실험 결과

실험은 총 3 가지 관점에서 수행되었다. 첫 번째로 분기 커버리지의 90%를 달성하기 위해 알고리즘을 수행한 시간을 비교한다. 총 20 회 수행한 평균을 냈으며, 단위는 초(second), 소수점은 첫 번째 자리에서 생략한다. 결과는 그림 3 의 그래프로 정리할 수 있다.

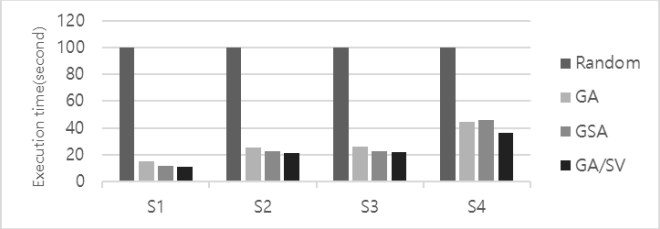

그림 3 SUT 별 알고리즘 수행 시간 비교

결과를 살펴보면, 본 논문에서 제안하는 방법(GA/SV)이 다른 GA 나 GSA 기법 보다 수행 시간이 적음을 알 수 있다. 특히, S1 과 S4 프로그램에서 일반적인 GA 와 비교해 각각 26%, 20% 빨라졌으며, GSA 와 비교하여도 5% 이상 개선되었음을 확인할 수 있다. S2 와 S3 프로그램에서도 GA 와 비교해 약 16% 개선되었으며, GSA 와도 2% 정도로 유의미하게 개선되었음을 보인다. 눈여겨볼 만한 점은 S4 의 경우 GSA 기법이 GA 기법보다 오히려 수행 시간이 많았는데, 이는 GSA 가 지역 해 문제에 빠지는 경우가 많았기 때문이다. 랜덤(Random) 알고리즘의 100 초를 초과하는 부분은 그림 상에서 표현하지 않았다.

두 번째로, 일정한 단위 시간마다 각 알고리즘의 평균 분기 커버리지를 비교한다. 시간의 단위는 5 초, 커버리지는 4 개의 SUT 에 대한 평균 분기 커버리지 값으로 작성하였다. 결과는 그림 4 와 같다.

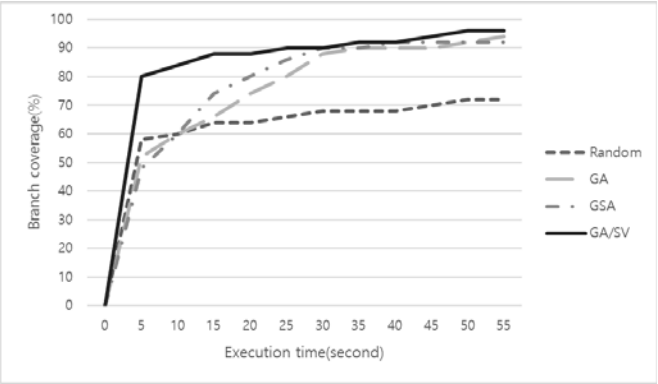

그림 4 평균 분기 커버리지

결과를 살펴보면, 본 논문에서 제안하는 방법이 여타 알고리즘보다 커버리지를 빠르게 달성하고 있음을 확인할 수 있다. 특히 0~5 초 구간에서 다른 알고리즘과 비교하여 약 20% 이상 차이를 보이는데, 이는 실험에 사용된 SUT 의 각 경로에서 불필요한 연산을 제외하여 빠르게 전역 해를 찾았음을 의미한다. GA 와 GSA 의 그래프는 유사한 형태를 보이지만, 15 초 이후부터는 GSA 가 좀 더 빠르게 커버리지를 달성한다.

마지막으로 알고리즘을 180 초 간 수행할 경우 달성하는 최대 분기 커버리지와, 이를 30 회씩 반복한 평균 소요 시간(초)을 기록하여 표 5 에 정리하였다. 평균이 160 초 이상이면, 표기를 생략하였다.

<표 5> 테스트 대상 시스템 별 분기 커버리지

SUT	Random	GA	GSA	GA/SV
S1	100%(136s)	100%(43s)	100%(29s)	100%(20s)
S2	100%	100%(77s)	100%(61s)	100%(45s)
S3	100%	100%(86s)	100%(58s)	100%(53s)
S4	77%	100%(113s)	100%(80s)	100%(77s)

5. 토의 및 결론

본 논문은 테스트 데이터 자동 생성을 지원하는 개선된 알고리즘을 제안한다. 실험을 통해 기존의 방법과 비교하여 우수한 성능임을 확인하였다. 또한, 제안하는 방법이 조건 변수에 따라 입력 변수를 제한함으로써 알고리즘의 성능이 향상됨을 보였으므로, 본 실험의 SUT 보다 입력 변수가 많은 프로그램에서 더욱 효과를 보일 것으로 기대된다.

향후 연구로 더 많은 프로그램에서 실험을 수행하여 제안하는 방법의 개선을 위한 추가적인 요소들을 분석하고, 사용성을 고려한 도구로 개발할 예정이다.

참고문헌

[1] P. C. Jorgensen, *Software Testing: a Craftsman's Approach*, CRC press, pp. 3-76, 2013.

[2] M. Utting, and B. Legeard, Practical Model - Based Testing: A Tools Approach, pp. 1-18, 2010.

[3] P. R. Srivastava, V. Ramachandran, M. Kumar, "Generation of test data using meta heuristic approach," TENCON 2008-2008 IEEE Region 10 Conference, pp.1-6, 2008.

[4] D. B. Mishra, R. Mishra, K. N. Das, and A. A. Acharya, "A Systematic Review of Software Testing Using Evolutionary Techniques." In Proc. of Sixth International Conference on Soft Computing for Problem Solving, Singapore, 2017.

[5] McMinn, Phil, "Search-based software test data generation: A survey," Software Testing Verification and Reliability, 2004.

[6] B. Korel, "Automated software test data generation," IEEE Transactions on Software Engineering, pp. 870-879, 1990.

[7] S. Back, H. Choi, J. W. Lee, and B. Lee, "Evolutionary Test Case Generation from UML-Diagram with Concurrency," In International Conference on Computer Science and its Applications, pp. 674-679, 2016.

[8] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," In Proc. of the the 6th joint meeting of the European Software Engineering conference, ACM, 2007.

[9] M. Harman, J. Yue, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," Software Testing, Verification and Validation (ICST), 2015.

[10] H. C. Wang, “A hybrid genetic algorithm for automatic test data generation," *Master's thesis*, Sun Yat-sen University, 2006.

[11] C. C. Michael, G. McGraw, M. A. Schatz, “Generating software test data by evolution,” IEEE Transactions on Software Engineering, Vol. 27, no.12, pp.1085-1110, 2001.

[12] B. F. Jones, H. H. Sthamer, and D.E. Eyres, "Automatic structural testing using genetic algorithms," Software Engineering Journal, Vol. 11, No.5, pp. 299-306, 1996.