요약 해석을 이용한 테스트 데이터 자동 생성 기법

한승희[aO] 강제성[a] 정인상[b] 권용래[a]
[a]한국과학기술원 전산학과, [b]한성 대학교
e-mail: [a]{shehan, jskang, kwon}@salmosa.kaist.ac.kr, [b]insang@hansung.ac.kr

Automatic Test Data Generation Using Abstract Interpretation

Seung Hee, Han[aO] Jae Sung, Kang[a] In Sang Chung[b] Yong Rae Kwon[a]
[a]Dept. of Computer Science, KAIST, [b]Hansung University

요 약

테스트 데이터의 자동 생성은 소프트웨어 테스팅에서 가장 중요하면서도 어려운 부분이다. 대부분의 테스트 데이터 자동 생성에 관한 연구는 명세로부터 테스트 데이터를 자동 생성하는 방식이며 이를 위해 정확한 정형적 명세를 필요로 한다. 본 논문에서는 프로그램을 실행하지 않고 프로그램의 동적인 특성을 분석할 수 있는 요약 해석(abstract interpretation) 방법과 선후 지배 관계(pre-, postdominance relationship)를 이용하여 프로그램 코드로부터 직접 테스트 데이터를 자동 생성할 수 있는 방법을 제안한다.

1. 서 론

테스트 데이터의 자동 생성은 소프트웨어 테스팅에서 가장 중요하고 어려운 부분이다. 그렇기 때문에, 대부분의 상업용 시스템에서 테스트 데이터 생성 과정은 거의 자동화가 이루어지지 않고 있는 실정이다. 또한 대부분의 테스팅 자동화 연구는 명세로부터 테스트 데이터를 자동으로 생성하는 방식이며, 이것이 가능하려면 정확한 정형적 명세가 필수적이다[1]. 그리고, 실제 코드로부터 테스트 데이터를 생성하는 연구 중 Mutation 테스팅 기법을 사용한 방법이 있기는 하나, 테스트하는 동안 너무나 많은 mutant 프로그램을 실행시켜야 한다는 단점이 있다[2].

본 연구에서는 명세가 아닌 프로그램 코드를 요약 해석 기법으로 분석한 뒤, 단정문(assertion) 삽입과 선후 지배관계를 이용함으로써 테스트 데이터를 자동 생성하고자 시도하였다. 요약 해석은 프로그램을 실행하지 않고 프로그램의 동적인 특성을 정적으로 분석할 수 있는 기법이며, 이러한 요약 해석을 기반으로 원하는 프로그램의 특정 컨트롤 포인트에 단정문을 삽입하여 필요한 조건을 확인한다. 여기에 선후 지배관계를 접목시킴으로써 좋은 품질의 테스트 데이터를 자동으로 생성할 수 있다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 기반이 되는 배경 연구에 대해 설명한다. 3장에서는 자동적으로 테스트 데이터를 생성해내기 위한 단계를 요약하고 각 단계마다의 과정을 좀 더 자세하게 서술한다. 4장에서는 논문에서 제안한 방법으로 구현한 툴에 대해 간단하게 설명하고 마지막으로 5장에서는 결론 및 향후 연구 방향을 제시한다.

2. 배경 연구

2.1 요약 해석

프로그램 분석의 목표는 분석과정이 끝난다는 것을 보장하면서, 모든 입력 데이터에 대해서 프로그램을 실행시켜 보지 않고도 실행시(run-time) 행위에 대해 되도록 많은 정보를 얻는 것이다. 그러한 프로그램 분석 기술 중 하나가 요약 해석이다[3]. 요약 해석을 통해서, 실제값(concrete value) 대신 요약값(abstract value)을 이용하여 프로그램 행위에 대해 안전하고 근접한 정보를 얻을 수 있다.

요약 해석은 "의미구조 모음(collecting semantics)"과 요약된 의미구조(abstract semantics)사이의 대응관계를 갈로이스 커넥션(Galois connections)을 이용하여 표현할 수 있다. 래티스(lattice)인 실제 도메인 $(D,\sqsubseteq)$이 있다고 가정하고, 실제 도메인을 근사화하는(approximating) 요약 도메인 $(D^A, \sqsubseteq^A)$을 정의한다. 여기서 $d_1{}^A \sqsubseteq^A d_2{}^A$ 는 $d_1{}^A$이 $d_2{}^A$보다 더 정확한 값이라는 것을 의미한다. 요약화 함수 $\alpha : D \rightarrow D^A$는 D에 있는 원소들을 D^A에 있는 근접한 원소에 대응시킨다. 구체화 함수 $\gamma : D^A \rightarrow D$는 D^A에 있는 요약값을 D에 있는 실제 의미로 대응시키는 함수이다. 두 함수 다 부분 정렬(partial order)을 준수해야 한다. d^A가 d의 타당한 근사값이라는 사실을 다음과 같이 표현할 수 있다. $\alpha(d) \sqsubseteq^A d^A$ 또는 $d \sqsubseteq \gamma(d^A)$. D와 D^A사이의 갈로이스 커넥션을 공식으로 표현하면,

$\forall d \in D : \forall d^A \in D^A : \alpha(d)\ O^A\ d^A \Leftrightarrow d \sqsubseteq \gamma(d^A)$

를 만족하는 $\alpha : D \rightarrow D^A$, $\gamma : D^A \rightarrow D$인 함수의 순서쌍 (α,γ)으로 나타낼 수 있다[4].

요약 해석을 이용하여 프로그램 디버깅시 단정문을 삽입하는 방법이 있다[5]. 본 논문에서는 두가지 단정문을 이용하겠다. 잠정적 단정문(intermittent assertion)은 프로그램의 주어진 컨트롤 포인트에서 결국에는 어떤 성질을 만족해야 한다는 것을 뜻하고, 영속적 단정문(invariant assertion)은 주어진 컨트롤 포인트에서 항상 어떤 속성을 만족해야 함을 요구한다. 따라서, 어떤 컨트롤 포인트에 잠정적 단정문을 참(true)으로 설정하면 그 블록을 적어도 한번은 지나가도록 하는 조건을 구할 수 있고, 영속적 단정문을 거짓(false)으로 설정하면, 항상 그 블록은 지나가지 않도록 하는 조건을 후방향 전달 기법(backward propagation)을 이용해 구할 수 있다.

2.2 지배관계, 슈퍼 블록

선후 지배관계를 이용한 효율적인 테스트 데이터 생성 방법인 이 기법은 프로그램의 커버리지 테스팅의 비용을 효과적으로 줄이기 위해 사용된다[6]. 먼저 프로그램의 흐름도 (flow graph)에서 선후 지배관계를 이용하여 노드 집합의 슈퍼 블록을 찾아내어 분할한다. 그러면, 슈퍼 블록의 어느 한 노드만 커버되더라도 그 슈퍼 블록내의 모든 노드들이 커버된다. 슈퍼 블록은 여러 개의 기본 블록을 가질 수 있다는 점에서 기본 블록과는 다르다.

슈퍼 블록을 구하는 방법은 다음과 같다. 먼저 흐름도로부터 선지배 트리(predominator tree)와 후지배 트리(postdominator tree)를 구하여 선후 지배관계를 표현한다. 그런 다음, 선지배 트리와 후지배 트리를 합쳐서 기본 블록 지배 그래프를 구한다. 이 그래프에서 강하게 연결된 컴포넌트들(strongly connected components)을 찾아내면 바로 이 컴포넌트가 슈퍼 블록이다.

다시, 슈퍼 블록간의 지배관계를 슈퍼 블록 지배관계 그래프 형태로 나타낸다. 이 그래프에서 바깥 노드(leaf node)만 모두 커버하면 내부에 있는 다른 모든 슈퍼 블록들도 다 커버된다. 그러므로, 이 방법을 이용하면, 커버리지 툴에서 테스트 데이터가 커버되는지 체크하기 위해 프로그램에 삽입하는 관찰자(probe)들의 개수를 줄일 수 있고, 따라서, 코드 크기와 실행 오버헤드(overhead)를 줄일 수 있다.

본 연구에서는 실제 이 방법을 적용하여 영속적 단정을 삽입하는 개수를 줄여서 효율적으로 테스트 데이터를 생성할 수 있도록 하였다.

3. 테스트 데이터 자동 생성 기법 연구

3.1 테스트 데이터 자동 생성 순서

본 논문에서 제시하는 방법은 <그림 1>의 순서로 진행된다. 먼저 프로그램 소스 코드를 요약 해석 방법을 이용하여 전방향(forward)으로 분석한다. 사용자가 프로그램의 흐름도에서 테스트하고자 하는 블록에 잠정적 단정을 참으로 설정하면, 테스트 데이터의 값의 범위를 좀 더 정확하게 알아내기 위해서 선후 지배관계를 이용하여 자동적으로 적절한 블록에 영속적 단정을 거짓으로 추가 설정한다. 이 두가지 종류의 단정문을 가지고 후방향 전달(backward propagation) 기법을 전방향 분석된 내용에 적용하면 단정문을 만족시키기 위한 조건이 나온다. 이 조건을 가지고 잠정적 단정을 참으로 설정한 블록을 커버하는 테스트 데이터를 자동으로 생성할 수 있다.

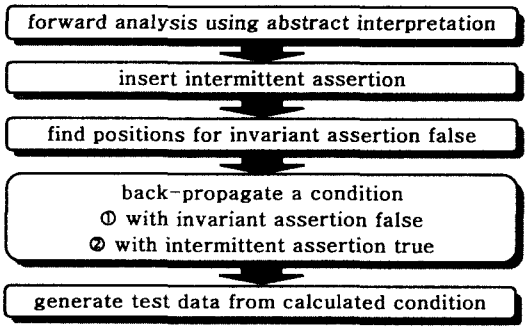

그림 1. 테스트 데이터 생성 과정

3.2 요약 해석을 이용한 프로그램 분석

요약 해석 기법은 앞에서 기술한 대로 프로그램의 동적 행위를 정적으로 분석하여서 정보를 얻을 수 있는 방법이다[3]. <그림 2> 에 있는 예제 코드를 이 방법으로 분석할 수 있다. <그림 3>은 예제 코드의 의미구조 등식의 전방향 체계(forward system of semantic equation)이다.

```
(0) read(i, j, k); (1)
if (i > j) then
  (2) max = i; (3)
else
  (4) max = j; (5)
end if; (6)
if (max < k)
  (7) max = k; (8)
-- else (9)
end if; (10)
print(max);
(11)
```

그림 2. 예제 코드

$x_0 = \top$
$x_1 = [\![\text{read(i,j,k)}]\!](x_0)$
$x_2 = [\![\text{i > j}]\!](x_1)$
$x_3 = [\![\text{max = i}]\!](x_2)$
$x_4 = [\![\text{i <= j}]\!](x_1)$
$x_5 = [\![\text{max = j}]\!](x_4)$
$x_6 = (x_3) \sqcup (x_5)$
$x_7 = [\![\text{max < k}]\!](x_6)$
$x_8 = [\![\text{max = k}]\!](x_7)$
$x_9 = [\![\text{max >= k}]\!](x_6)$
$x_{10} = (x_8) \sqcup (x_9)$
$x_{11} = [\![\text{print(max)}]\!](x_{10})$

그림 3. 전방향 의미구조 등식

<그림 3>에서 ⊔는 요약값을 합집합해주는 연산자이다. $[\![\text{i > j}]\!](x)$는 요약 테스트(abstract test)로 x위치의 프로그램 포인트(프로그램 상태)에서 반드시 만족해야 하는 조건을 나타낸다.

3.2. 테스트 블록 설정

프로그램을 분석하고 난 뒤, 사용자가 테스트하고자 하는 블록을 선택하여 잠정적 단정을 참으로 설정하면, 그 블록을 커버할 수 있는 테스트 데이터를 구하기 위한 조건을 계산해 낸다. 이 때, 결과로 나오는 조건을 보다 정확히 구하기 위해서, 영속적 단정을 거짓으로 설정할 블록을 자동으로 찾아낸다.

큰 프로그램에도 효율적으로 적용할 수 있기 위해서는 영속적 단정을 설정할 블록의 개수를 줄여야 하는데, 선후 지배관계를 이용하여 이를 해결한다. 우선 소스 프로그램에 대해 슈퍼 블록 선후 지배관계 트리를 만든다. 이 트리에서 잠정적 단정문을 참으로 설정한 블록과 선후 지배관계에 있지 않은 블록을 찾아서 모두 영속적 단정문 거짓을 삽입한다. 그러면, 단정문을 만족하는 조건을 보다 정확히 찾을 수 있다. 이렇게 영속적 단정문 거짓을 추가하는 이유는, 잠정적 단정문을 참으로 설정하는 것이 그 블록만이 아닌 다른 블록도 지나갈 수 있는 가능성을 가진 조건을 찾기 때문이다. 영속적 단정문 거짓을 관계없는 블록에 삽입하여 계산함으로써 조건의 범위를 어느 정도 줄여서 이런 가능성을 배제할 수 있다.

좀 더 정확한 범위를 찾아내고자 할 때에는 선후 지배관계에 있는 블록에 대해서도 영속적 단정문 거짓을 설정해 줄 수 있을 것이다. 한가지 방법으로 데이터 종속 관계를 따져볼 수 있다. 테스트하고자 하는 블록에서 사용하는 데이터에 대해, 선후 지배 관계에 있는 블록 중에서 정의(definition)나 사용(use)이 있는지 확인한다. 그 정의나 사용이 잠정적 단정을 참으로 설정한 조건을 위배하지 않는다면, 그 블록에 대해서도 영속적 단정을 거짓으로 설정해 줌으로써 더 좋은 품질의 테스트 데이터를 생성할 수 있다.

<그림 2>의 예제 코드에서 (7)을 지나는 테스트 데이터를 구하고자 할 때, 이 위치에 잠정적 단정을 참으로 설정한다. 그리고 나서, 선후 지배관계를 구하여, 선후 지배관계가 없는 위치를 찾도록 한다. 이 예제에서는 (9)의 위치가 (7)과 서로 지배관계가 없으므로 이 위치에 영속적 단정 거짓을 삽입한다. 또한, 선 지배관계에 있는 (2)나 (4)를 지나는 블록의 요약값이 (7)을 지나기 위한 서술조건(predicate)에 위배되지 않으므로 임의로 하나의 블록을 선택해서 영속적 단정을 거짓으로 설정할 수 있다.

3.3 단정문 만족 조건 계산

잠정적 단정문과 영속적 단정문을 다 설정하고 난 후에는, 요약 디버깅[5]에서 제시한 방법을 이용하여 설정한 단정문들을 만족시키는 조건을 계산한다. 조건 계산은 후방향 분석(backward analysis)을 통해 수행한다. 다음은 <그림 2> 예제 코드에 대한 의미구조 등식의 후방향 체계(backward system of semantic equation)이다.

예제 코드에서 (9)와 (4)의 위치에 설정한 영속적 단정을 거짓이 되게 하는 조건을 구하기 위해서는 x_4와 x_9에 $\alpha(\{false\})$라는 조건을 추가한다. 영속적 단정의 조건을 추가할 때는, 영속적 단정이 항상 만족해야 하므로 $\sqcap$연산자를 이용하여 추가한다. $\sqcap$는 요약값을 교집합해주는 연산자이다. $[\![max=i]\!]^{-1}$ 표시는 후방향 요약 할당문(backward abstract assignment)으로 전방향에서의 $[\![i=max]\!]$와 같다.

아래 식을 $\top$부터 시작하여 풀면 j<i<k라는 조건이 계산된다.

$x_0 = [\![read(i,j,k)]\!]^{-1}(x_1)$
$x_1 = [\![i>j]\!](x_2) \sqcup [\![i <=j]\!](x_4)$
$x_2 = [\![max=i]\!]^{-1}(x_3)$
$x_3 = (x_6)$
$x_4 = [\![max=j]\!]^{-1}(x_5) \sqcap \underline{\alpha(\{false\})}$
$x_5 = (x_6)$
$x_6 = [\![max<k]\!](x_7) \sqcup [\![max >= k]\!](x_9)$
$x_7 = [\![max=k]\!]^{-1}(x_8)$
$x_8 = (x_{10})$
$x_9 = (x_{10}) \sqcap \underline{\alpha(\{false\})}$
$x_{10} = [\![print(max)]\!]^{-1}(x_{11})$
$x_{11} = x_{11}$

그리고, (7)의 위치에 설정한 잠정적 단정이 참이 되도록 하는 조건을 구하려면, 아래의 등식에서 x_7에 $\alpha(\{true\})$를 추가한다. $\sqcup$는 요약값을 합집합해주는 연산자이다. 이 식을 $\bot$부터 시작하여 풀면 j<i<k 또는 i<=j<k라는 조건을 구할 수 있다.

$x_0 = [\![read(i,j,k)]\!]^{-1}(x_1)$
$x_1 = [\![i>j]\!](x_2) \sqcup [\![i <=j]\!](x_4)$
$x_2 = [\![max=i]\!]^{-1}(x_3)$
$x_3 = (x_6)$
$x_4 = [\![max=j]\!]^{-1}(x_5)$
$x_5 = (x_6)$
$x_6 = [\![max<k]\!](x_7) \sqcup [\![max >= k]\!](x_9)$
$x_7 = [\![max=k]\!]^{-1}(x_8) \sqcup \underline{\alpha(\{true\})}$
$x_8 = (x_{10})$
$x_9 = (x_{10})$
$x_{10} = [\![print(max)]\!]^{-1}(x_{11})$
$x_{11} = x_{11}$

이 두 조건을 합하면 j<i<k 라는 조건을 얻는다.

3.4 테스트 데이터 생성 및 실행

앞의 단계에서 구한 조건을 바탕으로 하여 테스트 데이터를 생성한다. 조건이 부분 범위의 요약값으로 계산되었을 때에는 무작위로 선택하거나 블랙 박스 테스팅 기법에서의 경계 조건(boundary condition)을 이용하는 방법과 비슷하게 테스트 데이터를 생성할 수 있다. 그리고, 예제에서처럼 입력데이터간의 조건식으로 나왔을 때에는 하나의 데이터를 기준으로 무작위로 실제값을 할당한 후 테스트하고자 하는 블록에 대해서만 요약해석 방법을 재적용하면 실제 타당한 테스트 데이터가 나온다. 예를 들어, 예제 프로그램의 j에 10을 할당하면 i는 [11, +∞], k는 [12, +∞]의 요약값을 구할 수 있다. 이 요약값과 앞에서 구한 조건식으로부터 (2)와 (7)을 지나는 (i, j, k)에 대한 테스트 데이터로 (11, 10, 12)를 사용할 수 있다.

지금까지 기술한 방법을 이용함으로써 테스트 데이터 생성까지의 과정을 모두 자동화함으로써 테스트를 훨씬 쉽고 효율적으로 수행할 수 있을 것이다.

4. 구현

본 논문에서 제시한 방법을 자바 언어로 구현하였다. 테스트 데이터 자동 생성을 위한 대상 언어는 자바이며 <그림 4>는 실제 구현한 프로토타입의 실행 모습이다.

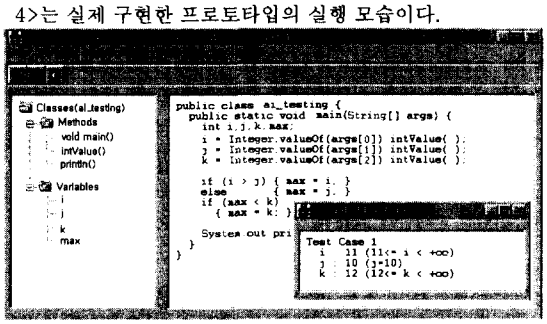

그림 4. 자바로 구현된 툴

5 결론 및 향후 연구방향

본 논문에서는 요약 해석 기법과 각 문장간의 선후 지배관계를 이용하여 테스트 데이터를 자동 생성하는 기법을 제시하였다.

향후에는 배열형도 지원할 것이고 프로시저간 레벨(interprocedural level)에서도 이 기법이 가능하도록 하는 방안을 연구할 예정이다. 또한, 계층성(hierarchy), 다형성(polymorphism) 등의 객체지향 언어의 특성에 맞는 테스트 데이터 자동 생성 기법에 대해서도 연구를 확장할 것이다. 마지막으로, 실험을 통해서 단정문 삽입에 대한 알고리즘과 테스트 데이터의 품질을 향상시킬 수 있는 방법을 모색할 것이다.

참고 문헌

[1] E. Weyuker, T. Goradia, A. Singh, "Automatically Generating Test Data from a Boolean Specification," IEEE Trans. on Software Eng., 1994
[2] M. Delamaro, J. Maldonado, A. P. Mathur, "Integration testing using interface mutations," Proc. of the 7th International Symp. on SRE, IEEE CSP, 1996
[3] P. Cousot, R. Cousot, "Static Determination of Dynamic Properties of Programs," Programmation. 1976
[4] P. Cousot, R. Cousot, "Abstract Interpretation : a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," Proc. of the 4th ACM Symp. on POPL, 1977.
[5] F. Bourdoncle, "Abstract debugging of higher order imperative languages," Proc. of the ACM SIGPLAN '93 Conf. on PLDI, 1993.
[6] H. Agrawal, "Dominators, Super Blocks, and Program Coverage," Proc. of the 21st ACM Symp. on POPL, 1994.