CUTIG: 정적 분석을 이용한 C언어 단위 테스트 데이타 추출 자동화 도구

(CUTIG: An Automated C Unit Test Data Generator Using Static Analysis)

김 택 수 † 박 복 남 †† 이 춘 우 †
(Taeksu Kim) (Boknam Park) (Chunwoo Lee)
김 기 문 † 서 윤 주 † 우 치 수 †††
(Kimoon Kim) (Yunju Seo) (Chisu Wu)

요 약 단위 시험은 지속적이고 반복적으로 수행되어야 하기 때문에 높은 비용을 필요로 하는 작업이다. 단위 시험의 자동화에 대한 많은 연구가 있었으나 테스트 데이타의 자동 추출에 대한 연구는 큰 성과를 이루지 못하고 있다. 본 연구에서는 소프트웨어의 소스 코드로부터 테스트 데이타를 자동으로 추출하는 방안에 대해 논의하고 각 단계의 알고리즘을 제시하였다. 또한 테스트 데이타 추출 자동화에 관한 이슈를 소개하고 테스트 데이타 추출 자동화 도구 CUTIG를 소개한다. CUTIG는 실제 소스코드를 이용하여 테스트 데이타를 추출하므로 소프트웨어의 요구사항 명세가 잘 작성되어 있지 않거나 실제 구현과 차이가 있는 경우에도 테스트 데이타를 생성할 수 있다. 또한 이 도구를 통해 개발자가 직접 테스트 데이타를 작성하는 데 소요되는 비용을 절감할 수 있기를 기대한다.

키워드 : 단위 시험, 테스트 자동화, 기호 실행

Abstract As unit testing should be performed repeatedly and continuously, it is a high-cost software development activity. Although there are many studies on unit test automation, there are less studies on automated test case generation which are worthy of note. In this paper, we discuss a study on automated test data generation from source codes and indicate algorithms for each stage. We also show some issues of test data generation and introduce an automated test data generating tool: CUTIG. As CUTIG generates test data not from require specifications but from source codes, software developers could generate test data when specifications are insufficient or discord with real implementation. Moreover we hope that the tool could help software developers to reduce cost for test data preparation.

Key words : Unit Test, Test Automation, Symbolic Execution

· 이 논문은 2008 한국소프트웨어공학학술대회에서 'CUTIG: 정적 분석을 이용한 단위 테스트 케이스 추출 자동화 도구'의 제목으로 발표된 논문을 확장한 것임

† 비 회 원 : 서울대학교 전기컴퓨터공학부
dolicoli@selab.snu.ac.kr
oniguni@selab.snu.ac.kr
gildream@selab.snu.ac.kr
iamonly@selab.snu.ac.kr
†† 정 회 원 : HandyPMG(핸디피엠지) 컨설팅 이사
bnpark@handypmg.co.kr
††† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
wuchisu@selab.snu.ac.kr

논문접수 : 2008년 4월 14일
심사완료 : 2008년 11월 12일

정보과학회논문지: 소프트웨어 및 응용 제36권 제1호(2009.1)

1. 서 론

단위 시험(unit test)은 단위 함수들 각각에 대해 지속적이고 반복적으로 수행하는 과정이기 때문에 많은 비용이 소모되는 작업이다. 단위 시험은 테스트 케이스(test case) 설계, 테스트 케이스 생성, 테스트 데이타 준비, 시험 수행, 결과 비교 및 결과 보고의 과정으로 이루어진다[1]. 이 과정에서 단위 시험의 비용을 줄이고자 하는 다양한 시도가 있었으며 자동화에 대한 연구 역시 활발히 이루어지고 있다.

테스트 자동화는 테스트 데이타 및 테스트 케이스 생성, 테스트 오라클(test oracle), 시뮬레이터(simulator), 프로그램 동적 분석(dynamic analysis), 테스트 결과 보고 등 다양한 영역에서 진행이 되고 있는데 이 중 시뮬레이터와 프로그램의 동적 분석, 결과 보고 등에서는 상당한 연구의 진척이 이루어졌으며 필요한 도구들 역시 다수 구현되어 있다. 하지만 상대적으로 테스트 데이타 및 테스트 케이스 생성에 대해서는 자동화 연구가 부족한 편이며 적용 가능한 도구 역시 많지 않다.

테스트 케이스는 기능별 요구사항 명세(functional requirement specification)로부터 생성되는 것이 일반적인데 자연어로 기술된 요구사항 명세의 경우 자연어의 특성으로 인해 테스트 케이스를 추출하는 것이 어려운 단점이 있다. 이를 보완하기 위해 정형화 기법을 이용해 명세를 작성하거나 설계 단계의 모델로부터 테스트 케이스를 추출을 하는 연구가 이루어지기는 했으나 사용이 어렵고 기술 습득에 드는 시간 및 비용의 이유로 현장에서 많이 활용되지 못하는 단점이 있다[2]. 또한 실제 프로그램을 구현하는 동안 요구사항의 변경이 일어나거나 소스 코드의 수정으로 인해 명세와 코드가 일치되지 않는 경우도 다수 존재하기 때문에 명세로부터 테스트 케이스를 자동적으로 추출하는 것은 쉽지 않다.

이와 같은 이유로 명세가 아니라 프로그램 소스 코드로부터 직접 테스트 케이스를 추출하고 테스트에 활용하는 방안에 대한 연구가 필요하다. 소스 코드로부터 테스트 케이스를 추출하는 경우 실제 수행 가능한 프로그램의 경로에 대해 테스트 케이스를 작성하게 되기 때문에 테스트 커버리지(test coverage)를 높일 수 있는 장점이 있다.

기호 실행(symbolic execution)을 활용해 테스트 케이스의 입력 값을 자동 생성하고자 하는 연구가 기존에 다수 존재하였다[3,4]. 기호 실행은 프로그램의 자유변수에 값이 아닌 기호를 대입하여 프로그램을 수행시킴으로써 하나의 경로를 수행하기 위한 자유 변수의 조건을 구하는 방법으로 소스 코드로부터 테스트 데이타를 추출하기에 적합한 기법이다. 하지만 기존의 연구에서는 구체적인 알고리즘의 제시나 구현 상의 쟁점에 대한 언급이 거의 없고 개념적인 내용의 제시에 그친 측면이 있어 실제 구현에 직접적으로 활용하기가 쉽지 않은 단점이 있다.

본 연구에서는 명세가 아닌 구현 소스 코드를 기반으로 테스트 케이스의 입력 값을 자동으로 추출하는 테스트 데이타 추출 자동화 방안에 대해서 연구하고 기호 실행을 활용해 소스 코드로부터 테스트 케이스의 입력 값을 추출하는 방법에 대해 논의한다. 또한 테스트 데이타 자동생성도구인 CUTIG(C Unit Test Input Generator)를 구현하였으며 각 단계의 알고리즘을 제시한다. CUTIG는 전처리된 C 소스 코드로부터 수행 가능한 경로를 추출하고 각 경로에 대해 기호 실행을 이용해 경로조건(path condition)을 생성하고 이를 만족하는 자유변수의 범위를 구함으로써, 테스트 케이스의 입력 값을 자동 생성한다.

본 논문은 다음과 같이 구성된다. 2장에서는 본 연구에서 논의하는 대상 언어 및 본 연구의 기본이 되는 기호 실행 등의 기본 개념을 기술한다. 3장에서는 기호 실행을 이용한 테스트 데이타 추출의 방법 및 알고리즘을 소개하며 4장에서는 테스트 케이스 입력 값 자동 생성 도구 CUTIG의 구조에 대해 설명한다. 그리고 실제 코드를 CUTIG에 적용한 사례를 5장에서 소개한다. 6장에서는 테스트 케이스 추출 자동화에 대한 기존의 연구를 살펴보고, 마지막으로 결론 및 향후 개선 방향을 제시한다.

2. 기본 개념

2.1 대상 언어

본 연구에서는 기호 실행을 이용하여 테스트 데이타를 추출한다. 이를 위해 대상 언어의 기능을 C언어의 일부로 한정하여 기호 실행을 적용하는데 대상 언어의 구문(syntax)을 그림 1과 같이 정의한다.

$$
\begin{array}{rcl}
C & \rightarrow & \texttt{skip} \\
 & | & x := E \mid *x := E \mid \&x := E \\
 & | & x.x := E \mid x \rightarrow x := E \mid x[n] := E \\
 & | & C \,;\, C \\
 & | & \texttt{if}\ E\ C\ C \\
 & | & \texttt{while}\ E\ C \\
E & \rightarrow & \texttt{readreal} \mid \texttt{r}\ (r \in \mathbb{R}) \\
 & | & E + E \mid E * E \\
 & | & -E \\
 & | & E.E \mid E \rightarrow E \\
 & | & E[E] \\
 & | & x \mid *x \mid \&x \\
 & | & E < E \\
 & | & E = E \\
 & | & E \,\&\, E \\
 & | & E \mid E
\end{array}
$$

그림 1 대상 언어의 구문

(1) 데이타 타입(Data type)

본 언어는 실수형 데이타 타입을 기본으로 한다. 이는 C 언어의 정수형, 부동 소수형을 모두 포함한다. 기호 실행의 경우 비트 연산을 포함하지 않기 때문에 실수형 자료를 비트에 따라서 구분하는 것은 큰 의미를 가지지 않기 때문에 문제를 단순화하기 위해 실수형 데이타 타입만을 고려하기로 한다.

또한 1차원 배열(1-dimension array)과 구조체(structure)를 지원한다. 구조체의 경우 C 언어는 중첩된 구조체(nested structure)가 언어의 구문에 포함되어 있으나 본 연구에서는 논의의 단순화를 위해 중첩된 구조체를 제외한다.

(2) 일반 연산자(Operator)

해당 언어는 사칙 연산을 위한 합, 곱, 음수 연산자를 지원한다. 또한 1차원 배열을 위한 배열의 인덱스 연산자를 지원하며 구조체를 지원하기 위한 구조체의 속성 연산자를 지원한다. C에서 지원하는 비트연산자의 경우 본 언어에서는 제외된다.

(3) 논리 연산자(Logical operator)

기호 실행은 경로조건을 판단하기 위해 논리 연산자를 필요로 한다. 본 언어에서는 대소 비교, 동등 연산자, 논리합과 논리곱 연산자를 포함한다.

(4) 명령문(Command)

본 언어는 순차적 실행(;), 분기문(if)과 반복문(while)을 지원한다.

2.2 기호 실행

기호 실행은 프로그램의 자유변수에 값이 아닌 기호를 대입하여 프로그램을 수행시킴으로써 하나의 경로를 수행하기 위한 자유 변수의 조건을 구하는 방법이다. 기호 실행의 각 과정에 대한 이해를 돕기 위해 그림 2의 소스 코드를 생각하기로 한다. 그림 2의 우측은 소스 코드의 제어흐름도(control flow graph)이다.

기호 실행을 활용한 자유 변수의 입력 조건을 추출하는 과정은 다음과 같다.

(1) 자유 변수를 기호로 대치한다.

프로그램의 수행 시 외부로부터 입력을 받는 자유 변수의 값을 모두 기호로 대치한다. 가능한 자유 변수로는 함수의 인자, 전역 변수 등이 있다. 그림 2의 예제의 경우 본 과정에서는 입력 값으로 지정된 자유 변수 a, b의 경우 각각 "a", "b"라는 기호로 대치된다.

자유 변수를 추출할 때 신경을 써야 할 내용은 수행하고자 하는 프로그램에서 외부 함수를 호출하는 경우인데 이 경우 두 가지의 선택이 가능하다. 첫 번째는 외부 함수를 대상 프로그램으로 포함시킴으로써 함수 호출부를 대상 프로그램에 포함시키는 것이다. 이 경우 외부 함수 호출부가 사라지기 때문에 기호 실행을 수행하는 데에는 아무런 문제가 없다. 두 번째 방법은 외부 함수의 결과물이 외부로부터 결정되기 때문에 자유 변수와 동등하게 취급하는 것이다. 이 경우 외부 함수의 호출 부는 자유 변수로 대치하여 수행이 가능하다.

```
1   int foo(int a, int b) {
2     int x, y;
3     x = a + 3;
4     y = b * 4;
5     z = b + 7;
6     if (x + y < z) {
7       z += x;
8     }else{
9       z -= y;
10    }
11
12    while (z < 20) {
13      z++;
14    }
15
16    return z;
17  }
```

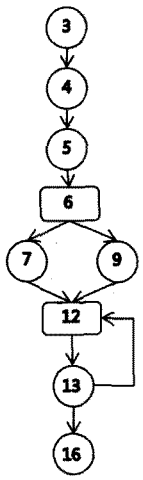

그림 2 예제 소스 코드 및 제어흐름도

(2) 프로그램의 수행 경로를 선택한다.

수행될 프로그램의 경로를 선택한다. 하나의 프로그램에서 수행될 수 있는 경로는 다양하다. 이는 분기문과 반복문의 영향인데 하나의 분기문은 가능한 경로를 두 배로 확장시키며 하나의 반복문은 가능한 경로를 무한대로 확장시킬 수 있다. 반복문의 종료 조건은 일반적으로 프로그램의 수행 시에 결정되기 때문에 정적 분석 단계에서는 가능한 경로는 무한대로 확장된다고 볼 수 있다.

가능한 경로 중 수행이 되어야 할 경로를 선택하는 것이 기호 실행의 두 번째 단계이다. 이 단계에서 모든 분기는 사라지며 대신 해당 경로를 수행하기 위한 조건으로 변경된다.

그림 2의 예제에서 소스 코드 i번이 수행되는 경우를 C_i, 조건 j번이 참이 되어야 하는 경우를 T_j, 조건 j번이 거짓이 되어야 하는 경우를 Fj로 표현하면 분기가 참일 경우, 반복을 세 번 수행하기 위한 경로는 {C_3, C_4, C_5, T_6, C_7, T_{12}, C_{13}, T_{12}, C_{13}, T_{12}, C_{13}, F_{12}, C_{16})이 된다.

(3) 경로의 수행 및 경로조건의 추출한다.

선택된 경로를 수행시키며 프로그램을 수행시킨다. 연산자가 사칙연산으로 이루어진 경우 이 과정을 거치면 프로그램의 변수들은 모두 자유 변수들의 합과 곱으로 표현이 가능하다.

수행 중 조건문의 경우 각 변수를 자유 변수들을 이용한 다항식으로 표현할 수 있으므로 자유 변수들의 부등식으로 표현이 되며 이를 경로조건으로 저장한다.

예를 들어 그림 2의 예제에서 C_5이 수행된 이후 변수 z의 값은 b+7이 되며 두 번 반복을 수행한 이후(T_{12}) 분기 조건은 -a+b+5<0이 된다.

(4) 경로조건을 풀이한다.

전체 수행이 완료된 후 모든 경로조건을 이용해 각 자유 변수의 범위를 풀이한다. 이 과정은 자유 변수들의 연립다항부등식을 푸는 문제로 치환될 수 있다. 그림 2 예제의 경우 최종적인 연립 부등식은 다음과 같다.

$$\begin{aligned} 5b-3 &< 0 \\ -a+b+3 &< 0 \\ -a+b+4 &< 0 \\ -a+b+5 &< 0 \\ -a+b+6 &\geq 0 \end{aligned}$$

본 연립부등식을 풀이하면 해는 $5<a\leq 6, b<\frac{3}{5}$이 된다.

연립부등식을 풀이한 결과 부등식이 풀이가 될 경우 대표 값을 선택하면 해당 경로를 수행할 수 있도록 하는 자유 변수의 조합이 결정된다. 만약 부등식이 해를 가지지 않는 경우 해당 경로는 수행 불가능한 경로가 된다.

3. 기호 실행을 이용한 테스트 데이타 추출의 자동화

2장에서 살펴보았듯이 기호 실행을 이용해 테스트 데이타를 추출하기 위해서는 자유 변수의 기호화, 경로의 추출, 경로의 수행 및 경로조건의 추출, 경로조건의 풀이의 과정을 거쳐야 한다. 자동화 도구의 구현을 위해서는 각 단계가 자동화되어야 하는데 기호 실행을 이용한 테스트 케이스 추출에 대한 기존의 연구에서는 모든 단계의 자동화에 대한 논의 보다는 경로의 수행 방법에 대한 개념적인 논의에 그친 측면이 많다. 이 장에서는 각 단계의 자동화를 위한 방법 및 알고리즘을 제시하고 쟁점들에 대해서 논의 한다.

3.1 자유 변수의 기호화

C 언어의 특징은 사용할 변수가 미리 선언이 되어야 한다는 점이다. 이 같은 특징으로 인해 기존에 존재하는 많은 C 언어 구문분석기(parser)를 활용하면 전역 변수, 함수의 인자, 지역 변수 등을 추출하는 것이 가능하다. 이 중 함수의 인자와 전역 변수의 경우 변수의 이름을 활용하여 기호화할 수 있기 때문에 자유 변수의 기호화는 간단하게 이루어질 수 있다.

본 과정에서 유의해야 할 사항은 함수 내부에서 재선언된 지역 변수의 경우인데 이는 전역 변수와 같은 이름을 가진 변수를 지역에서 재선언한 경우이다. 이 문제는 전처리기를 이용하여 재선언된 변수를 다른 변수명으로 변환하는 것으로 해결이 가능하며 실제로 많은 구문분석기가 이 방법을 활용하고 있다.

3.2 경로의 추출

테스트 데이타 추출의 자동화를 위해서는 소스 코드를 분석하여 가능한 경로를 자동으로 추출하기 위한 방법이 필요하다. 가능한 많은 경로를 추출하는 경우 그만큼 높은 테스트 커버리지를 확보할 수 있지만 너무 많은 경로를 선택하는 경우 테스트 수행에 드는 시간을 증가시킬 수 있기 때문에 프로그램으로부터 적절한 경로 집합을 추출하는 방법에 대한 자동화 알고리즘이 필요하다.

본 연구에서는 소스 코드로부터 경로를 추출하기 위해 다음의 조건을 만족시키는 최소의 경로를 추출하는 것을 목표로 한다.

- 모든 문장(statement)은 한번 이상 수행하여야 한다.
- 모든 분기문(branch)은 한번씩 수행하여야 한다.
- 모든 반복문(loop)은 0번 1번 2번 반복하여 수행하여야 한다.

경로란 시작과 끝을 가진 유한 길이의 명령문 연쇄(sequence)를 말한다. 예를 들어 그림 3의 최상단 소스 코드(a)는 분기문을 하나 가지게 되므로 두 개의 조건이 추출되게 되는데 각 조건은 그림 3의 (b), (c)와 같다.

경로에는 시작과 끝이 존재하기 때문에 경로에는 반복문이 포함되지 않으며 하나의 연쇄로 표현될 수 있어야 하기 때문에 분기문 역시 제거된다. 또한 경로에는 특정 지점에서 해당 경로를 수행하기 위한 조건이 추가되게 되는데 이를 조건(condition)이라 정의한다. 즉 경로는 명령문과 조건의 연쇄로 표현이 가능하다.

(a)
```
1 a = 3;
2 b = 7;
3 if (a < b) {
4   a++;
5 }else{
6   b++;
7 }
8 c = a + b;
```

(b)
```
1 a = 3;
2 b = 7;
3 Condition: (a < b);
4 a++;
5 c = a + b;
```

(c)
```
1 a = 3;
2 b = 7;
3 Condition: !(a < b);
4 b++;
5 c = a + b;
```

그림 3 소스 코드로부터의 경로 추출의 예

정의 1. 길이가 n인 경로 p의 i번째 항목을 c_i라 하자. 이 때 경로 p를 다음과 같이 표현한다.

$$p = [c_1; c_2; \cdots; c_n]$$

정의 2. 서로 다른 경로 p_1, p_2, … p_n을 원소로 가지는 집합 P를 경로 집합이라고 한다.

$$P = \{p_1, p_2, \cdots, p_n\}$$

소스 코드로부터 경로 집합을 추출하는 함수를 구현하기에 앞서 경로와 경로 집합에 대해 두 가지 연산(경로 합, 경로 곱)을 정의한다.

정의 3. 길이가 각각 m, n인 경로 $p_1=[c_1; c_2; \cdots, c_m]$, $p_2=[d_1; d_2; \cdots; d_n]$에 대해 경로 합 +은 다음과 같이 정의한다.

$$p_1 + p_2 = [c_1; c_2; \cdots; c_m; d_1; d_2; \cdots d_n]$$

경로 합은 두 경로를 순차적으로 수행할 경우의 최종적인 경로를 구하는 연산자이다.

정의 4. 두 경로 집합 $P_1=\{p_1, p_2, \cdots, p_m\}$과 $P_2=\{q_1, q_2, \cdots, q_n\}$에 대해 경로 곱 x은 다음과 같이 정의한다.

$$P_1 \times P_2 = \{ \quad p_1 + q_1, p_1 + q_2, \cdots, p_1 + q_n$$
$$p_2 + q_1, p_2 + q_2, \cdots, p_2 + q_n$$
$$\cdots$$
$$p_m + q_1, p_m + q_2, \cdots, p_m + q_n \quad \}$$

경로 곱은 두 경로 집합의 각 원소가 순차적으로 수행될 경우 가능한 전체의 경로를 원소를 가지는 새로운 집합을 정의하며 최종적인 집합의 원소의 개수는 각 경로 집합의 원소의 개수의 곱이 된다. 예를 들어 그림 2의 분기문을 거치게 되면 가능한 경로는 조건이 참일 경우와 거짓을 경우의 두 개의 경로를 만들어내며 반복문은 0번 수행, 1번 수행, 2번 수행의 세 개의 경로를 만들어낸다. 따라서 수행 가능한 총 경로는 6가지가 되는데 이 경우 분기문을 수행한 후의 경로 집합과 반복문을 수행한 후의 경로 집합의 경로 곱을 통해 전체 경로 집합을 구할 수 있다.

소스 코드로부터 경로 집합을 추출하는 get_path 함수는 명령문을 인자로 받아서 명령문으로부터 추출 가능한 모든 경로를 집합으로 반환하는 함수이므로 서명(signature)은 다음과 같다.

$$get\_path : Command \rightarrow Set\ of\ Path$$

get_path 함수는 명령문 중 분기문, 반복문과 그 외의 일반문의 세 가지 경우에 대해서 각각 다음과 같이 정의할 수 있다.

(1) 분기문

분기문의 경우 if E C_1 C_2의 구조를 가지게 된다. 먼저 조건이 참이 되는 경우의 경로 집합은 조건 Condition(E)를 원소로 가지는 경로 집합과 C_1에 대해서 추출 가능한 경로 집합을 경로곱 연산을 통해 구할 수 있다. 다음으로 조건이 거짓이 되는 경우의 경로 집합은 조건 Condition(!E)를 원소로 가지는 경로 집합과 C_2에 대해서 추출 가능한 경로 집합을 경로곱 연산을 통해 구할 수 있다. 이렇게 구한 두 경로 집합의 합집합이 조건문 if E C_1 C_2로부터 추출 가능한 경로 집합이다.

```
get_path C =
  match C with (if E C1 C2) ->
  let pathSet1 = {Condition(E)} X get_path(C1)
      and
      pathSet2 = {Condition(!E)} X get_path(C2)
  in
    pathSet1 U pathSet2
```

(2) 반복문

반복문의 경우 while E C_1의 형태로 표현된다. 본 연구에서는 모든 반복문의 경우 0번, 1번, 2번 반복하여 수행하는 경로에 대해서 추출하기 때문에 각각의 경우를 다음과 같이 조건문을 활용해 다시 쓸 수 있다.

```
0번: Condition(!E);skip
1번: Condition(E);C1;Condition(!E);skip
2번: Condition(E);C1;Condition(E);C1;
Condition(!E);skip
```

따라서 분기문의 경우 get_path 함수는 다음과 같이 구현 가능하다.

```
get_path C =
  match C with (while E C1) ->
  let pathSet1 = {Condition(!E)} and
      pathSet2 = {Condition(E)} X get_path(C) X
      {Condition(!E)} and
      pathSet3 = {Condition(E)} X get_path(C) X
      {Condition(E)} X get_path(C) X
      {Condition(!E)}
  in
    pathSet1 U pathSet2 U pathSet3
```

(3) 일반문

분기, 반복문이 아닌 일반문의 경우 기존의 경로에 해당 명령문을 추가한다.

```
get_path C = {C}
```

따라서 프로그램 소스 코드로부터 가능한 경로 집합을 구하는 extract_paths 함수의 전체 코드는 그림 4와 같다.

```
let get_path C =
   match C with
   (if E C1 C2) ->
   let pathSet1 = {Condition(E)} X get_path(C1) and
       pathSet2 = {Condition(!E)} X get_path(C2)
   in
      pathSet1 U pathSet2
  | (while E C1) ->
   let pathSet1 = {Condition(!E)} and
       pathSet2 = {Condition(E)} X get_path(C) X
  {Condition(!E)} and
       pathSet3 = {Condition(E)} X get_path(C) X
  {Condition(E)} X get_path(C) X {Condition(!E)}
   in
     pathSet1 U pathSet2 U pathSet3
  | _ -> {C}

let extract_paths P =
   foreach C in P
    U get_path C
```

그림 4 경로 집합 추출 함수

3.3 경로의 수행 및 경로조건의 추출

수행할 경로 집합이 정의되면 집합의 각 경로에 대해서 기호 실행을 통해 경로조건을 찾아내는 것이 테스트 데이타 추출의 다음 단계이다. 경로조건이란 경로를 수행하기 위해 필요한 조건을 자유 변수들의 관계로 표현한 것을 말하는데 결과는 자유 변수들의 다항 부등식의 형태로 표현된다. 예를 들어 그림 2의 예제에서 C_5이 수행된 이후 변수 z의 값은 b + 7이 되며 두 번 반복을 수행한 이후(T_{12}) 경로조건은 $-a + b + 5 < 0$이 된다.

경로조건 추출의 자동화를 구현하기 위해서는 프로그램 내에서 사용되는 변수들을 자유 변수들을 이용하여 표현하기 위한 인코딩(encoding) 방법과 변수들을 저장하기 위한 메모리를 정의해야 한다. 소스 코드의 자유 변수들을 $f_1, f_2, \cdots, f_n$이라고 할 경우 프로그램 내의 임의의 변수 v와 메모리 m은 다음과 같이 정의할 수 있다.

$$v \in Value = \sum_{i}^{n} c_i f_i^{m_i} \ (c_i . m_i \in \mathrm{N})$$

$$m \in Memory = Variable \rightarrow Value$$

또한 경로조건의 경우 v < 0 | v == 0 | !(v < 0) | !(v == 0)의 형태로 표현 가능하므로 경로조건 집합 pcs는 다음과 같이 정의 가능하다.

$$pcs = v < 0 | v == 0 | !(pcs) \ (v \in Value)$$

기호 실행을 이용하여 프로그램을 수행시킬 때 변수에 값을 지정하는 할당(assignment)문은 메모리를 변화시키며 조건문은 경로조건 집합을 변화시킨다. 그 외의 일반문의 경우 메모리와 경로조건 집합에 변화를 가하지 않기 때문에 변수간의 연산만 정의하면 충분하다. 본 연구에서 변수들은 자유변수의 다항식으로 정의되며 가능한 연산은 사칙 연산으로 제한되기 때문에 변수간의 사칙 연산은 수학적으로 정의된 연산을 활용한다.

경로로부터 경로조건 집합을 추출하는 함수 eval의 경우 서명은 다음과 같이 정의할 수 있다.

$$eval: \ path \rightarrow pcs$$

의사 코드는 그림 5와 같다. get_value 함수는 표현식을 메모리를 이용해 계산하는 함수이며 구현은 수학에서 정의된 다항식의 사칙 연산을 활용하면 구현할 수 있다.

3.4 경로 조건의 풀이

테스트 데이타 추출의 마지막 단계는 경로에 대해 추출된 경로조건 집합들을 이용해 각 자유 변수가 가질 수 있는 값을 선택하는 단계이다. 경로조건 집합은 자유 변수의 다항 부등식으로 표현될 수 있기 때문에 이 과정의 문제는 다항 부등식의 해를 푸는 문제로 치환될 수 있다. 그러나 다원 다차 연립 부등식의 일반 해를 구

```
evalSingleCommand (C, M, PCS) =
  match C with
    x := E -> let v = get_value (E) in (M/{x -> v}, PCS)
  | Condition(E) ->
    match E with
    E1 > 0 -> let v = get_value(E1) in (M, PCS U {Condition(v > 0)})
    | E1 == 0 -> let v = get_value(E1) in (M, PCS U {Condition(v == 0)})
    | !(E1 > 0) -> let v = get_value(E1) in (M, PCS U {Condition(!(v > 0))})
    | !(E1 == 0) -> let v = get_value(E1) in (M, PCS U {Condition(!(v == 0))})
    | _ -> (M, PCS)
eval path =
  let pcs = [] and memory = [] in
  match path with
  [] -> pcs
  | foreach c in path
      (memory, pcs) = evalSingleCommand (c, memory, pcs);
      Memory
```

그림 5 경로조건 추출 함수

하는 방법은 아직 발견되지 않았기 때문에 주어진 시간 내에 이 문제를 푸는 것은 쉽지 않은 일이다. 즉, 주어진 연립 부등식으로부터 미지수가 가질 수 있는 충분한 전 영역을 구해내는 것은 어렵다. 그러나 테스트 데이타의 경우 각 미지수의 정확한 영역을 계산하는 것이 아니라 영역에서 하나의 대표 값을 선택하는 것으로 충분하기 때문에 본 연구에서는 세 단계의 단순화된 부등식 풀이 알고리즘을 사용하여 자유변수 값을 계산한다.

1. 첫 번째 단계로 간단한 연산으로 구할 수 있는 미지수의 영역부터 구한다. 가장 간단한 형태의 부등식은 하나의 식으로 불능을 이끌어 내는 형태의 식이다. 이 경우는 경로조건이 해를 갖지 않기 때문에 해당 경로가 수행 불가능한 경로임을 의미하므로 해당 경로에 대한 테스트 케이스 입력 값을 추출하지 못하기 때문에 현재 계산되고 있는 경로에 대한 연산이 종료된다. 이를 위해 연립 부등식 가운데 상수항만 존재하는 부등식을 검색한 다음, 이 부등식이 참인지 거짓인지를 판별한다. 판별 값이 거짓일 경우 해당 부등식은 불능이므로 해당 경로에 대한 계산이 종료된다. 존재하는 모든 상수 항 부등식을 계산한 뒤 경로조건에서 부등식을 제거한다.

   다음으로 일원 일차 부등식을 계산하여 그 영역을 구한다. 일원 일차 부등식의 경우 둘 이상의 항이 불능인지 아닌지를 판단하는 것은 쉬운 일이므로 도출된 영역이 이전의 결과 값의 영역 내에 포함이 되는 것인지 파악한 뒤 그렇다면 남은 일원 일차 부등식을 계속 해서 계산하며 그렇지 않다면 해당 경로에 대한 계산을 종료한다.
2. 두 번째 단계는 부등식의 전체 차수를 낮춤으로써 문제의 복잡도를 단순화 하는 단계이다. 첫 번째 단계에서 구한 미지수에 대해 영역 내의 하나의 수를 대표 값으로 정한다. 그 다음 전체 연립 부등식에 대해 그 대표 값을 대입하여 전체 부등식의 차수를 낮춘다. 이렇게 차수가 낮아진 연립 부등식을 이용해 첫 번째 단계를 반복하여 다른 미지수에 대한 영역 값을 구한다.
3. 마지막 단계는 연산의 종료 단계이다. 위의 과정을 반복하는 도중 모든 미지수에 대해 영역 값을 구할 수 있게 되었다면 해당 경로로 수행하기 위한 테스트 케이스의 입력 값을 추출한 것이므로 결과를 반환하고 종료한다. 그렇지 않고 더 이상 미지수를 구할 수 없는 경우 해당 경로에 대한 연산이 종료된다.

그림 6은 세 개의 원소로 이루어진 경로조건 집합의 부등식 풀이 예이다. 이 경로를 수행하기 위한 첫 번째 단계로 상수 항 부등식을 검색한다. '1 > 0'이 검색이 되며 이 상수 부등식은 항상 참이므로 미지수와 관계없이 항상 성립한다.

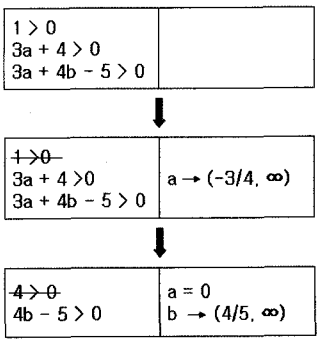

그림 6 부등식 풀이 예시

그 다음 차수가 낮은 1원 1차 부등식인 부등식 '3a + 4 > 0'이 검색되며 계산 결과 a는 -3/4보다 큰 값을 가져야 하며 a의 범위는 (-3/4, ∞)가 된다는 것을 알 수 있다. 그리고 더 이상 계산 가능한 부등식이 검색되지 않으므로 1단계를 종료한다.

두 번째 단계에서 첫 번째 단계에서 구해진 미지수 a의 범위 가운데 대표 값으로 0을 선택한다. 그리고 선택된 변수의 값을 경로조건 집합에 대입하여 전체 차수를 감소시킨다. '3a + 4 > 0'이던 부등식이 '4 > 0'으로 간소화되며 값은 항상 참이다. '3a + 4b - 5 > 0'은 '4b - 5 > 0'으로 변경되며 미지수 b의 범위는 (4/5, ∞)이다.

그리고 모든 미지수에 대해 범위를 구했으므로 전체 단계를 종료한다. 위의 과정을 통해 이 경로에 대한 테스트 케이스의 입력 값은 a=0, b=1이라는 것을 알 수 있다.

4. CUTIG

본 연구에서는 3장에서 논의한 테스트 데이타 추출의 알고리즘을 활용하여 테스트 데이타 추출 자동화 도구인 CUTIG를 구현하였다. CUTIG는 OCAML[5] 프로그래밍 언어로 구현되었으며 전처리된 C소스 파일을 분석하고 제어흐름도를 추출하기 위해서 CIL[6] 라이브러리를 사용하여서 구현되었다. 그림 7은 CUTIG의 구조를 보여준다.

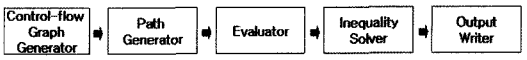

그림 7 CUTIG의 구조

CUTIG는 5개의 모듈로 구성되며 각 모듈은 다음과 같은 역할을 한다.

• 제어 흐름도 생성기(control-flow graph generator)

자유 변수를 확인하고 프로그램의 흐름을 확인하기

위해 전처리를 거친 소스 코드를 분석하여 제어흐름도를 생성한다.

- 경로 생성기(path generator)
  제어흐름도의 노드를 방문하면서 수행 가능한 경로를 생성한다.
- 경로조건 생성기(evaluator)
  경로를 만족하는 자유변수의 부등식을 생성한다.
- 부등식 풀이기(inequality solver)
  생성된 부등식을 푼다.
- 결과 생성기(output generator)
  부등식의 풀이 결과를 텍스트 파일로 출력한다.

CIL 라이브러리의 경우 전처리된 C 소스 파일을 입력으로 받아서 다른 형태로 변화시켜주는 중간단계 언어를 정의하고 있다. 이를 활용하면 C 소스 파일을 의미가 동일한 C++ 소스 파일이나 Java 파일로 변환하는 것이 가능하며 소스 코드의 제어흐름을 파악할 수 있는 제어흐름도로 변환하는 것 역시 가능하다.

CUTIG의 제어 흐름도 생성기 모듈에서는 CIL을 활용해 제어흐름도를 메모리에 저장한 후 경로 생성기 모듈에서 각 실행 라인을 분석하여 경로 집합을 추출하도록 하였다. 이 과정에서 앞서 논의했던 변수의 재정의나 외부 함수 호출의 경우를 수정하고 그 결과를 경로 조건 생성기 모듈로 전달하여 기호 실행을 통해 경로조건을 추출하였다. 부등식 풀이기 모듈은 경로조건 집합을 활용해 연립 부등식을 풀이하여 자유 변수의 대표 값을 선택하는 모듈이다. 마지막으로 결과 생성기 모듈은 가능한 모든 해 중 불능인 경우를 제외한 나머지 결과들을 텍스트 파일로 출력해 테스트 케이스 입력 값을 텍스트의 형태로 출력하는 모듈이다.

5. 적용사례

본 연구는 가전 제품에 포함되는 내장형 시스템을 개발하고 있는 업체의 단위 시험에 활용되기 위한 목적으로 수행되었다. 실제 개발된 소스 코드 중 적용 가능한 코드를 대상으로 CUTIG를 활용하여 테스트 케이스의 입력 값을 추출해 보았다.

그림 8은 스캔 모드와 현재 채널을 입력으로 받아 채널 검색을 하는 프로그램 코드의 일부이다. 입력 모드에 따라 전체 스캔을 하거나 일부에 대해서만 스캔을 하도록 분기가 주어지며 각 분기 내에서는 해당 채널을 반복적으로 검색하게 된다. 본 코드를 이용해 제어흐름도를 그리면 그림 9와 같다.

제어흐름도에 의하면 본 프로그램의 순환복잡도(cyclomatic complexity)는 5가 되어 크게 복잡하지 않은 단일 함수임을 알 수 있다. 하지만 기준을 만족하는 테스트 데이타의 개수는 총 15가지가 된다. 이처럼 분기

```
1   #define FULL_SCAN           0
2   #define UBOUND              25
3   #define TV_CHANNEL          13
4   #define MAX_CHANNEL         150
5
6   void scan(int mode, int c) {
7     int c_ch;
8     int v, offset, length;
9
10    c_ch = c;
11    if (mode > FULL_SCAN) {
12      v = c_ch + 10;
13      while (c_ch < UBOUND) {
14        storech(c_ch);
15        c_ch *= 2;
16      }
17    }else{
18      v = c_ch - TV_CHANNEL;
19      if (c_ch <= MAX_CHANNEL) {
20    c_ch = 2 * c_ch - 5;
21      }else{
22    c_ch = 3 * c_ch;
23      }
24    storech(c_ch);
25    }
26    while (v < c_ch + 10) {
27    put_to_mem(v);
28      v += 25;
29    }
30  }
```

그림 8 채널 검색 프로그램 코드

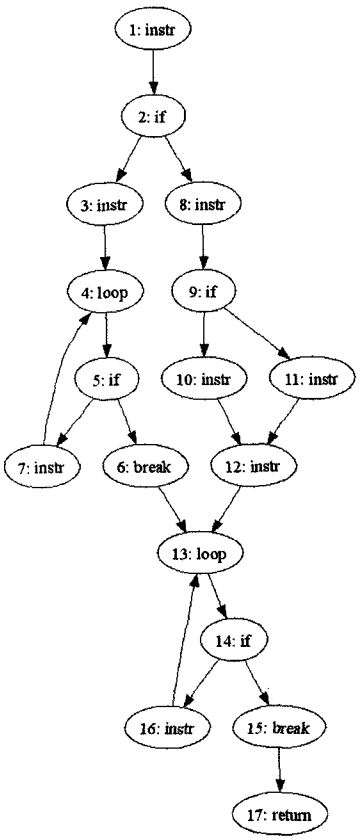

그림 9 채널 검색 프로그램의 제어흐름도

표 1 CUTIG에 의해 추출된 테스트 데이타

순번	변수	값의 범위	입력값
1	mode	(-∞, 0]	-1
	c	[7, 7]	7
2	mode	(-∞, 0]	-1
	c	[-18, 18]	-18
3	mode	(-∞, 0]	-1
	c	(-∞, 0]	-1
4	mode	(0, ∞)	1
	c	[7, 7]	7
5	mode	(0, ∞)	1
	c	[13, 13]	13
6	mode	(0, ∞)	1
	c	(-∞, ∞)	0
7	mode	(0, ∞)	1
	c	(25, ∞)	26
8	mode	(0, ∞)	1
	c	(-∞, ∞)	0

와 반복이 복합되어 있는 프로그램의 경우 가능한 테스트 데이타의 수가 급격히 증가되기 때문에 테스트 케이스를 작성하는데 드는 비용 역시 그에 따라서 커지게 됨을 알 수 있다.

CUTIG를 이용해 추출한 테스트 데이타는 표 1과 같다. 표에서 나타난 테스트 케이스의 입력 값은 총 8개인데 이는 가능한 총 15개의 경로 중 실제로 수행이 불가능한 경우가 존재하기 때문이다. 예를 들어 26번 라인의 반복 조건의 경우 18번 라인을 거칠 경우 실제의 v 값은 언제나 c_ch + 10 보다 적은 값이 되기 때문에 반복 수행을 0번 하는 것은 불가능하므로 해당 경우에 대한 2 종류의 경로는 수행이 불가능하다. CUTIG를 활용하면 이와 같이 불가능한 경로에 대해서 테스트 케이스를 작성하기 위한 시도에 드는 노력을 줄일 수 있는 부가적인 효과가 있다.

6. 관련 연구

테스트 케이스 생성에 관한 연구는 랜덤 테스트 케이스를 생성하는 접근법으로부터 시작되어 프로그램의 모델을 기반으로 하는 방식으로 발전되어 왔다. 이 방법들은 서로 영향을 끼치며 동시에 적용되기도 한다.

테스트 생성 연구의 초기부터 발생한 랜덤 테스트 생성은 테스트 입력을 무작위로 생성하기 때문에 만들기는 쉬우나 결과값의 수준이 대체로 낮은 편이다. 하지만 최근의 연구들을 보면 기존의 모델 기반 테스트 생성과 결합하여 그 가능성을 높이고 있다.

Sen은 "concolic testing"이라는 DART 접근법을 개발하였다[7]. 이는 기호 실행과 랜덤 테스트를 결합한 방식으로서 랜덤 테스트 생성과 기호 실행이 동시에 수행하면서 서로 도움을 준다. 이런 방법은 어느 정도 실행 가능한 테스트 입력을 생성하지만 조건을 풀어내는 과정이 임의의 수를 생성하는데 의지하기 때문에 부정확할 수 있다.

임의로 생성한 기존의 테스트 결과를 피드백(feedback)하는 방식도 존재한다[8]. 이는 피드백한 결과를 조건 또는 필터로써 임의의 테스트 생성에 도움을 주는 방식이다. 이런 방식 또한 Sen의 연구처럼 부정확하거나 너무 많은 수의 테스트 입력이 생성될 수 있다.

모델 기반 테스트 생성은 프로그램의 모델로부터 테스트를 생성하는 방법으로 그 역사가 매우 깊다. Moore는 이미 1956년도에 유한상태기계(finite-state machine) 기반의 테스트 생성을 연구하기 시작했다[9]. 모델 기반 테스트에는 다양한 모델을 사용할 수 있고 이를 통해서 나온 조건을 만족하는 입력 값을 찾는다.

기호 실행은 오래 전부터 모델 기반 테스트 생성에 사용되어 왔다[10]. 이 기법은 테스트 생성의 조건을 찾는데 유리하지만, 조건에 맞는 테스트 입력 값을 찾는 능력에 따라 전체적인 성능이 결정된다. 따라서 최근에는 모델 검증(model checking)과 같은 기술과 결합하여 연구가 진행되고 있다[11]. 그러나 모델 검증은 상태 폭발(state explosion)의 가능성이 있기 때문에 최상의 성능을 내는 조건을 찾는 것이 필요하다.

7. 결론 및 향후 연구

기호 실행을 활용한 단위 시험의 테스트 데이타의 자동 추출에 대한 기존의 연구는 많은 경우 실제 구현에 대한 알고리즘의 제시가 부족하고 개념적인 소개에 그친 측면이 있다. 본 연구에서는 단위 시험 케이스 생성과 테스트 실행의 자동화를 위한 방법을 연구하였으며 그 알고리즘을 제시하였다. 또한 이를 바탕으로 자동화 도구인 CUTIG를 구현하였다. CUTIG는 테스트하고자 하는 대상 코드를 기호 실행을 사용하여 경로를 추출하고 경로를 수행하기 위한 자유 변수들의 조건을 경로조건으로 변환하여 입력 가능한 변수의 값을 선택한다.

CUTIG를 활용하면 테스트 케이스의 입력 값을 자동으로 추출할 수 있으므로 테스트에 소요되는 비용을 줄일 수 있으며 코드 기반으로 제약사항내의 가능한 경로를 추출하여 테스트 커버리지를 높일 수 있었다. 또한 수행 불가능한 경로에 대한 테스트 입력 값을 배제할 수 있으므로 테스트 입력 값을 추출하는데 소요되는 시간을 보다 감소시킬 수 있는 이점도 있었다.

하지만 CUTIG는 몇 가지 한계를 가지고 있는데 그 첫 번째가 연산자의 제약이다. 기호 실행을 활용하기 위해서 CUTIG는 C언어에서 사용 가능한 모든 비트연산자(bit operator)를 배제하고 있다. 실제 비트연산은 C 소

스 코드에서 상당히 많이 사용되고 있는 연산자이기 때문에 CUTIG를 바로 다양한 C 소스 코드에 적용하는데 큰 제약으로 작용한다. 이를 보완하기 위한 방법으로는 다항식으로 변수의 값을 표현하고 있는 현재의 인코딩을 비트를 활용하여 표현하고 각 연산자에 대해 수행의 방법을 재정의하는 등의 작업이 동반되어야 할 것이다.

두 번째 CUTIG의 한계는 포인터 변수의 부정확성이다. 실제 포인터 변수는 C에서 정수형과 동일하게 취급이 되고 있기 때문에 CUTIG에서는 포인터 변수를 정수형 변수로 인지하여 수행이 된다. 하지만 실제 포인터는 수행 시(runtime)에 의미 있는 값을 나타내기 때문에 정적 분석을 이용한 테스트 데이타 추출의 근원적인 한계라 할 수 있다.

세 번째 한계는 CUTIG의 경우 반복문을 0 번, 1 번, 2 번 수행하는 경우에 대해서 경로 조건을 추출하고 있는데 보다 의미 있는 테스트 데이타가 되기 위해서는 반복문의 최대 반복 회수만큼 수행하는 경로를 추출하는 것이 필요하다. 이 최대 반복의 회수는 많은 경우 프로그램의 실제 수행 시에 결정되는 것이 대다수이기 때문에 역시 정적 분석만으로는 한계가 존재할 수 있다. 하지만 정적 분석을 통해 반복의 최대 값을 찾을 수 있는 경우에 대해서만이라도 최대 반복 회수를 활용하여 경로를 추출하는 과정이 필요할 것으로 판단되며 향후 연구를 통해 지속적으로 보완되어야 할 부분이다.

CUTIG에서 사용하는 부등식 풀이기의 정확도 역시 지속적으로 향상이 가능한 부분이다. 앞서 언급한 바와 같이 연립 다항 부등식의 일반 해를 찾는 방법은 아직 발견되어 있지 않기 때문에 가능한 보다 많은 경우에 대해서 적용 가능한 부등식 풀이 방법은 계속적인 연구를 통해 보완해 나갈 여지가 있는 부분이다. 향후에는 이러한 CUTIG의 한계를 보완하여 보다 정확하고 넓은 테스트 커버리지를 확보할 수 있는 테스트 데이타 추출 자동화 도구를 개발하고자 한다.

마지막으로, 현재의 사례 연구는 국내 전자 회사의 디지털 TV 소스 코드 중 상호 합의하에 선별된 소스 코드를 변경하여 수행한 결과를 제시한 수준에 그쳐있기 때문에 도구의 효용성을 검증하기에 부족한 단점이 있다. 연구의 타당성을 보다 확보하기 위하여 CUTIG를 이용하여 추출된 데이타를 통해 실제 테스트 오류를 찾아낸 빈도 및 기존의 테스트 데이타 추출 도구와 비교하여 테스트 커버리지 및 테스트 데이타의 정확도가 향상되었음을 검증하는 실험이 필요하지만 본문에 기술한 대로 현재로서는 CUTIG 프로그램의 많은 한계로 인해 포인터의 활용 빈도가 상대적으로 높은 실제 프로그램의 소스에 직접 적용하기가 무리가 있었다. 향후에는 추가적인 연구를 통해 CUTIG 프로그램을 개량한 후 많은 사례에 적용하여 실제 오류를 찾고 검사하는 데 유용하게 활용될 수 있음을 검증하는 작업을 진행하고자 한다.

참 고 문 헌

[1] I. Sommerville, "Software Engineering," 8th Edition, Addison-Wesley, 2007.

[2] B. Blanc, et. al., "Automated functional test case generation from data flow specifications using structural coverage criteria," European Congress ERTS 2006, 2006.

[3] V. Rusu, L. du Bousquet, and T. Jéron, "An approach to symbolic test generation," Proceedings of the Second International Conference on Integrated Formal Methods, pp. 338-357, 2000.

[4] C. Bigot, et. al., "Automatic Test Generation with AGATHA," Lecture Notes in Computer Science, Vol. 2619, pp. 591-596, 2003.

[5] K. Sen, D. Marinov, and G. Agha, "CUTE A concolic unit testing engine for C," Joint meeting of the European Soft. Eng. Conf. and ACM SIGSOFT Intl. Symp. on Foundations of Soft. Eng. (ESEC/FSE'05), pp. 263-272, 2005.

[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," Technical Report MSR-TR-2006-125, Microsoft Research, Redmond, 2006.

[7] E. F. Moore, "Gedanken-experiments on sequential machines," Automata Studies, pp. 129-153, 1956.

[8] J. C. King, "Symbolic execution and program testing," Communications of the ACM, Vol.19, pp. 385-394, 1976.

[9] W. Visser, C. S. pasareanu, and S. Huurshid, "Test Input Generation with Java PathFinder," ACM SIGSOFT Intl. Symp. on Software Testing and Analysis(ISSTA 2004), pp. 97-107, 2004.

[10] OCAML, Objective Caml, http://caml.inria.fr/.

[11] CIL, C Intermediate Language, http://www.cs.berkeley.edu/~necula/cil/.

김 택 수

2005년 서울대학교 수리과학부 졸업(학사). 2005년~현재 서울대학교 대학원 컴퓨터공학부 석박사통합과정(박사과정). 관심분야는 소프트웨어 테스팅, 테스트 자동화, 소프트웨어 아키텍처 역공학

박 복 남

1989년 서울대학교 공학사. 2000년 서강대학교 공학석사. 2001년 Carnegie Mellon Univ. MSE 축약 과정 이수. 1988년~1999년 SI/SM프로젝트 SW개발, 방법론, SEPG/SQA. 1999년~2000년 한국전산원 감리. 2000년~2002년 품질경영업

무(CMM/ISO9001/ SPICE). 2002년~2007년 삼성전자/LG전자 책임연구원. 2008년~핸디피엠지. 2000년~CMM/SPICE 심사원. 2003년~CSQE(ASQ). 2004년~한국정보과학회 SW공학연구회 산학위원. 2008년~한국SW프로세스품질인증심사원. 관심분야는 Embedded/IT서비스/SW/R&D 분야 Process 개선, SE, 방법론, SQA, Testing

이 춘 우

2002년 서울대학교 컴퓨터공학부 졸업(학사). 2002년~현재 서울대학교 대학원 컴퓨터공학부 석박사통합과정(박사과정) 관심분야는 소프트웨어 테스팅, 요구공학, 소프트웨어 재사용

김 기 문

2006년 서울대학교 컴퓨터공학부 졸업(학사). 2008년 서울대학교 대학원 컴퓨터공학부 졸업(석사). 현재 삼성전자 정보통신총괄 통신연구소 소프트웨어센터 연구원. 관심분야는 소프트웨어 테스팅, 적응형 소프트웨어

서 윤 주

2007년 중앙대학교 컴퓨터공학부 졸업(학사). 2007년~현재 서울대학교 대학원 컴퓨터공학부 석사과정. 관심분야는 임베디드 소프트웨어 테스팅

우 치 수

1965년~1972년 서울대학교 응용수학과 졸업(학사). 1972년~1974년 한국과학기술원 연구원. 1975년~1977년 서울대학교 대학원 전산과학과 졸업(석사). 1977년~1982년 서울대학교 대학원 전산과학과 졸업(박사). 1978년 영국 라퍼러 대학 연구원. 1975년~1982년 울산대학교 전자계산학과 조교수, 부교수. 1985~1986년 미국 미시간대학교 postdoc. 1988년~1990년 한국정보과학회 총무 이사. 1990년~1992년 한국정보과학회 학회지 편집위원장. 1992년~1993년 한국정보과학회 부회장. 1996년~1998년 한국정보과학회 소프트웨어공학연구회 운영위원장. 1982년~현재 서울대학교 컴퓨터공학부 교수