자바 원시 코드에서 논리적인 오류를 찾는 복합 디버깅 기술의 설계

Design of Hybrid Debugging Technique for Locating Logical Errors in Java Source Codes

고훈준
경인여자대학 컴퓨터정보학부

Hoon-Joon Kouh(hjkouh@kic.ac.kr)

요약

이전 논문은 자바 프로그램에서 논리적인 오류를 찾기 위한 기술로 HDT를 제안했었다. HDT는 알고리즘믹 프로그램 디버깅 기술을 이용하여 실행트리에서 오류를 포함하고 있는 메소드를 찾고, 단계적 프로그램 디버깅 기술을 이용하여 오류를 포함하고 있는 메소드에서 오류가 있는 문장을 찾아낸다. 이 기술은 전통적인 디버깅 기술보다 자바 프로그램에서 프로그래머가 디버깅하는 횟수를 줄였다. 그러나 최근에는 프로그램 크기가 증가하고 메소드의 수가 증가하고 있기 때문에 아직까지 HDT는 디버깅하는 횟수가 많다. 이 논문은 HDT에 프로그램 분할 기술을 적용하는 HDTS를 제안한다. 이 방법은 자바 프로그램을 디버깅할 때, HDT보다 프로그래머가 디버깅하는 횟수를 줄여 줄 수 있다. 특히, HDTS는 메소드와 문장의 수가 증가할수록 효율성이 증가한다.

■ 중심어 : | HDT | HDTS | 프로그램 분할 |

Abstract

In the previous work, we presented HDT for locating logical errors in Java programs. The HDT locates an erroneous method at an execution tree using an algorithmic program debugging technique and locates a statement with errors in the erroneous method using a step-wise program debugging. It reduced the number of programmer debugging in Java programs. But the HDT still increases the number of debugging because the size of the recent programs increases than the past programs and the number of methods is increasing. This paper proposes HDTS using a program slicing technique (PST) at the HDT. HDTS can reduce the number of programmer debugging. Specially, the more the number of methods and statements increases, the more HDTS has effects.

■ Keyword : | Hybrid Debugging Technique | Hybrid Debugging Technique with Slicing | Program Slicing |

I. 서 론

최근 소프트웨어는 과거에 비해서 점점 더 복잡해지고 많은 신뢰성을 요구하면서 프로그램 검증기술이 부각되고 있다. 일반적으로 프로그램 검증이란 프로그램이 실행 중에 오류를 발생하여 예기치 않은 동작을 하는 것을 찾아내고 더불어 수정까지 하는 것을 말한다. 오류를 가지고 있는 소프트웨어는 자동차, 비행기, 엘리베이

* 본 논문은 2005년도 정부재원(교육인적자원부 학술연구조성사업비)으로 한국학술진흥재단의 지원을 받아 연구되었습니다.(KRF-2005-003-D00339)

접수번호 : #060731-001
접수일자 : 2006년 07월 31일
심사완료일 : 2006년 09월 15일
교신저자 : 고훈준, e-mail : hjkouh@kic.ac.kr

터 등 많은 부분에서 인간에게 해를 끼칠 수 있기 때문에 오류가 없는 소프트웨어를 개발하거나 개발된 소프트웨어에서 오류를 찾아내는 기술들의 관심이 높아지고 있다.

최근에 많은 연구가 되고 있는 검증 기술은 프로그램 분석, 모델 검증, 자동 증명 방법을 사용하거나 이 세 가지를 총 동원하여 프로그램이 실행 중에 발생할 수 있는 실행시간 오류(runtime errors)를 발견하고 있다. 프로그램 분석이란 주어진 프로그램을 요약하여 프로그램이 가지는 성질을 예측하는 기술이고, 모델 검증은 검증하려는 프로그램이 가질 수 있는 모든 상태를 효과적으로 검색하여 명세를 만족하지 않는 경우가 발생하는지를 찾는 기술이다. 그리고 자동증명 방법은 수학의 명세 중에서 참인 것을 자동으로 증명하는 기술이다. 그러나 아직까지 논리적인 오류(logical errors)는 실행 결과에 대해서 프로그래머가 올바른 결과인지 아닌지를 결정해야 하기 때문에 이 기술들을 이용한 자동화분석은 어렵다. 따라서 프로그램의 논리적인 오류에 대한 분석은 실행시간 오류에 비해 많은 시간과 노력을 요구한다. 그 결과, 원시 프로그램의 크기가 증가하고 복잡해질수록 프로그램 내에 포함된 논리적인 오류를 발견하고 수정하는 것은 매우 어렵다. 특히, 자바와 같은 객체지향 프로그램은 객체지향의 특징들 때문에 절차지향 프로그램보다 디버깅 작업이 더욱 어렵다[1].

최근까지 논리적인 오류를 찾고 수정하는 방법은 테스팅과 디버깅을 사용하고 있다. 테스팅이란 프로그램에서 오류가 발생하는 입력 값을 발견하는 과정을 말한다. 디버깅이란 프로그램 테스트의 결과로부터 프로그램 내에 오류의 원인을 발견하고 수정하는 과정을 말한다. 일반적인 디버깅 기술은 프로그래머가 프로그램 실행 후에 실행 결과를 기반으로 오류의 원인이 되는 변수를 찾아서 프로그램을 역추적하거나 단계적 프로그램 디버깅 기술(step-wise program debugging technique)[6]를 이용하여 프로그램을 처음부터 단계적으로 실행하면서 논리적인 오류를 포함하고 있는 원시 프로그램을 분석하는 방법이다. 이 방법은 현재 대부분의 상용 컴파일러 시스템에서 이용되고 있는 방법이지만 최악의 경우 프로그래머가 프로그램의 실행순서에 따른 모든 문장을 한 문장씩 실행해야 하는 문제가 있다. 프로그램의 크기가 커지고 복잡해짐에 따라 프로그래머가 논리적인 오류를 발견하기 위해 디버깅에 참여하는 많은 횟수를 요구하는 디버깅 시간은 기하급수적으로 증가한다[3].

[1][8]에서는 객체지향의 개념을 가지고 있는 자바 프로그램에서 논리적인 오류를 찾는 디버깅 시간을 감소시키기 위해 Shapiro[10]에 의해 제안된 알고리즈믹 프로그램 디버깅 기술(algorithmic program debugging technique)[1][7]을 적용하였다. 그러나 알고리즈믹 프로그램 디버깅 기술은 오류가 포함된 메소드를 탐색하는 방법만 고려하였고, 오류가 포함된 문장까지 찾는 것은 고려하지 않았다. 또한 하나의 오류를 탐색하기 위한 방법만을 제시하였고 여러 개의 논리적인 오류를 찾는 것은 고려하지 않았다. [2][9]에서는 알고리즈믹 프로그램 디버깅 기술과 단계적 프로그램 디버깅 기술을 혼합하여 사용하는 HDT(Hybrid Debugging Technique)를 제안하였다. HDT는 알고리즈믹 프로그램 디버깅 기술을 이용하여 실행트리에서 오류가 포함된 메소드를 찾고, 단계적 프로그램 디버깅 기술을 이용하여 그 오류가 포함된 메소드 내에서 오류를 가진 문장을 찾는 방법이다. 이 방법은 실행 트리에서 오류가 포함된 노드를 발견하고 그 노드에 대응하는 메소드 내의 문장들을 디버깅해서 오류를 발견하기 때문에 디버거 사용자는 오류가 없는 노드의 문장들은 디버깅할 필요가 없다. 그 결과, 기존 단계적 프로그램 디버깅 기술보다 자바 프로그램에서 프로그래머가 디버깅하는 횟수를 감소시켰다. 그러나 여러 개의 논리적인 오류를 발견하고 오류가 모두 수정되었는지 확인하기 위해서는 디버깅 시스템이 오류가 없다는 메시지를 보낼 때까지 실행 트리를 매번 재생성 해야 하고 프로그래머는 이전에 correct라고 응답한 노드까지도 매번 응답해야 한다. 또한 불필요한 노드와 문장을 디버깅해야 하는 문제가 있다. 이 문제는 프로그래머가 디버깅하는 횟수를 증가시킨다.

본 논문에서는 자바 프로그램에서 논리적인 오류를 발견하는 HDT를 확장한 HDTS(Hybrid Debugging Technique with Slicing)를 제안한다. 이 디버깅 기술은 기존의 전통적인 단계적 프로그램 디버깅 기술과 알고리즈믹 프로그램 디버깅 기술을 혼합해서 프로그램 내

에 포함된 논리적인 오류를 발견하는 HDT에 프로그램 분할 기술(program slicing technique)[11]을 적용하여 프로그래머가 디버깅하는 횟수를 줄일 수 있는 방법이다. 이 패러다임의 중요성은 자바 프로그램에서 논리적인 오류를 발견하는 자동화 디버깅을 위한 시도이다.

II. 관련연구

이 장은 기존의 자바 프로그램에서 논리적인 오류를 발견하는 HDT의 기본 개념과 디버깅 기술을 서술하고, 프로그램의 코드의 양을 줄이는 분할 기술을 서술한다.

1. 실행트리

실행 트리(execution tree)는 프로그램을 실행하면서 메소드 단위로 실행에 대한 정보를 포함하고 있는 노드를 구성하고 프로그램 내의 모든 메소드의 호출관계를 나타낸 트리 구조로 $T=(N,E,S)$ 에 의해서 정의된다. 이때 N은 유한개의 노드 n들의 집합이고 E는 N에서 노드들의 관계를 나타내는 순서쌍인 간선의 집합이다. 간선 $(n_i, n_j) \in E$는 노드 n_i와 노드 n_j가 호출 관계임을 나타낸다. S는 $S \in N$을 만족하는 시작 노드이며 일반적으로 트리의 루트 노드가 된다. N에 포함된 노드 n는 $n=(o, c, n, in, out)$ 와 같이 정의될 수 있다. 이때 o는 객체 이름, c는 클래스 이름, m은 메소드 이름이다. in은 입력 변수들의 집합으로 하나의 입력 변수는 (v_i, x_i)로 표현된다. out은 출력 변수들의 집합으로 하나의 출력 변수는 (v_o, x_o)로 표현된다. 이때 v_i는 입력 변수 이름이고, v_0는 출력 변수 이름이다. 그리고 x_i와 x_0은 그 변수들의 값이다. 메소드 m는 입력 in에 의해 실행되고 출력 out을 반환한다. 따라서 N은 $\{(o_1, c_1, m_1, in_1, out_1), \cdots, (o_k, c_k, m_k, in_k, out_k)\}$와 같이 유한한 순서의 집합으로 정의될 수 있다. 메소드 m_1은 입력 in_1에 의해 실행되어 출력 out_1을 반환하고, 메소드 m_k은 입력 in_k에 의해 실행되고 출력 out_k을 반환한다. 그리고 $0 \leq i \leq j \leq k$와 같은 조건을 만족하는 $(o_i, c_i, m_i, in_i, out_i)$와 $(o_j, c_j, m_j, in_j, out_j)$을 각각 n_i와 n_j라고 가정하고 m_i에서 m_j을 호출한다면, n_i는 부모 노드, n_j는 자식 노드가 된다. 두 노드 사이의 간선은 $(n_i, n_j) \in E$로 표현될 수 있다. 실행 트리 T는 시작 노드 S로부터 단말 노드(terminal node)까지 왼쪽에서부터 오른쪽으로 메소드의 호출 순서에 따라 구성된다[9].

2. HDT

HDT는 알고리즈믹 프로그램 디버깅 기술과 단계적 프로그램 디버깅 기술을 혼합하여 사용하는 방법으로 프로그램에서 발생될 수 있는 논리적인 오류를 발견하고 수정할 수 있는 방법으로 다음 5단계에 의해 디버깅을 진행한다.

1 단계는 원시 프로그램을 실행하고, 각 메소드의 입력 값과 출력 값 등의 정보로부터 노드를 구성하고 메소드의 호출 순서에 따라 실행 트리를 생성한다. 2 단계는 프로그래머는 실행 트리의 루트 노드부터 탐색을 시작하거나 임의의 노드 하나를 선택함으로써 디버깅을 시작할 수 있다. 3 단계는 탐색은 실행 트리를 하위 레벨로부터 상위 레벨로 왼쪽에서부터 오른쪽으로 전위 탐색 방법으로 각각의 노드를 탐색하면서 오류가 포함된 노드를 발견한다. 각각의 노드를 탐색할 때, 프로그래머는 단지 메소드의 입력 값과 출력 값에 대해서 correct 또는 incorrect로 노드의 입력과 출력 정보가 올바른지 아닌지를 결정함으로써, 디버깅 시스템은 프로그래머의 응답으로부터 논리적인 오류가 포함된 메소드를 자동으로 발견한다. 4 단계는 오류가 포함된 노드가 발견되면 디버깅 시스템은 그 노드에 해당하는 메소드 내의 문장을 step-over, stop, break-point, go 명령어를 사용해서 디버깅한다. 프로그래머는 그 노드의 입력 값에 대한 정보로부터 노드에 대응하는 메소드 내의 모든 문장들을 단계적으로 실행하고, 의심되는 변수를 확인하면서 논리적인 오류가 포함되어 있는 문장을 발견하고 수정한다. 마지막으로 5단계는 더 이상 오류가 존재하지 않을 때까지 1단계부터 4단계까지 반복한다[2][9].

이 방법은 논리적인 오류가 포함된 메소드를 찾을 뿐 아니라 오류가 포함된 문장까지 연속해서 찾을 수 있다. 그리고 더 이상 오류가 없을 때까지 여러 개의 오류를 연속해서 발견할 수 있다[2]. 그러나 HDT는 실행 트리를

한번 생성하면 하나의 오류 밖에 발견할 수 없다. 따라서 여러 개의 오류를 발견하고 오류가 모두 올바르게 수정되었는지 확인하기 위해서는 디버깅 시스템이 오류가 없다는 메시지를 보낼 때까지 실행 트리를 여러 번 재 생성해야 하고 프로그래머는 이전에 correct라고 응답한 노드까지도 매번 응답을 해야 하는 문제가 발생된다.

3. 프로그램 분할 기술

프로그램 분할은 프로그램의 제어흐름과 자료흐름에 기반을 둔 기술로 M.Weiser[12]에 의해 처음으로 제안된 기술이다. 이 후 Horwitz와 Ottenstein에 의해 발전하였다. 이 기술은 프로그램의 이해와 분석, 프로그램의 유지보수와 복잡성 측정 그리고 프로그램을 디버깅할 때 검사해야 하는 프로그램 코드의 양을 줄이기 위한 방법 등으로 사용되고 있으며 현재 수많은 분야에서 사용되고 있는 기술이다. 분할의 기본 개념은 프로그램의 어떤 위치에서 특정한 변수의 값에 직접 또는 간접적으로 영향을 주는 문장들의 집합으로 프로그램의 위치 p와 변수 v에 관련된 분할은 p에서 변수 v에 영향을 주는 프로그램의 모든 문장들과 술어들로 구성된다. 따라서 분할은 프로그램에서 0개 이상의 문장을 제거함으로써 얻을 수 있는 실행할 수 있는 프로그램으로 정의될 수 있다[4].

분할은 일반적으로 정적 분할(static slicing)과 동적 분할(dynamic slicing)로 구분할 수 있으며, 정적 분할은 초기의 Wesier와 Horwitz에 의해 제안된 분할 방법으로 프로그램의 입력에 따른 가정 없이 프로그램의 임의의 위치와 변수들로 분할 기준으로 계산되는 개념이다. Weiser는 이 분할 기준에 대하여 자료흐름 방정식을 정의하여 간접적으로 관련이 있는 문장들의 집합으로 분할을 하였다. 그러나 이 방법은 프로시저 사이의 분할에서 정확한 결과를 얻지 못하였다. 그래서 Horwitz[5]는 프로그램의 종속성 그래프를 이용하여 프로시저 사이의 종속성을 표현하는 시스템 종속성 그래프를 제시하고 이를 이용하여 그래프 도달 가능성 문제로 프로시저 사이의 분할을 구하였다. 동적 분할은 Korel에 의해 처음 소개된 방법으로 프로그램 실행 시 주어진 프로그램 입력에 대하여 변수 값에 실제로 영향을 미치는 모든 문장으로 구성하는 방법이다. 이 분할은 프로그램 종속성 그래프를 이용한 동적 종속성 그래프(dynamic dependence graph)로부터 구한다[11].

III. 자바 프로그램을 위한 HDTS의 설계

본 장은 HDT의 방법을 확장시킨 HDTS를 제안하고 HDTS를 적용하기 위한 확장 실행트리를 제안한다.

1. HDTS

HDTS는 프로그래머가 HDT를 사용해서 프로그램을 디버깅할 때 프로그램 분할 기술을 사용하여 디버깅하는 횟수를 줄이는 방법이다. 본 논문에서 사용하는 프로그램 분할은 순수 프로그램 분할의 개념을 확장하여 실행트리에서 오류를 포함하고 있는 메소드 내의 문장 중에서 디버깅에 관련이 없는 문장과 디버깅에 불필요한 노드를 제거하여 노드 수를 줄이는 모든 방법을 포함한다.

본 논문에서 분할은 다음 네 가지 경우에 사용된다. 첫 번째는 프로그래머가 incorrect라고 응답하고, 오류를 포함하는 출력 변수 하나를 선택하면, 디버깅 시스템은 그 오류에 종속성이 존재하지 않는 자식 노드들을 제거한다. 그리고 프로그래머는 디버깅 시스템으로부터 그 자식 노드들에 대한 질문을 받지 않는다(Slicing1). 두 번째는 만약 프로그래머가 실행 트리에서 오류를 포함하고 있는 노드(메소드)를 발견하면, 디버깅 시스템은 오류가 포함되어 있는 노드에 해당하는 메소드 내에서 그 오류를 포함하는 변수를 분할 기준으로 그 변수와 관련이 없는 문장들을 정적 분할한다. 그러면 프로그래머가 단계적 프로그램 디버깅을 할 때 디버깅하는 횟수를 줄일 수 있다. 이 경우는 Weiser의 정적 분할의 개념을 그대로 사용할 수 있다(Slicing2). 세 번째는 오류를 포함하고 있는 문장을 발견하고 수정한 후에 다시 실행 트리의 노드를 탐색할 때 디버깅 시스템은 아직 탐색하지 않은 노드들 중에서 오류를 포함하는 변수와 종속성이 존재하는 노드들을 제거한다. 그리고 실행 트리의 마지막 노드까지 탐색을 계속한다(Slicing3). 네 번째는 실행트리를 다시 생성할 때 이전 실행 트리의 탐색에서 프로

그래머가 correct라고 응답한 노드와 그 자식 노드들을 제거한다(Slicing4).

HDTS에서 Slicing1, Slicing3, Slicing4의 경우는 실행 트리를 디버깅할 때 불필요한 노드들을 제거시키는 방법이고, Slicing2의 경우는 오류가 포함된 메소드의 원시 프로그램에서 불필요한 문장들을 제거시키는 방법이다. HDTS의 알고리즘은 [알고리즘 1]과 같다.

알고리즘 1. HDTS: HDT with Slicing의 알고리즘

```
============================================
Input: Incorrect Program
1 procedure HDTS(P)
2 begin
//let   t={(o_1,c_1,m_1,in_1,out_1),…,(o_n,c_n,m_n,in_n,out_n)}
be the top level trace nodes of (o_0, c_0, m_0, in_0, out_0)
3 repeat
4   Build an execution tree from P
5   Slicing4
6   n := (o_0, c_0, m_0, in_0, out_0) //select a start node
7   debug := False
8   i := 1
9   if(Query( (o_0, c_0, m_0, in_0, out_0)) = correct) then
10     debug := True
11  else
12    if ∃t then
13      Slicing1
14      while i<n do
15     if(Query( (o_i, c_i, m_i, in_i, out_i)) = correct) then
//let (o_j, c_j, m_j, in_j, out_j) be the right sibling node
of (o_i, c_i, m_i, in_i, out_i)
16          if (o_j, c_j, m_j, in_j, out_j)<> NIL then
17            i := j
18            continue
19          else
20            Slicing2
21            Statement_debug(the parent node of
              (o_i, c_i, m_i, in_i, out_i))
22            Slicing3
23          fi
24        fi
25        i := i + 1
26      od
27      Slicing2
28      Statement_debug( (o_i, c_i, m_i, in_i, out_i))
29    else
30      Slicing2
31      Statement_debug( (o_0, c_0, m_0, in_0, out_0))
32    fi
33  fi
34 until (debug = True)
35 write("There are no errors")
36 end
============================================
```

[알고리즘 1]에서 t는 프로그래머가 실행 트리를 탐색하기 위해 시작 단계에서 선택한 노드의 자식 노드들이고 실행 트리의 탐색은 마지막 노드까지 탐색하여 여러 개의 오류를 발견할 수 있으며, 오류가 없을 때까지 실행 트리를 재 생성하여 탐색이 가능하다. 변수 debug의 값이 False인 경우에는 아직 실행 트리에 오류가 있는 노드가 존재한다는 것을 의미하고 True인 경우에는 더 이상 오류가 없음을 의미한다. 실행 트리의 탐색에서 오류가 포함된 함수가 발견되면 정적 분할인 Slicing2를 수행한 후에 함수 Statement_debug를 호출하여 단계적 프로그램 디버깅 기술을 사용한다. 프로그래머는 step-over, stop 명령어와 break-point, go 명령어를 이용하여 오류가 포함된 함수 내의 문장들을 실행하고 제어하면서 오류가 있는 문장을 발견한다.

2. 확장 실행트리

HDTS의 프로그램 분할은 크게 두 가지 측면에서 사용될 수 있다. 첫째는 정적 분할 기술을 사용하여 메소드 내의 문장들을 분할하는 것과 둘째는 실행 트리에서 동적 분할 기술을 사용하는 노드의 분할이다. 정적 분할은 Weiser의 방법을 사용한다. 실행 트리에서 노드의 동적 분할을 수행하기 위해서는 Horwitz의 시스템 종속성 그래프 *SDG*와 linkage grammar[5]를 기반으로 기존 실행 트리에서 각 노드의 입력 변수와 출력 변수 사이의 종속성 관계와 노드와 노드 사이의 종속성 관계를 추가한 확장 실행 트리를 생성한다. 그리고 그래프 도달 가능성 문제[5]를 이용하여 노드와 노드 사이의 분할을 수행한다. 따라서 실행 트리 $G=(N, E, S)$ 에서 간선의 집합 E를 다음과 같이 확장한다.

$$E = E_1 \cup E_2 \cup E_3$$

E_1: 노드 간에 제어 종속 관계를 나타내는 간선의 집합이다.

E_2: 각 노드에서 입력 변수와 출력 변수 사이의 데이터 종속 관계를 나타내는 간선의 집합이다. 데이터 종속 관계는 내부 데이터 종속 관계와 외부 데이터 종속 관계가 있다. 내부 데이터 종속 관계(internal data dependency)는 하나의 노드에서 입력 변수와 출력 변수와의 관계를 나타내는 간

선의 집합이다. 외부 데이터 종속 관계(external data dependency)는 부모와 자식 관계 또는 형제 관계에 있는 노드들 사이의 입력과 출력 변수 사이에 관계를 나타내는 간선의 집합이다.

E_3: 오류를 포함하고 있는 노드의 클래스 이름 c와 메소드 이름 m이 동일한 노드들의 출력 변수 사이의 관계를 나타내는 간선의 집합이다.

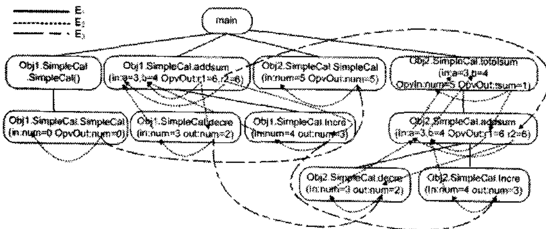


그림 1. [그림 2]로부터 생성된 확장 실행트리

[그림 1]은 [그림 2]의 예제 프로그램으로부터 생성된 확장 실행 트리로써 기존 실행 트리보다 두 가지의 간선 E_2와 E_3가 추가된 트리이다. 확장 실행 트리에서 분할은 그래프 도달 가능성 문제를 이용하여 Slicing1과 Slicing3을 수행한다.

IV. HDTS를 이용한 자바 프로그램의 디버깅

본 장은 [그림 2]의 프로그램을 HDTS를 이용한 디버깅 방법으로 설명한다. 프로그래머가 프로그램 디버깅을 위해서 [그림 2]의 자바 프로그램을 실행하면 [그림 1]과 같은 확장 실행 트리가 생성되고, 프로그래머는 실행 트리의 노드 하나를 선택함으로써 디버깅을 시작한다.

1 단계: 알고리즈믹 프로그램 디버깅

일반적인 디버깅의 시작은 실행 트리의 루트 노드로부터 탐색을 시작한다. 디버깅 시스템의 질문에 대해서 프로그래머는 correct 또는 incorrect로 응답할 수 있고, 만약 프로그래머가 incorrect로 응답했을 때 추가로 오류의 원인이 되는 출력 변수를 지정할 수 있다.

```
class SimpleCal {
  int num, r1, r2, tsum;
  SimpleCal() {
      this(0);
  }
  SimpleCal(int num) {
      this.num = num;
  }
  int decre(int num) {
      num = num - 1;
      return num;
  }
  int incre(int num) {
     num = num - 1;
     return num;
  }
 int addsum(int a, int b) {
     r1 = decre(a) + b
     r2 = incre(b) + a
  }
  int totalsum(int a, int b) {
       int r3, r4
       addsum(a,b);
       r3 = r1 + 1;
       r4 = r2 + r1;
       tsum = r1 - num;
  }
}
public class Calculation {
   public static void main(String args[]) {
     SimpleCal Obj1 = new SimpleCal();
     Obj1.addsum(3,4);
     SimpleCal Obj2 = new SimpleCal(5);
     Obj2.totalsum(3,4);
     System.out.println(Obj1.r1+ "  " + Obj1.r2);
     System.out.println(Obj2.tsum)
   }
 }
```

그림 2. 자바 프로그램의 예

Obj1.SimpleCal.SimpleCal(in:num=0 OpvOut:num=0)? $ correct
Obj1.SimpleCal.addsum(in:a=3, b=4 OpvOut:r1=5, r2=6)? $ incorrect, r2

프로그래머는 addsum 메소드에 대해서 incorrect라고 응답하였고 출력 변수 r1와 r2중 오류의 원인이 되는 출력 변수 r2를 선택하였다.

2 단계: 프로그램 분할 (Slicing1)

프로그래머가 응답한 오류가 있는 변수 r2를 분할 기준으로 현재 노드의 자식 노드에서 분할 기준 변수 r2와

종속성이 없는 노드를 모두 제거한다. 그 결과 [그림 1]에서 변수 r2와 데이터 종속성이 없는 노드 Obj1.SimpleCal.decre(in:num=3 out:num=2)는 [그림 3]과 같이 제거된다.

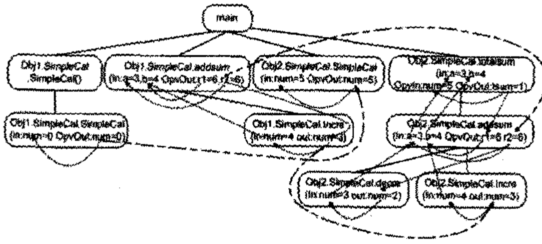


그림 3. 오류의 원인이 되는 변수 r2와 종속성이 없는 자식 노드가 제거된 트리

3단계: 알고리즈믹 프로그램 디버깅

2단계에서 노드 Obj1.SimpleCal.decre(in:num=3 out:num=2)가 실행 트리에서 제거되었기 때문에 디버깅 시스템은 그 다음 노드에 대해 프로그래머에게 질문을 한다.

Obj1.SimpleCal.incre(in:num=4 out:num=3)?$ incorrect

디버깅 시스템은 프로그래머의 응답 결과로부터 메소드 incre에 논리적인 오류가 있음을 자동으로 발견한다.

4단계: 프로그램 분할 (Slicing2)

오류가 포함된 메소드 incre의 출력 변수 num을 분할 기준으로 정적 분할을 한다. 그러나 메소드 incre는 분할될 문장이 없다.

5단계: 단계적 프로그램 디버깅

프로그래머는 텍스트 편집기에서 메소드 incre의 소스 프로그램을 보면서 단계적 프로그램 디버깅을 시작한다. 프로그래머는 step-over 또는 break-point, go 명령을 이용하여 문장을 탐색한다.

num(3)=num(4)-1 ? $ stop

프로그래머는 문장 num=num-1이 오류가 있음을 발견할 수 있고 문장 num=num+1로 수정할 수 있다. 그리고 단계적 프로그램 디버깅은 프로그래머가 stop 명령을 하거나 메소드의 문장을 모두 실행했을 때 종료된다.

6 단계: 프로그램 분할 (Slicing3)

디버깅 시스템은 실행 트리의 다음 노드를 탐색하기 전에 아직 탐색하지 않은 노드들 중에서 오류의 원인이 되는 변수 num과 종속성이 있는 노드와 오류를 포함하고 있는 메소드와 클래스 이름과 메소드 이름이 동일한 노드를 모두 제거한다. 따라서 노드 Obj1.SimpleCal.incre(in:num=4 out:num=3)에서 간선 E_2을 이용해서 변수 num과 종속성이 있는 노드를 모두 제거하고 간선 E_3을 이용해서 클래스이름과 메소드 이름이 동일한 노드를 제거한다. 그 결과 실행 트리에서 [그림 4]와 같이 오른쪽 네 개의 노드가 제거된다.

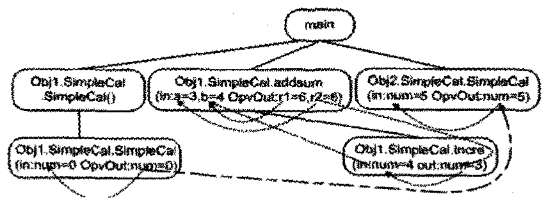


그림 4. 오류의 원인이 되는 변수 num과 종속성이 존재하는 모든 노드가 제거된 실행 트리

7 단계: 알고리즈믹 프로그램 디버깅

디버깅 시스템은 다음 노드를 탐색한다.

Obj2.SimpleCal.SimpleCal(in:num=5 OpvOut:num=5)? $ correct

탐색하지 않은 노드가 더 이상 존재하지 않으므로 디버깅 시스템은 [그림 5]와 같이 실행 트리를 다시 생성한다.

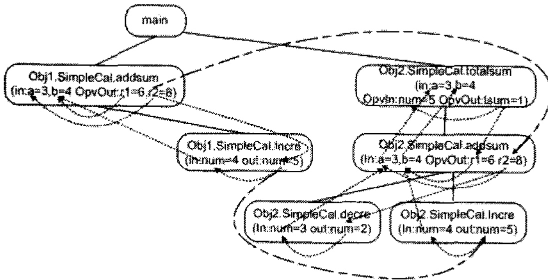


그림 5. 첫 번째 오류 수정 후 생성된 재실행 트리

8단계: 프로그램 분할 (Slicing4)

실행 트리를 다시 생성할 때는 전 단계까지 프로그래머가 correct라고 응답한 올바른 노드는 모두 제거된다. 따라서 재실행 트리를 탐색할 때 다음 두 노드는 프로그래머에게 질문하지 않는다.

Obj2.SimpleCal.SimpleCal(in:num=5 OpvOut:num=5)

Obj1.SimpleCal.SimpleCal(in:num=0 OpvOut:num=0)

9단계: 알고리즈믹 프로그램 디버깅

디버깅 시스템은 다시 생성한 실행 트리를 탐색한다.

Obj1.SimpleCal.addsum(in:a=3,b=4 OpvOut:r1=6,r2=8)? $ correct

Obj2.SimpleCal.totalsum(in:a=3,b=4 OpvIn:num=5 OpvOut:tsum=1)? $ incorrect

Obj2.SimpleCal.addsum(in:a=3,b=4 OpvOut:r1=6,r2=8)? $ correct

이 결과로부터 디버깅 시스템은 메소드 totalsum에 오류가 있음을 발견한다.

10단계: 프로그램 분할 (Slicing2)

메소드 totalsum의 출력 변수 tsum을 분할 기준 변수로 하여 프로그램을 정적 분할한다. 그 결과 문장 r3=r1+1과 r4=r2+r1이 삭제된다.

11단계: 단계적 프로그램 디버깅

프로그래머는 텍스트 에디터 창에서 메소드 totalsum의 원시 프로그램을 보여주면서 단계적 프로그램 디버깅을 시작한다.

addsum(a(3),b(4))? $ step-over

tsum(1) = r1(6) num(5)? $ stop

오류가 포함된 문장 tsum=r1-num을 발견할 수 있고, 프로그래머는 오류가 포함된 문장을 문장 tsum=r1+num으로 수정할 수 있다. 디버깅 시스템은 탐색하지 않은 노드가 더 이상 없으므로 다시 실행 트리를 생성한다.

12단계: 프로그램 분할 (Slicing4)

실행 트리를 재 생성할 때 이전 단계에서 프로그래머가 correct라고 응답한 오류가 없는 노드들은 제거된다. 그 결과는 [그림 6]과 같다.

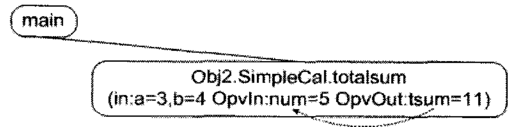


그림 6. 두 번째 오류 수정 후 생성된 재실행 트리

13단계: 알고리즈믹 프로그램 디버깅

디버깅 시스템은 수정된 자바 프로그램으로부터 생성된 [그림 6]의 실행 트리를 탐색한다.

Obj2.SimpleCal.totalsum(in:a=3, b=4 OpvIn:num=5 OpvOut:tsum=11)? $ correct

프로그래머는 디버깅 시스템의 질문에 대해서 correct라고 응답을 했고 더 이상의 노드가 없기 때문에 디버깅 시스템은 프로그램에 대해 더 이상 오류가 없다고 인식하고 프로그램 디버깅을 종료한다. 그 결과 프로그래머는 [그림 2]의 자바 프로그램으로부터 두 개의 논리적인 오류를 발견하였다.

HDTS는 HDT를 확장하여 네 가지의 경우에 프로그램 분할 기술을 적용하였고, 디버깅에 불필요한 노드 또

는 문장을 제거할 수 있었다. 그리고 실행 트리를 한번 생성하여 여러 개의 논리적인 오류를 찾을 수 있고 HDT보다 실행 트리를 재 생성하는 횟수를 줄일 수 있다. 또한, 프로그래머가 디버깅에 참여하는 횟수를 HDT보다 더 줄일 수 있다.

V. 비교 및 평가

본 논문에서 평가 기준은 프로그래머가 디버깅 시스템에 접근하는 횟수로 하였다. 그리고 기존의 단계적 프로그램 디버깅 기술 두 가지와 본 논문에서 제시한 HDT와 분할을 적용한 HDTS를 비교하였다. 단계적 프로그램 디버깅 기술로 프로그램을 디버깅하는 것은 프로그래머마다 다양하겠지만 본 논문에서는 두 가지 방법으로 가정하여 평가하였다.

첫 번째 단계적 프로그램 디버깅 기술(SWPD1: step-wise program debugging 1)은 main 메소드에서 step-over를 이용하여 호출 메소드의 반환 값을 확인하면서 오류가 포함된 메소드를 찾는다. 그리고 break-point를 오류가 포함된 메소드에 설정하고, 명령어 go로 디버깅을 다시 시작해서 그 메소드까지 이동한 후 그 메소드 내에서 step-over로 오류를 발견하거나 호출 메소드에 오류가 있을 경우에는 앞의 순서를 반복해서 논리적인 오류를 발견한다. 그리고 프로그램을 실행해서 프로그램이 오류가 없는지 확인한다. 만약 오류가 여러 개 존재할 경우에는 이러한 순서를 반복 실행하여 오류를 발견한다. 예를 들어, [그림 2]의 자바 프로그램인 경우 디버깅을 시작해서 Obj1.SimpleCal.addsum(3,4); 문장을 실행하면 메소드 addsum내에 오류가 있음을 알 수 있다. 그러나 현재 방법은 Obj1.SimpleCal.addsum(3,4); 문장을 이미 실행했기 때문에 메소드 addsum내의 문장을 디버깅하기 위해서는 디버깅을 다시 시작해야 한다. 그래서 메소드 addsum내의 문장들을 디버깅하기 위해 break-point와 명령어 go를 사용하여 이동하고, step-over 명령어를 이용해서 디버깅해야 한다.

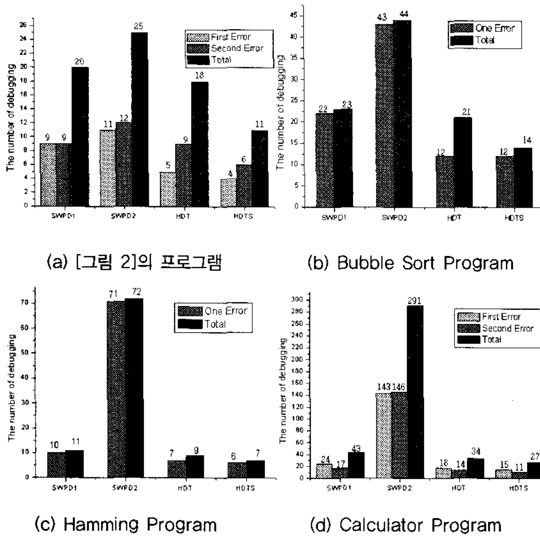
두 번째 단계적 프로그램 디버깅 기술(SWPD2: step-wise program debugging 2)은 처음부터 실행되는 경로의 모든 문장을 실행하면서 오류를 발견한다. 오류를 수정한 후 프로그램을 실행하여 프로그램에 오류가 있는지 확인한다. 오류가 여러 개 존재할 경우에는 이러한 순서를 반복 실행해서 오류를 발견한다.

본 논문에서 평가를 위한 예제 프로그램은 [표 1]의 네 가지 프로그램을 사용하였다.

표 1. 평가를 위한 예제 프로그램

	문장수	메소드 수	논리적인 오류 수	실행 트리의 노드 수
〈그림 2〉	29	7	2	11
Bubble Sort	35	6	1	17
Hamming	43	6	1	9
Calculator	111	10	2	29

[표 1]에서 '문장 수'는 각 예제 프로그램의 모든 문장의 수를 나타내고, '메소드 수'는 각 예제 프로그램의 main 메소드와 생성자를 포함한 메소드의 수를 나타낸다. Bubble Sort 프로그램은 여러 개의 정수형 데이터를 오름차순으로 정렬하기 위한 프로그램으로 여섯 개의 메소드로 구성된 프로그램이다. 평가를 하기 위해 논리적인 오류가 한 개 존재하도록 설정하고, 입력 값 '5, 2, 3, 1, 9, 8'로 실험하였다. Hamming 프로그램은 정보 비트에 패리티 비트를 계산하여 추가하는 프로그램으로 여섯 개의 메소드로 구성된 프로그램이다. 평가를 위해 논리적인 오류가 한 개 존재하도록 설정하고, 입력 값 '111011'로 실험하였다. Calculator 프로그램은 재귀 하강 파서(recursive descent parser)를 이용하여 산술 연산을 하는 계산기 프로그램으로 열 개의 메소드로 구성된다. Calculator 프로그램으로 평가를 하기 위해 두 가지 메소드에 각각 한 개의 논리적인 오류를 설정하였다. 그리고 입력 값 '2+3*4'로 실험하였다. 디버깅을 하기 위해 HDTS 시스템으로 생성한 실행 트리의 노드 수는 Bubble Sort 프로그램은 17개, Hamming 프로그램은 9개, Calculator 프로그램은 29개였다.



(a) [그림 2]의 프로그램 (b) Bubble Sort Program

(c) Hamming Program (d) Calculator Program

그림 7. [표 1]의 프로그램들에 대한 디버깅 평가 결과

이 평가는 프로그래머마다 약간의 오차는 발생할 수 있으나 단계적 프로그램 디버깅 기술의 경우, 연산에 관련된 문장을 디버깅하는 횟수만을 포함했으며, break-point를 설정하는 횟수는 제외하였다. 그리고 오류가 발견되지 않을 때까지 디버깅하는 횟수를 포함하였다.

[그림 7]의 (a)는 Java 프로그램에 포함된 두 개의 오류를 발견하고 수정하기 위해 프로그래머가 디버깅 시스템에 접근한 횟수를 비교한 그래프로써, 첫 번째 오류와 두 번째 오류를 발견하기까지의 디버깅 횟수와 총 디버깅 횟수를 비교하였다. 총 디버깅 횟수는 첫 번째 오류와 두 번째 오류를 발견하기 위한 디버깅 횟수와 프로그램 내에 오류가 없음을 확인하기 위한 실행 횟수의 합을 나타낸다. HDT가 단계적 프로그램 디버깅 SWPD1와 SWPD2보다 첫 번째 오류를 발견하기 위해 프로그래머가 디버깅하는 횟수가 감소되었음을 알 수 있다. 그러나 두 번째 오류를 발견하기 위한 디버깅 결과는 SWPD1과 HDT가 동일한 결과가 나왔다. 그 이유는 HDT가 두 번째 오류를 발견하기 위해 이전에 correct라고 응답한 노드에 대해 다시 응답을 하였기 때문에 상대적으로 디버깅 횟수가 증가하였다. 그러나 HDTS에서는 분할을 적용하여 오류가 없는 노드를 제거하였기 때문에 두 번째 오류를 발견할 때 디버깅 횟수가 감소하였음을 알 수 있다. (b)는 Bubble Sort 프로그램에 포함된 논리적인 오류 한 개를 수정하기 위해서 프로그래머가 디버깅 시스템에 접근하는 횟수를 비교한 그래프이다. Bubble Sort 프로그램은 [그림 2]의 예제 프로그램보다 노드 수와 문장 수가 많기 때문에 SWPD2의 디버깅 횟수가 다른 방법에 비하여 크게 증가하였다. HDT의 경우, 오류를 발견하기까지의 디버깅 횟수는 HDTS와 동일하지만 오류가 더 이상 존재하지 않는지 확인하기 위한 실행 횟수가 HDTS 보다 많기 때문에 비효율적이다. (c)는 Hamming 프로그램에 포함된 논리적인 오류 한 개를 수정하기 위해서 프로그래머가 디버깅 시스템에 접근하는 횟수를 비교한 그래프이다. Bubble Sort 프로그램보다 실행 트리의 노드의 수가 적어서 SWPD2를 제외한 세가지 디버깅 방법에서는 프로그래머가 디버깅 시스템에 접근하는 횟수가 적었다. SWPD2는 메소드 내에서 반복문의 실행 횟수가 많아 Bubble Sort보다 디버깅 횟수가 증가하였음을 알 수 있다. (d)는 Calculator 프로그램에 대한 디버깅 결과이다. Calculator 프로그램은 메소드 호출이 많고, 앞의 두 예제 프로그램보다 실행 트리의 레벨이 크다. 따라서 SWPD2가 다른 두 예제 프로그램과 비교해서 디버깅 횟수가 매우 증가하였음을 알 수 있다. SWPD1은 SWPD2보다 실행 횟수가 적지만, break-point를 사용하는 횟수가 많고 다른 방법보다 프로그램을 여러 번 재실행해야 하는 문제가 있다.

[표 1]의 네 가지 예제 프로그램에 대한 테스트 결과 자바 프로그램에 포함된 논리적인 오류를 발견하고 수정하는데 본 논문에서 제안한 HDTS가 기존의 단계적 프로그램 디버깅 기술과 HDT보다 프로그래머가 디버깅하는 횟수를 감소시켰다. 특히, HDTS는 메소드의 수가 증가하고 각 메소드에 포함된 문장의 수가 증가할수록 전통적인 디버깅 방법과 HDT 보다 프로그래머의 디버깅 횟수를 더욱더 감소시킬 수 있어 자바 프로그램을 디버깅하는 경우에도 효과적으로 적용될 수 있다.

VI. 결 론

본 논문에서는 디버거 사용자가 자바 프로그램에서 논리적인 오류를 디버깅하는 횟수를 줄이기 위해서 이전에 제안한 HDT에 프로그램 분할 기술을 적용하는 HDTS을 제안하였다. HDTS는 알고리즈믹 프로그램 디버깅 기술과 단계적 프로그램 디버깅 기술을 혼합해서 사용하는 HDT에 프로그램 분할 기술을 적용하여 사용자가 디버깅하는 횟수를 최소화하는 프로그램 디버깅 기술이다.

HDT는 알고리즈믹 프로그램 디버깅 기술과 단계적 프로그램 디버깅 기술을 혼합하여 기존의 전통적인 디버깅 방법만을 가지고 프로그램을 디버깅하는 방법보다 쉽고 빠르게 디버깅할 수 있었다. 그러나 여러 개의 논리적인 오류를 발견하고 오류가 모두 수정되었는지 확인하기 위해서는 디버깅 시스템이 오류가 없다는 메시지를 보낼 때까지 실행 트리를 매번 재생성 해야 하고 프로그래머는 이전에 correct라고 응답한 노드까지도 매번 응답해야 했다. 또한 불필요한 노드와 문장을 디버깅 해야 하는 문제가 있었다. 이 문제는 프로그래머가 디버깅하는 횟수를 증가시켰다.

본 논문에서는 네 가지 경우에 분할 기술을 HDT에 적용하여 사용자가 디버깅하는 횟수를 줄였다. 첫째, 프로그래머가 실행 트리의 탐색으로부터 어느 노드에서 incorrect라고 응답하고 incorrect의 원인이 되는 변수를 선택했을 때 그 변수와 종속성이 없는 자식 트리를 제거하였다. 둘째, HDTS가 오류가 포함된 노드인 메소드를 발견하면 단계적 프로그램 디버깅을 시작하기 전에 정적 분할을 적용하여 디버깅에 관련이 없는 문장들을 제거하고 디버깅을 시작하였다. 셋째, 오류 교정 후 아직 탐색하지 않은 노드에서 오류와 관련된 변수와 종속성이 존재하는 노드를 제거하였다. 넷째, 실행 트리를 다시 생성할 때 이전 실행 트리의 탐색에서 correct라고 응답한 노드와 자식 노드를 모두 제거하였다. 이러한 방법을 적용하기 위해서 본 논문에서는 변수들 사이에 종속성이 표현된 확장 실행 트리를 정의하였다. 그 결과 실행 트리를 한번 생성하면 여러 개의 오류를 찾을 수 있고, 디버깅에 불필요한 노드와 문장을 제거 할 수 있어서 이전에 제시한 HDT보다 메소드와 문장의 수가 증가할수록 사용자가 디버깅에 참여하는 횟수를 더 많이 줄일 수 있었다.

참 고 문 헌

[1] 고훈준, 유원희, "자바 언어를 위한 알고리즈믹 프로그램 디버깅 기술의 설계", 한국정보처리학회 논문지 A, 제11-A권, 제1호, pp.97-108, 2004.

[2] 고훈준, 장경수, "자바 프로그램의 효율적인 디버깅을 위한 HDT의 설계", 경인논집, 제12호, pp.277-293, 2004.

[3] H. Agrawal, *Toward Automatic Debugging of Computer Programs*, Ph. D. Thesis, Purdue University, 1991.

[4] Z. Chen and B. Xu, "Slicing Object-Oriented Java Programs," ACM SIGPLAN Notices, Vol.36, No.4, pp.33-40, 2001.

[5] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," P. of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, Vol.23, No.7, pp.35-46, 1998.

[6] M. Khouzam and T. Kunz, "Single Stepping in Event-Visualization Tools," P. of the 1996 CAS Conference, pp.103-114, 1996.

[7] G. Kokai, L. Harmath, and T. Gyim'othy, "Algorithmic Debugging and Testing of Prolog Programs," The Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments Leuven- ICLP '97, pp.14-21, 1997.

[8] H. J. Kouh and W. H. Yoo, "Algorithmic Debugging in Java Programs," ACIS Annual International Conference on Computer and

Information Science - ICIS'02, pp.569-574, 2002.

[9] H. J. Kouh and W. H. Yoo, "The Efficient Debugging System for Locating Logical Errors in Java Programs," P. of International Conference on Computational Science and Its Application - ICCSA 2003, LNCS, Vol.2667, pp.684-693, 2003.

[10] E. Shapiro, *Algorithmic Program Debugging*, MIT Press, Cambridge, 1983.

[11] F. Tip, "A survey of program slicing techniques," J. of Programming Languages, Vol.3, No.3, pp.121-189, 1995.

[12] M. Weiser, "Program Slicing," IEEE Transaction on Software Engineering, Vol.10, No.4, pp.352-357, July 1984.

저 자 소 개

고 훈 준(Hoon-Joon Kouh) 정회원

- 1998년 2월 : 인하대학교 생물공학과(공학사)
- 2000년 2월 : 인하대학교 전자계산공학과(공학석사)
- 2004년 2월 : 인하대학교 전자계산공학과(공학박사)
- 2004년 3월~현재 : 경인여자대학 컴퓨터정보학부 교수

<관심분야> : 디버거, 웹서비스, 프로그래밍 언어, 생물정보학 등