자바 언어를 위한 알고리즈믹 디버깅 기술의 설계

고 훈 준†·유 원 희††

요 약

본 논문은 자바 프로그램에서 논리적인 오류를 발견하기 위해 알고리즈믹 디버깅 기술의 이용을 제안한다. 알고리즈믹 디버깅 기술은 원시 프로그램으로부터 실행 트리를 생성하고, 실행 트리로부터 프로그램 내에 포함된 논리적인 오류를 하향식 방법으로 발견하는 반자동화 디버깅 기술이다. 따라서 알고리즈믹 디버깅 기술은 다양한 프로그래밍 언어에 알맞은 실행 트리를 생성하는 것이 중요하다. 본 논문에서는 자바 프로그램으로부터 실행 트리를 생성하는 방법을 제안하고 예제를 통해서 자바 프로그램을 위한 알고리즈믹 디버깅 기술을 확인한다. 이 접근은 전통적인 순차적 디버깅 기술보다 사용자와 디버깅 시스템 사이에서 상호작용의 수를 줄일 수 있다.

Design of an Algorithmic Debugging Technique for Java Language

Hoon Joon Kouh†·Weon Hee Yoo††

ABSTRACT

This paper proposes to use an algorithmic debugging technique for locating logical errors in Java programs. The algorithmic debugging is a semi-automated debugging technique that builds an execution tree from a source program and locates logical errors, if any, included in the program from the execution tree with top-down method. So, it is very important to build a suitable execution tree from the various programming languages. In this paper we propose the method for building an execution tree from Java programs and walk through an example. This approach could reduce the number of interactions between a user and a debugging system than the traditional step-wise debugging technique.

키워드 : 알고리즈믹 디버깅(Algorithmic Debugging), 자바(Java), 실행 트리(Execution Tree)

1. 서 론

프로그램 디버깅은 프로그래밍 작업의 많은 시간을 요구하는 것 중의 하나이다. 특히, 프로그램의 크기가 크고 복잡한 프로그램을 디버깅하거나, 다른 프로그래머에 의해 작성된 프로그램을 디버깅하는 것은 쉽지 않다[1, 2]. 과거에는 많은 프로그래머들이 절차지향 언어인 명령형 언어를 이용하여 프로그래밍을 하였으나, 최근에는 객체지향 언어를 이용하여 프로그래밍을 하는 경향이 증가하고 있다. 그 중 대표적인 객체지향 언어 중의 하나가 바로 자바 언어이다.

현재 자바 프로그램 내에 포함되어 있는 논리적인 오류(logical errors)를 디버깅하는 기술은 절차지향 언어에서 사용하던 전통적인 디버깅 기술을 사용하고 있다. 그러나 최근의 자바 프로그램은 클래스와 클래스 내에 메소드의 수가 증가하고 있다. 그래서 전통적인 디버깅 기술로 다양하고 복잡한 자바 프로그램을 디버깅하는 일은 쉽지 않다[3]. 따라서 최근에는 자바 프로그램을 효율적으로 디버깅하기 위한 많은 연구가 진행되고 있다. 이러한 연구는 기존의 논리형 언어, 함수형 언어, 명령형 언어에서 연구하던 다양한 디버깅 기술들을 자바 언어에 적용하려는 시도였다. 지금까지 이와 같은 접근은 slicing[4, 5], model-based diagnosis [6-8], query-based debugger[9] 등을 확장하여 자바 프로그램을 디버깅하는 연구가 진행되었다. 위의 모든 연구는 사용자가 디버깅에 참여하는 횟수를 최소화시키면서 프로그램 내에 있는 오류를 빠르게 찾고, 쉽게 수정하려는 시도이다. 그러나 아직까지 이들 방법은 작은 자바 프로그램이나 일부의 자바 프로그램에만 적용이 가능하다.

본 논문에서는 알고리즈믹 디버깅 기술(algorithmic debugging technique)[10-12]을 자바 프로그램에 적용하는 방법을 제안한다. 알고리즈믹 디버깅 기술은 Shapiro[11]에 의해 제안된 소프트웨어 디버깅 기술로서, 논리형 언어인 프롤로그(Prolog)에서 프로그램 디버깅을 위한 이론적인 구조

† 준 회 원 : 인하대학교 대학원 전자계산공학과
†† 종신회원 : 인하대학교 전자계산공학과 교수
논문접수 : 2003년 4월 25일, 심사완료 : 2003년 10월 22일

로 처음 제안되었다. 이 기술은 지금까지 논리형 언어, 함수형 언어, 명령형 언어에는 적용되었으나 객체지향 언어인 자바에는 적용된 사례가 없다.

알고리즈믹 디버깅 기술은 프로그램 내의 모든 프로시저/함수의 호출관계를 표현한 함수 수준의 실행 트리(execution tree)를 생성하고, 그 트리로부터 사용자와 디버깅 시스템이 상호 작용함으로써 프로그램 내에 포함된 논리적인 오류를 하향식 방법(top-down method)으로 발견하는 반자동화 디버깅 기술이다. 따라서 이 기술로 효율적인 디버깅을 하기 위해서는 디버깅에 필요한 다양한 정보를 가지고 있는 실행 트리를 생성하는 것이 중요하다. 그러나 실행 트리를 구성하는 방법은 언어가 가지고 있는 다양한 특성 때문에 프로그래밍 언어의 종류에 따라 고려 사항이 다르다. 일반적으로 논리형 언어와 함수형 언어는 부작용(side-effect)이 없기 때문에 Shapiro의 순수 알고리즈믹 디버깅 기술로 실행 트리를 생성하고 탐색할 수 있다. 반면에 지연 함수형 언어(lazy functional languages)는 평가되지 않은 표현식에 의해 불완전한 실행 트리가 생성되고, 사용자가 그 실행 트리를 이해하기 어렵게 만든다. 이 문제를 해결하기 위해 Nilsson은 strictification 기술을 제안하였다[13, 14]. 명령형 언어는 전역 변수와 참조형 매개변수에 의해서 함수의 부작용을 가지기 때문에 불완전한 실행 트리가 생성된다. 이 문제를 해결하기 위한 방법은 부작용을 완전히 제거하거나 알고리즈믹 디버깅 기술의 원리에 충돌이 되는 프로그램 구조만 변형하는 것이다. 또한 실행 트리의 각 노드에 저장되는 추적 정보(trace information)의 내용도 프로그래밍 언어에 따라 다르다. 이와 같은 논의는 Shahmehri에 의해 이루어 졌다[10]. 따라서 이들 방법에 서술되어 있는 고려 사항은 객체지향 언어인 자바 프로그램에는 적당하지 않다.

자바는 기존의 논리형 언어, 함수형 언어, 그리고 명령형 언어와는 다른 특징을 가지고 있다. 첫째, 자바는 여러 개의 클래스(class)로 구성되고 그 내부에 여러 개의 생성자(constructor)와 메소드(method)를 가질 수 있다. 둘째, 자바는 다양한 종류의 멤버 변수(member variables)를 가진다. 셋째, 하나의 클래스로부터 여러 객체를 생성할 때마다 각 객체들은 동일한 이름의 멤버 변수를 가진다. 넷째, 다형성을 지원하기 때문에 동일한 이름의 생성자와 메소드를 가질 수 있다. 그리고 마지막으로 상속성에 의해 부모의 메소드와 멤버 변수를 상속 받을 수 있으며 메소드를 재정의(overriding)할 수 있다[15].

본 논문에서는 순차적인 자바 프로그램에서의 논리적인 오류에 대한 디버깅만 고려한다. 그리고 자바 프로그램으로부터 실행 트리를 생성하는 방법을 제안하고 예제를 통해서 자바에서 알고리즈믹 디버깅 기술을 확인한다.

본 논문의 구성은 다음과 같다. 2장은 자바의 전통적인 디버깅 기술과 문제점을 서술하고 간략하게 알고리즈믹 디버깅 기술을 서술한다. 3장은 객체지향 언어인 자바 프로그램으로부터 실행 트리를 생성하는 방법을 제안한다. 그리고 4장은 예제 프로그램을 통하여 알고리즈믹 디버깅 기술의 타당성을 보이고, 5장은 알고리즈믹 디버깅 시스템의 구조 및 구현을 서술하고 평가한다. 마지막으로, 6장에서는 결론 및 향후 연구 계획을 제시한다.

2. 배 경

본 장에서는 자바의 전통적인 디버깅 기술과 문제점을 서술하고, 알고리즈믹 디버깅 기술의 원리를 서술한다.

2.1 자바 프로그램에서 전통적인 디버깅

자바 프로그램에서 논리적인 오류를 발견하기 위한 전통적인 디버깅 기술은 두 가지가 있다. 첫째, 사용자는 원시 프로그램을 직접 분석하거나 의심이 되는 위치에서 'System.out.println'와 같은 화면 출력 명령어를 사용하여 디버깅한다. 이 방법은 아직까지 주된 프로그램 디버깅 방법으로 다양한 프로그래밍 언어에서 단순하고 효율적이다. 그러나 사용자가 논리적인 오류의 위치를 정확하게 예상할 수 없기 때문에 디버깅이 쉽지 않다. 둘째, 순차적 디버깅 기술이 있다. 사용자는 step-over, step-into, go, 그리고 break-point와 같은 명령어를 가지고 자바 프로그램의 문장들을 순차적으로 디버깅할 수 있다. step-over와 step-into는 일반적으로 한번 명령에 하나의 문장을 실행한다. go와 같은 명령어는 break-point와 함께 여러 문장을 한번에 실행한다. 사용자는 break-point를 가지고 하나 또는 그 이상의 문장을 지정할 수 있고 문장의 실행을 제어한다. 사용자는 이들 명령어로부터 추적 정보를 얻는다. 추적 정보는 각각의 위치에 도착할 때마다 단순한 메시지를 포함하거나 어떤 변수들의 값들을 포함한다. 그러나 최악의 경우 사용자가 프로그램의 모든 문장을 한 문장씩 실행해야 한다. 따라서 이 기술은 사용자가 오류를 발견하기 위해 디버깅 시스템에 접근하는 많은 횟수를 요구한다.

2.2 알고리즈믹 디버깅 기술의 기본 원리

본 절에서는 실행 트리의 기본 구성 방법과 실행 트리의 탐색으로 나누어 알고리즈믹 디버깅 기술을 서술한다.

2.2.1 실행 트리

프로그램의 실행 트리는 프로그램의 실행에 대한 정보를 포함하고 있는 트리 구조로 각 노드는 프로그램의 실행에

서 호출하는 함수/프로시저로 구성된다. 하나의 노드는 함수/프로시저 이름과 입/출력 매개변수의 이름과 값들로서 추적 정보가 구성된다. 실행 트리의 구조는 트리 $T=(N, E, S)$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 다음과 같이 정의될 수 있다.

$$n = (p, in, out)$$

이때 p는 함수/프로시저 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_o는 출력 변수 이름이다. 그리고 x_i와 x_o은 그 변수들의 값이다. 함수 p는 입력 in에 의해 실행되고 출력 out을 반환한다.

따라서 N은 다음과 같이 유한한 순서의 집합으로 정의될 수 있다.

$$N = \{(p_1, in_1, out_1), (p_2, in_2, out_2), \cdots, (p_k, in_k, out_k)\}$$

함수 p_1는 입력 in_1에 의해 실행되고 출력 out_1을 반환하고, 함수 p_2는 입력 in_2에 의해 실행되고 출력 out_2을 반환한다. 그리고 함수 p_k는 입력 in_k에 의해 실행되고 출력 out_k을 반환한다.

$0 \le i \le j \le k$를 만족하는 (p_i, in_i, out_i)와 (p_j, in_j, out_j)을 각각 n_i와 n_j라고 가정하고 p_i에서 p_j을 호출한다면, n_i는 부모 노드, n_j는 자식 노드가 된다. 그리고 그 관계는 간선 $(n_i, n_j) \in E$로 표현될 수 있다. 이것은 노드 n_i로부터 노드 n_j로의 잠재적인 제어의 이동과 대응된다. 이러한 실행 트리 T는 시작 노드 S로부터 단말 노드(terminal node)까지 왼쪽에서부터 오른쪽으로 함수/프로시저의 호출 순서에 따라 구성된다.

2.2.2 실행 트리의 탐색

알고리즈믹 디버깅 기술은 실행 트리를 구성하고 이 트리를 탐색하여 프로그램 내에 포함된 논리적인 오류를 발견하는 반자동화 디버깅 기술이다. 이 기술은 (알고리즘 1)과 같이 실행 트리의 루트 노드 또는 임의의 노드를 선택함으로써 디버깅을 시작할 수 있다. 따라서 (p_0, in_0, out_0)는루트 노드 또는 사용자가 처음에 선택한 노드이다. 그리고 오류를 발견할 때까지 부트리 t를 탐색한다. 탐색은 하위 레벨부터 상위 레벨로 전위 탐색(preorder traversal)에 의해 각각의 노드들을 탐색하고 각 노드에서 기대할 수 있는 행동에 대해 사용자에게 질문함으로써 동작한다. 사용자는 각 함수의 행동에 대해 correct 또는 incorrect로 응답할 수 있다. 사용자가 correct라고 응답할 경우에는 디버깅 시스템은 현재 노드와 그 자식 노드에 오류가 없다고 인식한다. 그리고 더 이상 자식 노드는 탐색하지 않고 형제 노드로 이동하여 탐색한다. 사용자가 incorrect라고 응답할 경우에는 디버깅 시스템은 현재 노드와 그 자식 노드 중에 오류가 있다고 보고 자식 노드로 이동하여 질문한다. 탐색은 더 이상 자식 노드가 없거나 사용자가 correct라는 응답을 할 때까지 진행된다. 그래서 만약 디버깅 시스템이 탐색을 중지한다면, 어떤 함수 f는 두 가지 경우에 오류를 가진다 : 첫 번째 경우는 함수 f는 함수 호출이 없다. 두 번째 경우는 함수 f 내에 있는 모든 호출되는 함수들은 사용자의 기대에 만족한다. 따라서 디버깅 시스템의 출력은 입력 매개변수와 기대되는 결과에 의한 사용자의 응답으로 생성되고 오류가 포함된 어떤 함수를 나타낸다.

```
Input : (p0, in0, out0) ∈ N at T
Output : (pj, inj, outj) ∈ N at T  or NIL
1 procedure Algorithmic_Debugging ((p0, in0, out0),
                                   (pj, inj, outj))
2 begin
    let t= (p1, in1, out1),(p2, in2, out2),…,(pn, inn, outn)
    be the top level trace nodes of (p0, in0, out0)
3   i := 1
4   if (Query ((p0, in0, out0)) = correct) then
5     return NIL
6   else
7     if ∃t then
8       while i = n do
9         if (Query ((pi, ini, outi)) = correct) then
10          if the sibling node of (pi, ini, outi) then
11            (p0, in0, out0) := the sibling node
                                 of (pi, ini, outi)
12            Algorithmic_Debugging ((p0, in0, out0),
                                     (pj, inj, outj))
13            break
14          else
15            return (pj, inj, outj) := the parent node of
                                        (pi, ini, outi)
16          fi
17        fi
18        i := i + 1
19      od
20      if i = n then
21        return (pj, inj, outj) := (pi, ini, outi)
22      fi
23    else
24      return (pj, inj, outj) := (p0, in0, out0)
```

```
25    fi
26  fi
27 end
```

(알고리즘 1) 알고리즈믹 디버깅 기술의 알고리즘

알고리즈믹 디버깅 기술의 예는 (그림 1)과 같다. 프로그램은 함수 *p*, *q*, *r*로 구성되고 입력 매개변수 *a*와 *c*, 출력 매개변수 *b*와 *d*를 가지는 함수 *p*를 생각해보자. 변수 *b*의 값은 함수 *q*를 호출함으로써 계산되고 변수 *d*는 함수 *r*을 호출함으로써 계산된다. 프로그램을 실행하기 위해 다음을 가정하자.

첫째, 입력 값 a'와 c'를 갖는 함수 *p*는 출력 값 b'와 d'를 반환한다.
둘째, 오류는 잘못된 출력 값 d'가 발생하는 함수 *r*에 있다.

```
function p(int a, c ; int b, d) {
        q(a, b) ;     r(c, d) ;
}
function q(int a ; int b) {
        ...
}
function r(int c ;  int d) {
        ...
}
```

(그림 1) 함수 *q*와 *r*을 호출하는 함수 *p*의 예

디버깅 시스템이 (그림 1)의 프로그램을 실행하면 실행 트리 *T*는 다음과 같이 생성된다.

$N = \{n_1, n_2, n_3\}$

$E = \{(n_1, n_2), (n_1, n_3)\}$

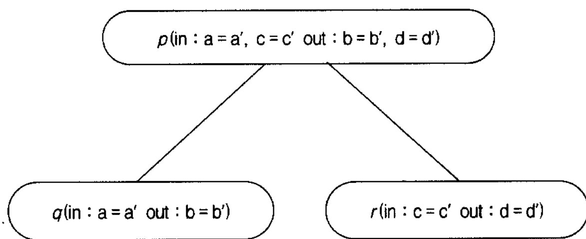
$S = \{n_1\}$

$n_1 = (p, \{(a, a'), (c, c')\}, \{(b, b'), (d, d')\})$

$n_2 = (q, \{(a, a')\}, \{(b, b')\})$

$n_3 = (r, \{(c, c')\}, \{(d, d')\})$

실행 트리를 도식화하면 (그림 2)와 같다. 각 노드에는 함수 이름, 입력 값, 출력 값이 표시된다.



(그림 2) (그림 1)의 프로그램으로부터 생성된 실행 트리

(그림 2)의 실행 트리를 기반으로 동작하는 알고리즈믹 디버깅 시스템과 사용자와의 대화는 다음과 같다.

p (in : a = a', c = c', out : b = b', d = d')?
$ incorrect
q (in : a = a', out : b = b')?
$ correct
r (in : c = c', out : d = d')?
$ incorrect
An error has been localized inside the body of procedure r

이탤릭체는 디버깅 시스템이 사용자에게 질문하는 문장이고, '$'에 의해 표현되는 것은 사용자의 응답을 나타낸다. 노드 n_1과 n_3에 대해 사용자가 incorrect라고 응답하였고, 디버깅 시스템은 이 정보로부터 함수 *r*에 논리적인 오류가 있음을 사용자에게 알려준다.

이러한 알고리즈믹 디버깅 기술은 사용자가 전체 프로그램을 정확히 이해하지 못한 경우에도 함수 단위로 입력과 출력 값을 예상하여 응답함으로써 디버깅할 수 있기 때문에 함수/프로시저의 호출이 많은 프로그램의 경우, 알고리즈믹 디버깅 기술은 순차적 디버깅 기술보다 시스템이 사용자에게 요구하는 부담이 줄어든다.

3. 자바 프로그램을 위한 알고리즈믹 디버깅 기술의 설계

앞 절에서 서술한 알고리즈믹 디버깅 기술로 자바 프로그램을 디버깅하기 위해서는 그 언어에 알맞은 실행 트리를 생성해야 한다. 따라서 함수를 노드로 사용하는 대신에 생성자와 메소드를 기반으로 이들의 호출 관계를 트리로 표현하는 실행 트리를 구성하는 것이 중요하다. 본 논문에서는 알고리즈믹 디버깅 원리에 상충되는 프로그램 구조만 변형해서 실행 트리를 생성하는 논의를 한다.

3.1 자바 프로그램의 실행 트리

자바 프로그램의 실행 트리는 노드로서 함수/프로시저 대신에 생성자를 포함한 메소드들 사이의 호출 관계로 구성된다. 일반적인 실행 트리 구조는 그래프 $T = (N, E, S)$에 의해서 정의된다. 이때 *N*의 한 노드는 자바 프로그램에서 메소드(생성자를 포함)에 대응된다. 그리고 시작 노드 *S*는 *main* 메소드이다. 자바 프로그램에서는 다형성에 의해 동일한 이름의 메소드가 존재할 수 있다. 따라서 2.2.1에서 정의한 노드 (*p*, *in*, *out*)에서는 동일한 이름의 메소드를 구별할 수 있는 정보가 불충분하다. 그러므로 동일한 메소드 이름을 사용자가 구별하기 위해서는 노드에 클래스 이름과

객체의 이름과 같은 정보를 추가하여 다음과 같이 정의해야한다.

$$n = (o, c, m, in, out)$$

이때 *o*는 객체 이름, *c*는 클래스 이름이고 *m*은 메소드 또는 생성자 이름이다. *in*은 입력 변수의 집합이고 *out*은 출력 변수의 집합이다.

3.2 실행 트리의 노드

자바 프로그램에서 실행 트리의 노드는 생성자를 포함한 메소드에 해당된다. 생성자는 클래스로부터 객체를 생성할 때 단 한번만 실행되는 메소드로 주로 초기화 기능을 위해 사용된다. 메소드는 클래스 내에서 객체가 할 수 있는 동작을 정의한 것으로, 기존의 절차지향 언어에서 사용하는 함수/프로시저의 형식과 같다. 본 논문에서는 자바 프로그램으로부터 실행 트리의 노드를 다음과 같이 정의한다. 첫째, 생성자에서 입력 변수 *in*은 매개변수와 생성자 내에서 참조(reference)되는 멤버 변수이고 출력 변수 *out*은 변경(modification)되는 매개변수와 멤버 변수이다. 둘째, 메소드에서 입력 변수 *in*은 매개변수와 메소드 내에서 참조되는 멤버 변수들이고 출력 변수 *out*은 반환 변수와 변경되는 멤버 변수이다. 다음 프로그램의 예를 보자.

```
class SimpleCal {
        int num ;
        public SimpleCal() {
                this(0) ;
        }
        public SimpleCal(int num) {
                this.num = num ;
        }
        int incre(int num) {
                num = num + 1 ;      return num ;
        }
        int addsum (int a, int b) {
        int c ;
                a = incre(a) ; b = incre(b) ;
                c = a + b ;          return c ;
        }
        int totalsum (int a, int b) {
                int var1, var2 ;
                var1 = addsum(a, b) ;
                var2 = var1 + num ;  return var2 ;
        }
}
```

(그림 3) *SimpleCal* 클래스에서 생성자와 메소드의 예

(그림 3)은 두 개의 생성자와 세 개의 메소드로 구성된 클래스 *SimpleCal*이다. 같은 이름의 생성자는 중복 생성자(overloading constructor)이다. 중복 생성자는 타입의 종류 또는 매개변수의 개수가 다르기 때문에, 실제 동작하는 생성자를 구분할 수 있다. 따라서 실행 트리를 생성하는데 문제가 되지 않는다. 첫 번째 생성자는 매개변수가 없고 두 번째 생성자는 두 개의 매개변수를 입력받아 두 멤버 변수에 할당한다. 이 클래스로부터 Obj1, Obj2 객체의 선언과 생성 명령은 다음과 같다.

```
SimpleCal Obj1 = new SimpleCal() ;
SimpleCal Obj2 = new SimpleCal(5) ;
```

그리고 Obj1, Obj2 객체가 실제로 생성자를 호출하여 실행하는 과정은 다음과 같이 표현될 수 있다.

```
Obj1.SimpleCal() ;
Obj1.SimpleCal(in : num = 0 out : this.num = 0) ;
Obj2.SimpleCal(in : num = 5 out : this.num = 5) ;
```

이 문장들의 표현 형식은 '객체이름.생성자이름([매개변수의입/출력정보])'이다. 생성자의 경우 클래스이름과 생성자 이름이 동일하므로 생략한다. (그림 3)에서 *this* 메소드는 키워드로서 입력 값 0으로 한 개의 매개변수를 갖는 생성자를 호출한다. 출력변수는 할당받는 왼쪽 변수 모두이다.

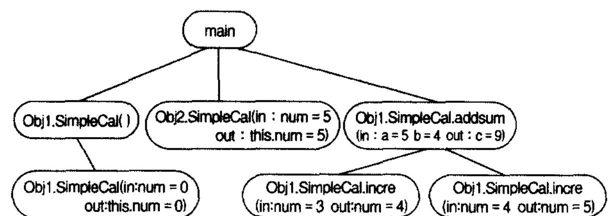
만약, Obj1 객체가 *addsum* 메소드를 호출하고, 그 결과를 *sum* 변수에 할당한다고 가정하면 그 표현식은 다음과 같다.

```
sum = Obj1.addsum(3, 4) ;
```

그리고 Obj1 객체에서 *addsum* 메소드를 호출한 후 프로그램이 실행되는 과정은 다음과 같다.

```
Obj1.SimpleCal.addsum(in : a = 3, b = 4 out : c = 9) ;
Obj1.SimpleCal.incre(in : num = 3 out : num = 4) ;
Obj1.SimpleCal.incre(in : num = 4 out : num = 5) ;
```

이 문장들의 표현 형식은 '객체이름.클래스이름.메소드이름([매개변수의입/출력정보])'이다. 메소드의 경우 클래스 이름의 사용은 상속 관계에서 부모 클래스의 메소드와 자식 클래스의 재정의 메소드(overriding method)와 구별을 위해 필요하다. 위의 결과로부터 실행 트리를 구성하면 (그림 4)와 같다.



(그림 4) 클래스 *SimpleCal*에서 객체 Obj1과 Obj2에 의해 생성된 실행 트리

3.3 자바 API

자바 API는 많은 내장 패키지를 가지고 있고 그 패키지에는 다양한 기능을 제공하는 메소드가 있다. 자바 프로그래머는 자바 API에 있는 미리 정의된 소프트웨어 패키지들을 이용하여 쉽게 프로그램을 작성할 수 있다. 패키지는 많은 클래스와 다른 패키지들로 구성된다. 또한 클래스는 많은 변수와 메소드를 가지고 있다. 이 메소드들은 미리 정의된 메소드들이기 때문에 올바르게 실행된다고 가정한다. 따라서 실행 트리를 생성할 때 API에 있는 메소드는 노드로 포함되지 않는다. 명백히, 이것은 실행 트리의 노드 수를 줄일 수 있고, 실행 트리를 생성하는 시간도 줄일 수 있다.

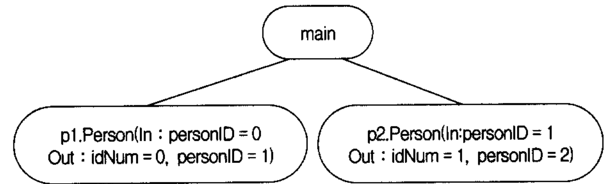
3.4 노드의 입/출력 변수의 표현 정의

객체지향 언어인 자바에서 실행 트리의 노드에서 표현되는 입/출력 변수는 매개변수와 멤버 변수로 구성된다. 일반적으로 노드의 입/출력 변수를 위한 기호는 in, out으로 표현된다. 멤버 변수는 객체 변수, 클래스 변수, 종단 변수로 구분할 수 있고, 객체 변수는 객체 속성 변수와 객체 참조 변수로 구분할 수 있다. 이러한 멤버 변수들은 클래스 내의 메소드들 내에서 자유롭게 값을 할당받거나 다른 변수에게 값을 할당할 수 있다. 따라서 이 변수들의 값에 대한 변화는 사용자에게 중요한 정보가 된다. 다음 (그림 5)의 예제를 보자.

```
class Person {
        long idNum ;
        static long personID = 0 ;
        public Person() {
                idNum = personID ;
                personID = personID + 1 ;
        }
}
class Employee {
        public static void main(String args[ ]) {
                Person p1 = new Person() ;
                Person p2 = new Person() ;
        }
}
```

(그림 5) 멤버 변수의 예 : idNum은 객체 속성 변수, personID는 클래스 변수

Person 클래스는 객체 속성 변수 idNum, 클래스 변수 personID, 그리고 *Person* 생성자로 구성된다. 메인 클래스 *Employee*는 *Person* 클래스로부터 p1, p2의 객체를 인스턴스(instance)한다. *Person* 생성자는 객체를 하나 생성할 때마다 personID를 1씩 증가시키고 그 값을 idNum 변수에 할당하는 연산을 수행한다. 일반적으로 (그림 5)로부터 실행 트리를 생성하면 매개변수가 없기 때문에 실행 트리에서 추적정보의 입/출력 정보가 없게 된다. 따라서 (그림 6)와 같이 멤버 변수를 고려하여 실행 트리를 생성해야 한다.

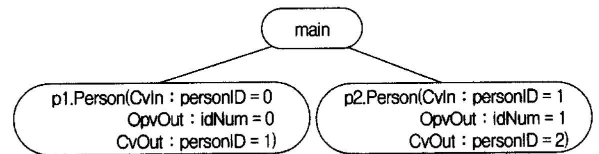


(그림 6) (그림 5)의 프로그램으로부터 멤버 변수의 정의와 할당을 고려해서 생성한 실행 트리

또한 변수의 종류는 다양하기 때문에 객체 속성 변수, 객체 참조 변수, 클래스 변수 그리고 부모 클래스의 변수를 구분하여 표시하면 사용자는 각 메소드의 구조와 흐름을 쉽게 이해할 수 있다. 따라서 멤버 변수 종류를 실행 트리에 적용하여 표현하는 것이 필요하다. 따라서 본 논문에서는 멤버 변수를 다음과 같이 구분하여 정의한다.

- 객체 속성 변수가 메소드 내에서 참조될 경우에는 메소드의 입력 변수 기호로 OpvIn(object property input variable)을 사용하고, 메소드 내에서 변경될 경우에는 메소드의 출력 변수 기호로 OpvOut(object property output variable)을 사용한다.
- 객체 참조 변수가 메소드 내에서 참조될 경우에는 OrvIn(object reference input variable)으로, 메소드 내에서 변경될 경우에는 OrvOut(object reference output variable)으로 나타낸다.
- 클래스 변수가 메소드 내에서 참조될 경우에는 CvIn(class input variable)으로, 메소드 내에서 변경될 경우에는 CvOut(class output variable)으로 나타낸다.
- 부모 클래스의 변수를 상속받아 메소드 내에서 참조될 경우에는 PvIn(parent input variable)으로 메소드 내에서 변경될 경우에는 PvOut(parent output variable)으로 나타낸다.

이러한 정의를 (그림 6)에 적용하면 (그림 7)의 실행 트리를 얻을 수 있다.



(그림 7) (그림 6)의 실행 트리를 멤버 변수의 종류에 따라 구분하여 나타낸 실행 트리

3.5 노드의 입/출력 값의 표현 정의

객체는 클래스로부터 생성되며 필요한 정보는 객체가 생

성될 때 생성자의 매개변수를 통하여 정보를 받는다. 또한 객체는 메소드를 사용하여 필요한 동작을 하고, 메소드의 매개변수를 사용하여 객체가 동작하는데 필요한 정보를 보내고 받는다. 따라서 실행 트리에서 입/출력에 사용되는 x_i와 x_o의 매개변수에 대한 기본 자료형은 다음과 같이 정의된다.

- 불린형 변수인 boolean 변수는 입/출력 변수에서 true 또는 false 값으로 표현된다.
- 정수형 변수인 byte, short, integer, long 변수는 입력 변수에서는 2진수, 8진수, 10진수, 16진수 형태로 표현될 수 있고 출력변수에서는 모두 10진수 형태로 표현된다.
- 실수형 변수인 float, double 변수는 입력 변수에서는 10진수 또는 10의 지수 형태로 표현되고 출력 변수에서는 모두 10진수 형태로 표현된다.
- 문자형 변수인 char 변수는 '문자'로 표현된다.

매개변수가 참조 자료형을 갖는 변수일 경우 다음과 같이 정의된다.

- 문자열 객체 변수는 "문자열"로서 표현된다.
- 배열 객체 변수는 원소의 나열로서 표현된다. 일반적으로 $n+1$개의 원소를 갖는 일차원 배열의 경우는 $[a_0, a_1, \cdots, a_n]$와 같이 표현된다. a_i는 배열 $i+1$번째 원소를 나타낸다 $(0 \le i \le n)$.
  다차원 배열은 배열의 배열로 표현된다. 예를 들어, $n+1$행과 $m+1$열로 구성된 이차원 배열은 $[[a_{00}, a_{01}, \cdots, a_{0m}], [a_{10}, a_{11}, \cdots, a_{1m}], \cdots, [a_{n0}, a_{n1}, \cdots, a_{nm}]]$와 같이 표현된다. a_{ij}는 배열 $i+1$행 $j+1$열을 나타낸다 $(0 \le i \le n, 0 \le j \le m)$.
- 클래스 객체 변수는 객체에 대한 데이터인 멤버 변수들을 변수 이름과 값의 쌍으로 나열된다. 만약 객체가 n개의 멤버 변수 $x_1, x_2, x_3, \cdots, x_n$와 이들 변수에 대한 값 $v_1, v_2, v_3, \cdots, v_n$을 갖고 있다면 $\langle x_1 = v_1, x_2 = v_2, x_3 = v_3, \cdots, x_n = v_n \rangle$로 표현된다.

4. 알고리즈믹 디버깅 기술을 이용한 자바 프로그램의 테스트

(그림 3)과 (그림 8)은 두 개의 클래스 *SimpleCal*과 *Calculation*으로 구성된 자바 프로그램이다. *SimpleCal* 클래스는 두 개의 생성자와 세 개의 메소드로 구성된다. 그리고 *Calculation* 클래스는 프로그램이 시작하는 *main* 메소드로 구성된다.

```
public class Calculation {
   public static void main(String args[ ]) {
         int a, b ;
         SimpleCal Obj1 = new SimpleCal() ;
         a = Obj1.addsum(3,4) ;
         SimpleCal Obj2 = new SimpleCal(5) ;
         b = Obj2.totalsum(3,4) ;
         System.out.println(a+ " " + b) ;
   }
}
```

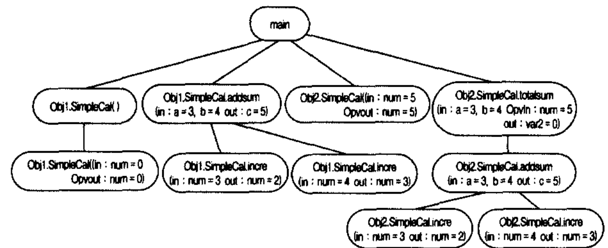
(그림 8) 알고리즈믹 디버깅 기술의 테스트를 위한 자바 프로그램의 예

(그림 8)의 프로그램을 실행하기 위해 다음과 같이 두 개의 오류를 가정하자.

첫째, 오류는 잘못된 출력 값 num을 발생하는 *incre* 메소드에 있다. *incre* 메소드에서 문장 '*num = num - 1*'은 오류이고 문장 '*num = num + 1*'이 올바른 문장이다.

둘째, 오류는 잘못된 출력 값 var2를 발생하는 *totalsum* 메소드에 있다. *totalsum* 메소드에서 문장 '*var2 = var1 - num*'은 오류이고 '*var2 = var1 + num*'이 올바른 문장이다.

사용자는 9와 14의 실행 결과를 기대하였으나, 5와 0을 얻었다. 따라서 예제 프로그램을 디버깅하기 위해 (그림 9)의 실행 트리를 생성한다.



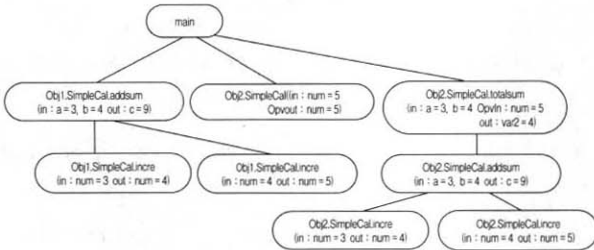
(그림 9) (그림 3)과 (그림 8)의 프로그램에 대한 실행 트리

알고리즈믹 디버깅 기술을 이용해서 (그림 9)의 실행 트리를 탐색하는 과정은 다음과 같다.

Obj1.SimpleCal (in : num = 0 OpvOut : num = 0)?
$ correct
Obj1.SimpleCal.addsum (in : a = 3, b = 4 out : c = 5)?
$ incorrect
Obj1.SimpleCal.incre (in : num = 3 out : num = 2)?
$ incorrect
An error has been localized inside the body of method incre(int) in SimpleCal class

따라서 사용자는 오류가 포함된 *incre* 메소드로부터 오류

가 있는 문장 '*num = num-1*'을 발견하고 '*num = num + 1*'로 수정할 수 있다. 그리고 사용자는 다시 자바 프로그램을 컴파일하고 실행하여 실행 트리를 생성한다. 이때 이전 단계에서 사용자가 correct라고 응답한 단말 노드로부터 위쪽으로 incorrect라고 응답한 노드 전까지는 오류가 없는 노드이기 때문에 (그림 10)에서 제거되었다.

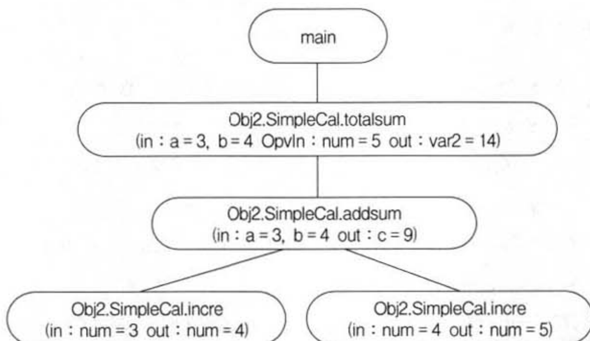


(그림 10) *incre* 메소드 내의 첫 번째 오류를 수정 후 생성한 재실행 트리

생성된 실행 트리로부터 디버깅하는 과정은 다음과 같다.

Obj1.SimpleCal.addsum (in : a = 3, b = 4 out : c = 9)?
$ correct
Obj2.SimpleCal (in : num = 5 OpvOut : num = 5)?
$ correct
Obj2.SimpleCal.totalsum (in : a = 3, b = 4 OpvIn : num = 5 out : var2 = 4)?
$ incorrect
Obj2.SimpleCal.addsum (in : a = 3, b = 4 out : c = 9)?
$ correct
An error has been localized inside the body of method totalsum(int, int) in SimpleCal class

탐색 결과, *totalsum* 메소드 내에 오류가 포함되어 있는 것을 알 수 있다. 따라서 사용자는 "*var2 = var1 - num*"을 "*var2 = var1 + num*"로 수정할 수 있다.



(그림 11) *totalsum* 메소드내의 두 번째 오류를 수정한 후 생성한 재실행 트리

사용자는 다시 자바 프로그램을 컴파일하고 실행해서 실행 트리를 생성하고 다시 노드를 탐색한다.

Obj2.SimpleCal.totalsum (in : a = 3, b = 4 OpvIn : num = 5 out : var2 = 14)?
$ correct
There are no errors.

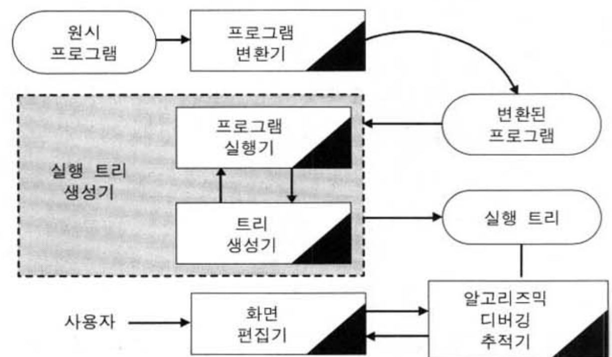
이 결과로 사용자는 더 이상 논리적인 오류가 없음을 알 수 있고 두 개 논리적인 오류를 수정할 수 있다. 사용자는 단지 메소드의 입력 값과 출력 값이 올바른지 아닌지를 결정하고, 이 정보로부터 알고리즈믹 디버깅 시스템은 논리적인 오류를 자동으로 발견한다.

5. 현재 알고리즈믹 디거깅 시스템의 구현 및 평가

본 장에서는 자바 프로그램에서 논리적인 오류를 발견하는 알고리즈믹 디버깅 기술의 구현 상황을 서술하고, 전통적인 순차적 디버깅 기술과 알고리즈믹 디버깅 기술을 비교한다.

5.1 구 현

자바 프로그램의 논리적인 오류를 발견하는 알고리즈믹 디버깅 시스템의 구현은 (그림 12)와 같이 프로그램 변환기(program translator), 실행 트리 생성기(execution tree generator), 알고리즈믹 디버깅 추적기(algorithmic debugging tracer), 그리고 화면 편집기(editor view)로 나누어 구현하였다.



(그림 12) 알고리즈믹 디버깅 시스템의 기능적인 구조

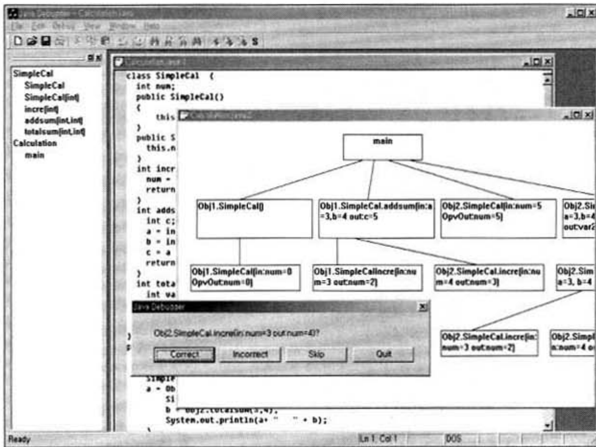
프로그램 변환기는 원시 프로그램을 입력받아 상속관계와 프로그램의 자료흐름과 제어흐름을 기반으로 멤버 변수의 입/출력 정보로부터 부작용을 고려한 변형된 프로그램과 이들 정보를 저장하고 있는 심볼 테이블(symbol table)을 생성한다.

실행 트리 생성기는 프로그램 실행기(program executor)와 트리 생성기(tree generator)로 구성된다. 프로그램 실행기는 코드 생성과 가상 기계로 구성되며 프로그램 변환기에 의해 변형된 프로그램을 가상 기계에서 수행할 수 있는 코드로 생성하고 가상 기계에서 실행한다. 트리 생성기는 가상 기계에서 한 개의 생성자 또는 메소드를 실행할 때마다 실행 결과를 한 개의 노드로 만들어 트리를 구성한다.

현재 프로그램 변환기과 실행 트리 생성기의 프로그램 실행기는 [16]의 컴파일러 시스템을 기반으로 구현되었다. 이는 변형된 프로그램을 스택 기반 가상 기계에서 수행 가능한 기계 코드로 변환한다. 그리고 가상 기계에서 기계 코드를 실행하고 실행 결과는 정보 테이블(information table)에 저장된다. 정보 테이블은 현재 실행 중인 객체 이름, 클래스 이름, 변수 이름과 값을 저장하고 있다. 트리 생성기는 정보 테이블에 저장된 정보를 이용하여 메소드와 생성자를 노드로 만들어 실행 순서에 따라 하향식으로 실행 트리를 생성한다. 따라서 생성된 실행 트리는 프로그램의 *main* 메소드가 실행 트리의 루트 노드가 되고 실행 순서에 따라 왼쪽부터 오른쪽으로 자식 노드와 형제 노드로 구성된다. 이러한 실행 트리는 프로그램의 실행 결과에 대한 실행 순서의 정보를 가지고 있다.

따라서 알고리즈믹 디버깅 추적기는 실행 트리를 탐색하고 사용자에게 질문하고 사용자의 correct 또는 incorrect의 응답에 의해 동작한다. 이 추적기는 2.2절에서 제시한 (알고리즘 1)에 의해 구현되었다.

프로그램 디버깅을 위한 편집기는 사용자가 편리하게 디버깅할 수 있도록 (그림 13)과 같이 윈도우즈 환경 하에서 개발하였다.



(그림 13) 자바 프로그램을 위한 알고리즘 디버깅 시스템의 그래픽 사용자 인터페이스

(그림 13)의 화면 편집기는 클래스와 메소드의 정보를 보여주는 윈도우와 프로그램 편집기가 있으며 알고리즈믹 디버깅 시스템이 사용자에게 질문하는 메시지 상자를 보여주고 있다.

5.2 평 가

본 논문에서 평가 기준은 사용자가 디버깅 시스템에 접근하는 횟수로 하였다. 그리고 기존의 순차적 디버깅 기술 두 가지와 알고리즈믹 디버깅 기술을 비교하였다. 순차적 디버깅 기술로 디버깅하는 방법은 사용자마다 다양하겠지만 일반적인 두 가지 방법으로 정의하여 실험하였다. 첫 번째 순차적 디버깅 방법(SWD1)은 *main* 메소드에서 step-over를 이용하여 메소드의 반환값을 확인하면서 오류가 포함된 메소드를 찾는다. 그리고 디버깅을 다시 시작해서 break-point와 명령어 go를 사용하거나 step-over와 step-into 명령어를 사용해서 그 메소드까지 이동한 후 그 메소드 내에서 오류를 발견하거나 호출 메소드가 있을 경우 앞의 방법을 반복해서 논리적인 오류를 발견한다. 4 장의 예제 프로그램인 경우 디버깅을 시작해서 'a = Obj1.SimpleCal.addsum(3, 4) ;' 문장을 실행하면 *addsum* 메소드 내에 오류가 있음을 알 수 있다. 그러나 현재 방법은 그 문장을 벌써 실행했기 때문에 디버깅을 처음부터 다시 시작해야만 한다. 그리고 *addsum* 메소드 내의 문장들을 디버깅하기 위해 break-point와 명령어 go를 사용하여 그 메소드까지 이동하고 step-over 명령어를 사용해서 문장을 디버깅한다. 두 번째 순차적 디버깅 방법(SWD2)은 최악의 경우 처음부터 실행되는 경로의 모든 문장을 실행하면서 디버깅한다. 평가를 위한 예제 프로그램은 <표 1>을 사용하였다.

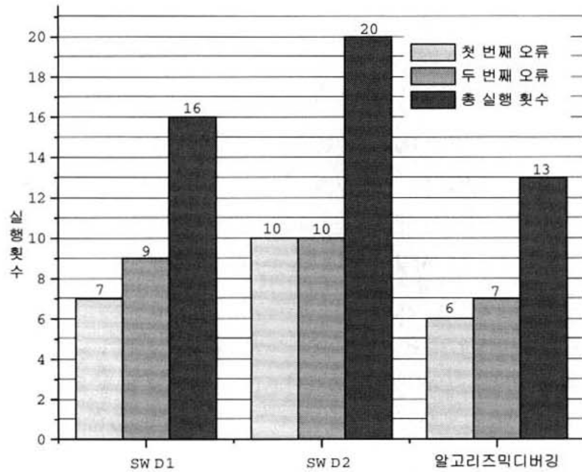
〈표 1〉 평가를 위한 예제 프로그램의 구성

	메소드(생성자) 수	논리적인 오류 수	실행 트리의 노드 수
4 장의 예	6	2	9
Bubble Sort	6	1	17
Calculator	10	2	29

<표 1>에서 Bubble Sort 프로그램은 입력 값 '5, 2, 3, 1, 9, 8'으로 실행하고, 생성된 실행 트리의 노드 수는 17개였다. 그리고 Calculator 프로그램은 재귀 하강 파서(recursive descent parser)를 이용하여 산술 연산을 하는 계산기 프로그램으로 입력 값 '2 + 3 * 4'로 실행하고, 생성된 실행 트리의 노드 수는 29개였다. 그리고 Bubble Sort는 논리적인 오류가 한개 존재하고, Calculator는 두 개가 존재하도록 하여 평가하였다.

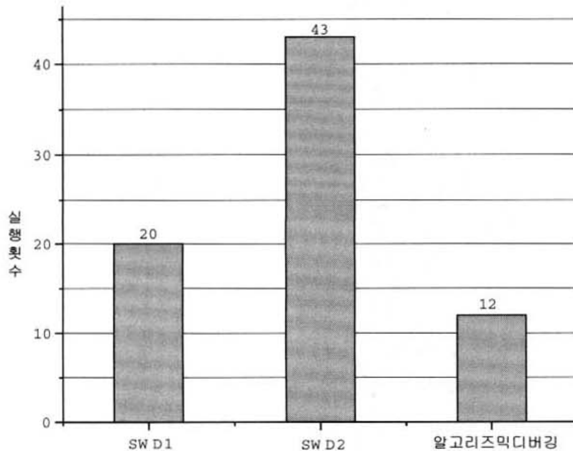
이 평가는 사용자마다 약간의 오차는 발생할 수 있으나

연산에 관련된 문장을 디버깅하는 횟수만 포함했으며 순차적 디버깅 기술의 경우 사용자가 프로그램을 재실행하는 횟수와 break-point를 설정하는 횟수는 제외하였다. 알고리즈믹 디버깅 기술의 경우에는 논리적인 오류를 포함하고 있는 메소드 내의 문장을 탐색하는 횟수를 순차적 디버깅 기술과 동일하게 적용하였다.



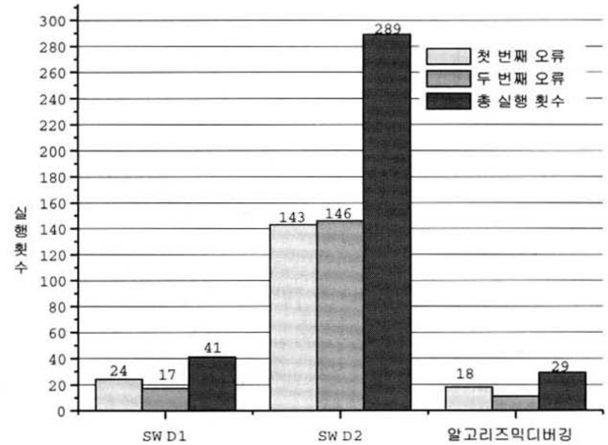
(그림 14) 4장의 예제 프로그램에 대한 디버깅 결과

(그림 14)는 4장의 테스트 프로그램에 대한 디버깅한 결과를 비교한 그래프로 첫 번째 오류와 두 번째 오류를 발견하기까지의 디버깅 횟수와 총 실행 횟수를 비교하였다. 그 결과 알고리즈믹 디버깅 기술이 순차적 디버깅 기술 두 가지 보다 효율적임을 알 수 있다.



(그림 15) Bubble Sort 프로그램에 대한 디버깅 결과

(그림 15)는 Bubble Sort 프로그램에 포함된 한 개의 오류를 수정하기 위해서 사용자가 디버깅 시스템에 접근한 횟수를 비교한 그래프이다. (그림 14)의 예제 프로그램보다 메소드의 호출과 문장의 수가 증가하여 순차적 디버깅 기술의 디버깅 횟수가 알고리즈믹 디버깅 기술보다 증가하였다.



(그림 16) Calculator 프로그램에 대한 디버깅 결과

(그림 16)의 Calculator 프로그램은 메소드 호출이 많고, 실행 트리의 레벨이 크다. 따라서 SWD2는 다른 두 예제 프로그램에 비해서 디버깅 횟수가 매우 증가하였음을 알 수 있다. SWD1는 알고리즈믹 디버깅 기술보다 실행 횟수가 약간 많지만, 프로그램을 재실행하는 횟수와 break-point를 사용하는 횟수가 많기 때문에 비효율적이다.

<표 1>의 세 가지 예제 프로그램에 대한 테스트 결과 자바 프로그램에 포함된 논리적인 오류를 발견하는데 본 논문에서 적용한 알고리즈믹 디버깅 기술이 전통적인 순차적 디버깅 기술보다 사용자가 디버깅 시스템에 접근하는 횟수가 적음을 알 수 있다. 따라서 알고리즈믹 디버깅 기술은 메소드의 수가 증가할수록 전통적인 순차적 디버깅 기술보다 사용자의 디버깅 횟수를 감소시킬 수 있고, 자바 프로그램을 디버깅하기 위해서 효과적으로 적용될 수 있다.

6. 결 론

현재 자바 프로그램 내에 포함되어 있는 논리적인 오류를 디버깅하는 기술은 절차지향 언어에서 사용하던 전통적인 방법을 사용하고 있다. 그러나 다양하고 복잡한 자바 프로그램을 디버깅하는 일은 쉽지 않고 객체지향 프로그램에는 적당하지 않다.

본 논문에서는 객체지향 프로그램인 자바 프로그램의 논리적인 오류를 디버깅하기 위한 방법으로 알고리즈믹 디버깅 기술을 시도하였다. 알고리즈믹 디버깅 기술은 실행 트리를 구성하고 이 트리를 탐색하여 프로그램 내에 포함된 논리적인 오류를 발견하는 반자동화 디버깅 기술이다. 따라서 알고리즈믹 디버깅 기술에서 실행 트리를 생성하는 것은 매우 중요하다. 이 기술은 지금까지 논리형 언어, 명령

형 언어, 함수형 언어에 적용하는 방법이 제안되었으나 객체지향 언어에 적용된 경우는 처음이었다.

본 논문에서는 자바 프로그램에서 실행 트리를 생성하는 방법과 자바 프로그램의 탐색 과정을 서술하였다. 트리를 생성하기 위해서 매개변수, 메소드, 생성자, 자바 API 그리고 멤버 변수와 상속과 다형성을 고려하였다. 먼저 사용자에게 좀 더 정확한 정보를 제공하기 위해 실행 트리의 구성에서 객체 이름과 클래스 이름을 추가하였다. 그리고 실행 트리의 노드는 함수 대신 각 객체에 대한 생성자를 포함해서 메소드를 기반으로 구성하였다. 또한 노드의 입/출력 변수와 값을 정의하였다. 변수는 매개변수와 멤버 변수로 구분하고 멤버 변수는 종류에 따라 구분하여 정의하였다. 값에 대한 자료형은 기본 자료형과 참조 자료형으로 구분하고 기본 자료형은 불린형(boolean), 정수형(byte, short, integer, long), 실수형(float, double), 문자형(char)으로 구분하여 값의 표현 구조를 정의하였다. 참조 자료형은 문자열 객체 변수, 배열 객체 변수, 클래스 객체 변수로 구분해서 값의 표현 구조를 정의하였다. 그리고 예제 프로그램을 통해서 자바를 위한 알고리즈믹 디버깅 기술을 확인하였다. 그 결과 자바 프로그램에 대한 데이터의 변화와 흐름을 실행 트리에서 정확히 표현할 수 있어 사용자에게 정확한 정보의 흐름을 제공할 수 있었다. 그리고 기존의 전통적인 순차적 디버깅 기술보다 사용자와 디버깅 시스템과의 상호작용 횟수를 줄일 수 있었다.

향후 연구 계획은 본 논문에서 제시한 실행 트리를 이용하여 자바 프로그램을 디버깅할 때, 기존의 알고리즈믹 디버깅 기술보다 효율적인 디버깅을 할 수 있는 확장된 디버깅 방법과 스레드 프로그램을 고려한 디버깅 방법을 연구하는 것이다.

참 고 문 헌

[1] M. Auguston, "A language for debugging automation," *In Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Knowledge Systems Institute, pp.108-115, June, 1994.

[2] P. Fritzson, M. Auguston and N. Shahmehri, "Using Assertions in Declarative and Operational Models of Automated Debugging," *The Journal of Systems and Software 25*, pp. 223-239, 1994.

[3] H. J. Kouh and W. H. Yoo, "Algorithmic Debugging in Java Programs," *ACIS Annual International Conference on Computer and Information Science (ICIS'02)*, August, 2002.

[4] G. Kovacs, F. Magyar and T. Gyimothy, "Static Slicing of JAVA Programs," *Research Group on Artificial Intelligence(RGAI)*, Hungarian Academy of Sciences, Jozsef Attila University, HUNGARY, December, 1996.

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

[6] C. Mateis, M. Stumptner and F. Wotawa, "Debugging of Java Programs using a model-based approach," *In Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, Scotland, 1999.

[7] C. Mateis, M. Stumptner and Franz Wotawa, "Locating bugs in java Programs-first results of the Java Diagnosis Experiments (Jade) project," *In Proceedings of IEA/AIE*, New Orleans, Springer-Verlag, 2000.

[8] M. Stumptner, D. Wieland and F. Wotawa, "Analysing models for software debugging," *Technische Universitat Wien Institut fur Informations systeme Database and Artificial Intelligence Group*, 2001.

[9] R. Lencevicius, "On-the-fly Query-Based Debugging with Examples," *In Proceeding of the Fourth International Workshop on Automated and Algorithmic Debugging, AADEBUG '2000*, Munich, Germany, August, 2000.

[10] P. Fritzson, N. Shahmehri, M. Kamkar and T. Gyimothy, "Generalized Algorithmic Debugging and Testing," *In Proceeding of the 1991 ACM SIGPLAN Conference*, Toronto, Canada, pp.317-326, June, 1991.

[11] E. Shapiro, *Algorithmic Program Debugging*, MIT Press, May, 1982.

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

[13] H. Nilsson and P. Fritzson, "Algorithmic debugging for lazy functional languages," *In Proceedings of PLILP'92 - Symposium on Programming Language Implementation and Logic Programming, Leuven*, Belgium, August, 1992. LNCS 631, Springer-Verlag, 1992.

[14] H. Nilsson and P. Fritzson, "Lazy Algorithmic Debugging : Ideas for Practical Implementation," *The First International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, 1993.

[15] J. Gosling, B. Joy and G. Steels, *Java Languages Specification*, Addison-Wesley, 1996.

[16] 고훈준, 유원희, "테스트 시스템을 위한 프로그래밍 언어와 컴파일러 설계", *한국정보과학회 논문지 : 컴퓨팅의 실제, 한국정보과학회*, 제8권 제3호, 2002.

고 훈 준

e-mail : hjkouh@hanmail.net
1998년 인하대학교 생물공학과(공학사)
2000년 인하대학교 전자계산공학과(공학석사)
2002년 인하대학교 전자계산공학과 박사과정 수료
관심분야 : 프로그래밍 언어, 컴파일러, 디버거, 생물 정보학 등

유 원 희

e-mail : whyoo@inha.ac.kr
1975년 서울대학교 공과대학 응용수학과
1978년 서울대학교 대학원 계산학 전공(이학석사)
1985년 서울대학교 대학원 계산학 전공(이학박사)
1992년~1993년 University of California, Irvine 객원연구원
1979년~현재 인하대학교 전자계산공학과 교수
관심분야 : 프로그래밍 언어, 컴파일러, 실시간 시스템, 병렬시스템 등