자바 프로그램을 위한 효율적인 디버깅 방법
A Efficient Debugging Method for Java Programs

고 훈 준* 양 창 모** 유 원 희*
(Hoon-Joon Koun) (Chang-Mo Yang) (Weon-Hee Yoo)

요약 자바 언어는 다양한 플랫폼과 다양한 분야에서 사용되고 있는 대표적인 객체지향 언어이다. 자바 언어는 객체지향의 특징 때문에 전통적인 절차지향 언어 보다 프로그램의 구조가 단순하다. 그러나 복잡한 자바 프로그램을 디버깅하는 일은 쉽지 않다. 디버깅은 항상 소프트웨어 발전의 많은 비용이 드는 부분이다. 자바 프로그램의 구문 오류는 현재 디버깅 시스템에 의해서 쉽게 발견된다. 그러나 자바 프로그램에 포함된 논리적인 오류는 발견하기가 어렵다. 자바 프로그램을 위한 기존의 디버깅 기술은 절차지향언어에서 사용하는 순차적인 방법을 사용하고 있다. 불행히도, 이 전통적인 방법들은 종종 특별한 프로그램의 오류를 찾는데 적당하지 않다. 이는 프로그램의 크기가 커지고 복잡해짐에 따라 디버깅하는데 걸리는 시간이 프로그램을 개발하는 시간의 많은 부분을 차지하게 된다. 디버거 사용자가 자바 프로그램 내에 포함되어 있는 오류를 쉽게 찾아내는 일은 효율적인 소프트웨어 개발에서 매우 중요하다.
본 논문에서는 사용자가 자바프로그램을 좀 더 빠르게 디버깅을 할 수 있도록 알고리즈믹 디버깅 방법을 적용한다. 알고리즈믹 디버깅 방법은 함수의 호출관계를 실행 트리로 구성하고 이를 검사하여 에러가 포함되어 있는 함수를 발견하는 방법이다. 따라서 기존의 순차적인 방법보다 디버깅하는 횟수를 줄일 수 있다.

Abstract Java language is a representative object-oriented language that is used at the various platform and fields. A structure of java language is simpler than traditional procedural-oriented language because of characters of object-oriented language But it is difficult to debug complicated java programs. Debugging has always been a costly part of software development. Syntax errors of java programs is easily found by the current debugging system. But it is difficult to locate logical errors included in java programs. Traditional debugging techniques locating logical errors in java program have been still used with conventional methods that are used at procedural-oriented languages. Unfortunately, these traditional methods are often inadequate for the task of isolating specific program errors. Debugger users may spend considerable time debugging code of program development with sequential methods according as program size is large and is complicated. It is important to easily locate errors included in java program in the software development.

In this paper, we apply algorithmic debugging method that debugger user can easily debug programs to java program. This method executes a program and makes an execution tree from calling relation of functions. And it locates errors at the execution tree. So, Algorithmic debugging method can reduce the number of debugging than conventional sequential method.

1. 서 론

자바 언어는 다양한 플랫폼과 다양한 분야에서 사용되고 있는 대표적인 객체지향 프로그래밍 언어(object-oriented language)이다[4]. 자바 언어는 객체지향의 특징 때문에 전통적인 절차 지향 언어(traditional procedural-oriented language)보다 프로그램이 매우 단순하다. 그러나 복잡한 자바 프로그램을 디버깅하는 일은 쉽지 않다. 현재 자바 프로그램은 JDK(java development kit)를 사용하거나 J Builder, Visual Cafe 등과 같은 비쥬얼 도구를 사용하여 프로그램을 개발하고 있다. 그리고 프로그램의 개발 효율과 개발 속도를 위해 다양한 개발 도구들이 현재 개발되고 있다.

자바 프로그램의 구문 오류는 현재 디버깅 시스템에 의해서 쉽게 발견된다. 그러나 이와 같은 개발 도구에서 자바 프로그램의 논리적인 오류(logical errors)를 발견하는 디버깅 기술은 어렵다. 여전히 기존의 절차지향 언어에서 사용하는 순차적인 방법(sequential method)을 사용하고 있다. 객체지향 프로그램을 디버깅하는 것은 절차지향 프로그램을 디버깅하는 것과 같지 않다. 자바 프로그램을 개발하는 초보자와 전문가들은 프로그램의 크기가 커지고 복잡해짐에 따라 프로그램을 개발하는 시간보다 디버깅하는데 걸리는 시간의 비율이 증가하고 있다. 따라서 사용자가 효율적으로 프로그램을 디버깅하는 기술은 매우 중요하다[1].

지금까지 많은 디버깅 기술들이 연구되어 있다[1,3]. 그 중에서 하향식 방법(top-down method)으로 디버깅을 하는 대표적인 기술로는 알고리즈

이 논문은 2001학년도 인하대학교의 지원에 의하여 연구되었음(INHA-22000)

* 인하대학교 전자계산공학과
** 청주교육대학교 컴퓨터공학과

믹 디버깅 방법(algorithmic debugging method)이 있다[2,5,6,7]. 알고리즈믹 디버깅 방법은 Shapiro[6,7]에 의해 제안되어 논리 언어에 적용되었다. 그 후 알고리즈믹 디버깅 기술은 많은 사람들에 의해 C, Pascal 과 같은 명령형 언어와 Hakell과 같은 함수언어에 적용되어 왔다.

본 논문에서는 순수 객체지향 언어인 자바 프로그램에 포함되어 있는 논리적인 오류를 좀 더 쉽게 발견하기 위해 알고리즈믹 디버깅 방법을 적용한다. 자바프로그램을 알고리즈믹 디버깅 방법에 적용하기 위해서 본 논문에서는 생성자(constructor), 메소드(method)를 고려한다. 이 방법은 많은 사용자 정의 클래스와 메소드로 구성된 자바 프로그램의 경우 매우 효율적인 디버깅이 가능하다.

2. 배 경

이 절에서는 자바 프로그램에서 사용하고 있는 전통적인 디버깅 방법과 알고리즈믹 디버깅 방법을 서술한다.

2.1 전통적인 디버깅 방법

일반적으로 자바 프로그램의 논리적인 오류를 디버깅하는 방법은 JDK에 포함되어 있는 JDB 디버깅 도구를 사용하거나 J-builder, Visual cafe와 같은 자동화 도구에 있는 디버깅 도구를 사용한다. 이들 도구들은 자바 프로그램 내에 포함되어 있는 논리적인 오류를 찾기 위해 기존의 절차 지향 언어에서 사용하던 방법을 사용하고 있다. 첫째는 소스 프로그램을 직접 분석하거나 논리적인 오류가 의심되는 위치에 "*System.out.println()*"과 같이 화면에 값을 출력하는 명령어를 삽입하여 디버깅을 한다. 그러나 이 방법은 디버거 사용자가 에러의 위치를 정확히 예상하기가 어렵기 때문에 프로그램을 이해하고 디버깅하는 것은 어렵다. 둘째는 "step-over", "step-into", "go", "break-point" 명령어를 이용하여 프로그램의 각 문장을 조사하고 오류가 포함된 문장을 찾는다. 그러나 프로그램의 크기가 커지고 복잡해짐에 따라 수많은 클래스와 그 클래스 내에 있는 생성자와 메소드들을 순차적으로 디버깅하는 시간은 기하급수적으로 증가한다.

이와 같은 방법들은 프로그램의 크기가 커질수록 디버거 사용자에게 너무 많은 시간과 예측 그리고 실행 작업을 요구하기 때문에 효율적인 디버깅 작업이 어렵다.

2.2 알고리즈믹 디버깅 방법

알고리즈믹 디버깅 방법은 모든 함수의 호출관계를 입력 값과 출력 값으로 실행 트리(execution tree)로 구성한다. 그리고 그 실행 트리의 상위 레벨부터 하위 레벨로 탐색하면서 오류를 발견하는 방법이다. 이 방법은 Shapiro에 의해 처음으로 제안된 소프트웨어 디버깅 기술로서 프롤로그(prolog) 언어에서 처음으로 제안되었다.

이 방법은 프로그램을 실행하고 함수 이름 또는 입력/출력 、매개변수의 값 등, 추적 정보(trace information)를 저장하는 함수 수준의 실행 트리를 생성한다. 실행 트리는 함수를 노드로 만들어 프로그램의 실행 순서에 따라 루트로부터 하향식으로 구성된다. 그리고 호출된 함수는 자식 노드가 되고 왼쪽에서부터 오른쪽으로 구성된다.

알고리즈믹 디버깅 방법은 실행 트리의 루트부터 하향식 방법으로 추적하고 각 함수에서 기대할 수 있는 행동에 대해 사용자에게 질문함으로써 동작한다. 사용자는 각 함수의 행동에 대해 "예" 또는 "아니오"로 대답할 수 있다. "예"라고 대답할 경우에는 현재 함수와 그의 하부 자식 함수에 오류가 없다는 뜻이고 더 이상 자식 함수로 추적하지 않고 형제 함수 노드로 이동하여 추적한다. "아니오"라고 대답할 경우에는 현재 노드와 그의 자식 노드 중에 오류가 있다고 보고 자식 노드로 이동하여 질문한다. 추적은 더 이상 자식 노드가 없거나 사용자가 "예"라는 대답을 할 때까지 진행된다. 프로그램에 오류가 발견되면 이 오류는 자신 함수에 존재하거나 이 함수를 호출한 상위 레벨의 함수에 오류가 있다고 판단하여 오류를 찾아낸다. 따라서 검색이 끝나면 오류는 다음과 같은 경우 중 한 가지일 때 함수 p안에 발생한다.

첫째, 함수 p는 함수 호출이 없다.
둘째, 함수 p의 몸체로부터 수행되는 모든 함수 호출은 사용자의 기대를 만족한다.

디버거의 출력은 입력 매개변수와 기대되는 결과에 의해 사용자의 대답으로 생성되고 오류는 어떤 함수 몸체 안에서 발견된다.

알고리즈믹 디버깅 이론의 예는 [그림 1]과 같다. 입력 매개변수 a와 c, 출력 매개변수 b와 d를 가지는 함수 p를 생각해보자.

변수 b의 값은 함수 q를 호출함으로서 계산되고 변수 d는 함수 r을 호출함으로서 계산된다. 프로

그램을 실행하기 위해 다음을 가정하자.

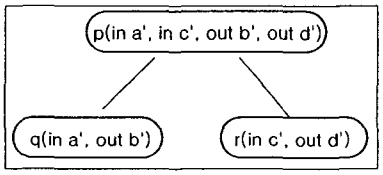
첫째, 입력 값 *a'*와 *c'*를 갖는 함수 *p*는 출력 값 *b'* 와 *d'*를 반환한다.

둘째, 오류는 잘못된 출력 값 *d'*가 발생하는 함수 *r*에 있다.

```
function p(a, c: integer; b, d: integer) {
        q(a, b);    r(c, d);
}
function q(a: integer; b: integer) {
        • • •
}
function r(c: integer; d: integer) {
        • • •
}
```

[그림 1] 함수 *q*와 *r*을 호출하는 함수 *p*의 예

디버깅 시스템이 [그림 1]의 프로그램을 실행하면 [그림 2]와 같이 함수 단위로 구성된 실행 트리가 생성된다. 각 노드에는 함수의 이름, 입력 값, 출력 값이 표시된다.



[그림 2] 실행 트리

[그림 2]의 실행 트리를 가지고 동작하는 알고리즈믹 디버깅 시스템과 디버거 사용자와의 대화는 다음과 같다.

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

이탤릭체는 디버거 시스템이 디버거 사용자에게 질문하는 문장이다. 굵은 글씨체는 디버깅 시스템의 출력을 나타내고, '$'에 의해 표현되는 것은 사용자의 대답을 나타낸다.

이와 같은 디버깅 시스템은 사용자가 전체 프로그램을 정확히 이해하지 못한 경우에도 함수 단위로 입력과 출력 값을 예상하여 답변함으로써 디버깅을 할 수 있기 때문에 기존 일반적인 디버깅 방법에서 디버깅 시스템이 사용자에게 요구하는 부담이 줄어든다.

3. 자바프로그램을 위한 디버깅 방법

이 절에서는 자바프로그램을 알고리즈믹 디버깅 방법으로 디버깅하기 위해 고려해야하는 사항들을 서술한다. 알고리즈믹 디버깅 방법은 함수의 호출관계를 트리로 나타내는 방법이기 때문에 객체지향 언어에서는 생성자와 메소드의 호출관계를 실행 트리로 구성한다.

3.1 생성자

생성자는 클래스로부터 객체를 생성할 때 단 한번만 실행되는 메소드로 주로 초기화 기능을 위해 사용된다. 따라서 생성자에서 left-hand side의 변수는 모두 출력변수로 가정한다. 다음 프로그램의 예를 보자.

```
class SimpleCal  {
        int num1 ,num2;
        public SimpleCal() {
                this(0,0);
        }
        public SimpleCal(int num1, int num2) {
                this.num1=num1;
                this.num2=num2;
        }
}
```

[그림 3] *SimpleCal* 클래스에서 생성자의 예

[그림 3]과 같이 *SimpleCal* 클래스에는 *SimpleCal* 생성자가 두 개 있다. 오버로딩 생성자이다. 첫 번째 생성자는 매개변수가 없고 두 번째 생성자는 두개의 매개변수를 입력받아 두 멤버 변수에 할당한다. 이 클래스로부터 S1, S2 객체를 선언과 생성하는 명령은 다음과 같다.

SimpleCal S1 = new SimpleCal();

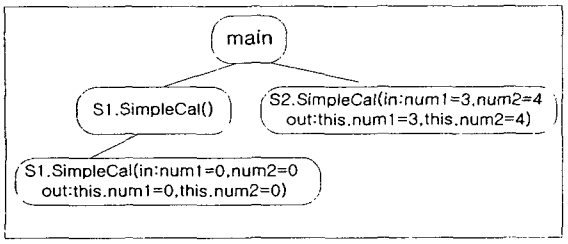
SimpleCal S2 = new SimpleCal(3, 4);
알고리즈믹 디버깅을 위해 S1 객체가 생성되는 과정의 표현은 다음과 같다.

S1.SimpleCal();
S1.SimpleCal(in:num1=0, num2=0
out: this.num1=0, this.num2=0);

키워드 *this*는 입력 값 0 과 0으로 두 개의 매개변수를 가지는 생성자를 호출한다. 출력변수는 할당받는 left-hand side 변수 모두이다. S2 객체가 생성되는 과정은 다음과 같이 표현된다.

S2.SimpleCal(in:num1=3, num2=4
out:this.num1=3, this.num2=4);

위의 결과로부터 실행 트리를 구성하면 [그림 4]와 같다.



[그림 4] *SimpleCal* 생성자로부터 생성한 실행 트리

실행 트리는 프로그램의 실행 순서에 따라 main 메소드로부터 생성자들이 계층구조를 이룬다.

3.2 메소드

메소드는 클래스 내에서 객체가 할 수 있는 행동을 정의한 것으로, 기존의 절차지향 언어에서 사용하는 프로시저/함수의 형식과 같다. 메소드는 실행한 결과 값을 돌려주기 위해 return 문을 사용한다.

SimpleCal 클래스에서 메소드는 [그림 5]와 같이 *incre*와 *addsum* 메소드를 정의할 수 있다. *addsum* 메소드는 두 개의 정수를 입력받아 덧셈을 하고 그 결과 값을 돌려주는 메소드이고, *incre* 메소드는 한 개 정수를 입력받아 1을 증가시키고 그 결과 값을 돌려주는 메소드이다.

```
class SimpleCal  {
    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;
    }
}
```

[그림 5] *SimpleCal* 클래스에서 메소드의 예

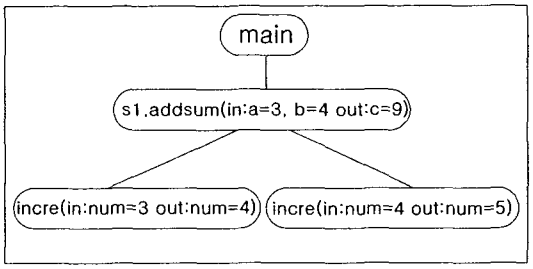
s1 객체가 *addsum* 메소드를 호출하고, 그 결과를 *sum* 변수에 할당한다고 가정하면 그 표현식은 다음과 같다.

sum = s1.addsum(3,4);

s1 객체에서 *addsum*메소드를 호출한 후 프로그램이 실행된 결과는 다음과 같다.

s1.addsum(in:a=3, b=4, out:c=9);
s1.incre(in:num=3 out:num=4);
s1.incre(in:num=4 out:num=5);

매개변수는 in과 out으로 구분하여 표현되고, 값이 표현된다. 위의 실행 결과로부터 실행 트리를 구성하면 [그림 6]과 같다.



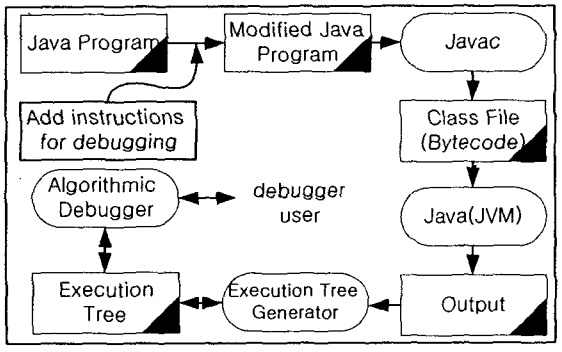
[그림 6] s1 객체가 입력 값 3, 4로 *addsum* 메소드를 호출하여 실행한 결과로 구성된 실행 트리

4. 시스템 구조와 테스트

이 절에서는 자바프로그램에서 알고리즈믹 디버깅을 실행하기 위한 시스템 구조를 서술한다. 그리고 예제를 가지고 디버깅 방법을 설명한다.

4.1 알고리즈믹 디버깅 시스템 구조

알고리즈믹 디버깅 방법으로 자바 프로그램을 디버깅하기 위해 [그림 7]과 같이 설계한다.



[그림 7] 자바 프로그램을 디버깅하기 위한 알고리즈믹 디버깅 시스템의 구조

[그림 7]에서와 같이 컴파일러와 가상머신은 JDK를 사용한다. 알고리즈믹 디버깅 방법은 입력값과 출력값을 기반으로 한다. 따라서 알고리즈믹 디버깅을 하기 위해서는 생성자와 메소드의 이름과 입력 변수와 출력 변수의 값이 요구된다. 따라서 자바 소스 프로그램을 *"javac"*로 컴파일하기 전에 디버깅을 위한 코드를 삽입한다. 수정된 자바 프로그램은 *"javac"*에 의해서 컴파일되고 *"java"* 명령어에 의해 실행된다.

본 논문에서 제안된 시스템은 프로그램의 실행 순서에 따라서 출력 데이터를 가지고 실행 트리를 만든다. 그리고 디버거 사용자는 알고리즈믹 디버거를 가지고 실행 트리로부터 자바 프로그램을 디버깅한다.

4.2 테스트

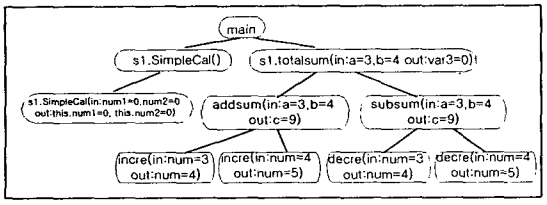
[그림 8]은 두 개의 클래스 *SimpleCal* 과 *Calculation*으로 구성된 자바 프로그램이다. *SimpleCal* 클래스는 두 개의 생성자와 다섯 개의 메소드로 구성된다. 그리고 *Calculation* 클래스는 프로그램이 시작하는 메인 메소드를 나타낸다.

```
class SimpleCal  {
    int num1 ,num2;
    public SimpleCal()
    {     this(0,0);     }
    public SimpleCal(int num1, int num2)  {
          this.num1=num1;      this.num2=num2;
    }
    int decre(int num) {
         num = num +1;      return 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 subsum(int a, int b)  {
          int c;
          a = decre(a);          b = decre(b);
          c = a + b;             return c;
      }
      int totalsum(int a, int b) {
          int var1, var2, var3;
          var1=addsum(a,b);
         var2=subsum(a,b);
          var3= var2-var1;   return var3;
      }
}
public class Calculation  {
      public static void main(String args[]) {
          int a, b, d;
          a =  Integer.parseInt(args[0]);
          b = Integer.parseInt(args[1]);
          SimpleCal s1 = new SimpleCal();
          d = s1.totalsum(a,b);
      }
}
```

[그림 8] 자바 프로그램의 예

[그림 8]에서 *decre* 메소드와 *totalsum* 메소드에 각각 오류가 포함되어 있다고 가정하자. *decre* 메소드에서는 문장 *"num=num+1"*은 오류이고 문장 *"num=num-1"*은 올바른 문장이다. 그리고 *totalsum* 메소드에서는 문장 *"var3=var2-var1"*은 오류이고 *"var3=var2+var1"*은 올바른 문장이다.

[그림 8]의 자바 프로그램을 [그림 7]의 시스템에 의해 실행하면 [그림 9]와 같은 실행 트리가 생성된다.



[그림 9] [그림 8]의 프로그램에 대한 실행 트리

알고리즈믹 디버깅 방법을 사용해서 실행 트리를 탐색하는 과정은 다음과 같다.

S1.SimpleCal(in:num1=0, num2=0
out:num1=0, num2=0)?

$ 예

S1.totalsum(in:a=3, b=3 out:c=0)?

$ 아니오

addsum(in:a=3, b=4 out:c=9)?

$ 예

subsum(in:a=3, b=4 out:c=9)?

$ 아니오

decre(in:num=3 out:num=4)?

$ 아니오

An error has been localized inside the body of method decre

오류를 발견한 후 디버거 사용자는 *decre* 함수로부터 오류가 문장 *"num=num+1"*을 *"num=num-1"*로 수정할 수 있다. 그리고 사용자는 다시 자바 프로그램을 컴파일하여 실행하여 실행 트리를 생성한다. 생성된 실행 트리로부터 디버깅하는 과정은 다음과 같다.

S1.SimpleCal(in:num1=0, num2=0
out:num1=0, num2=0)?

$ 예

totalsum(in:3, 3 out:4)?

$ 아니오

addsum(in:3, 4 out:9)?

$ 예

subsum(in:3, 4 out:5)?

$ 아니오

An error has been localized inside the body of method totalsum

이 디버깅 결과로 *totalsum* 메소드 내에 오류가 포함되어 있는 것을 알 수 있다. 그리고 사용자는 *"var3=var2-var1"*을 *"var3=var2+var1"*로 쉽게 수정할 수 있다. 디버거 사용자는 다시 자바 프로그램을 컴파일하고 실행해서 실행 트리를 생성하고 알고리즈믹 디버깅을 다시 한번 수행한다.

S1.SimpleCal(in:0, 0,out:0, 0)?

$ 예

totalsum(in:3, 3 out:14)?

$ 예

There are no errors.

이 결과로 디버거 사용자는 더 이상 논리적인 오류가 없음을 알 수 있고 두 개 논리적인 오류를 수정할 수 있다. 디버거 사용자는 단지 메소드의 입력 값과 출력 값이 올바른지 아닌지를 결정함으로써 알고리즈믹 디버깅 시스템은 논리적인 오류를 자동으로 발견한다. 기존의 전통적인 순차적인 방법으로 디버깅 할 경우에는 사용자가 프로그램을 이해하고 실행하기 위해서는 각 문장들을 순차적으로 실행해야 하기 때문에 최소 20 번 이상의 실행을 해야한다. 그러나 알고리즈믹 디버깅 방법을 사용하면 자바 프로그램에서도 위 결과와 같이 단지 11번의 결정만으로 쉽게 두 개의 논리적인 오류를 발견할 수 있음을 알 수 있다. 클래스와 메소드, 그리고 문장의 수가 많을 수 록 더욱 효과적이다. 따라서 전통적인 디버깅 방법보다 알고리즈믹 디버깅 방법은 자바프로그램을 쉽게 디버깅 할 수 있다.

5. 결 론

자바 언어는 순수 객체지향 언어로 다양한 플랫폼과 분야에 적용되고 있다. 자바 언어는 객체지향의 특징 때문에 전통적인 절차 지향 언어보다 매우 단순하다. 그러나 다양하고 복잡한 자바 프로그램을 디버깅하는 일은 쉽지 않다. 현재 자바 프로그램 내에 포함되어 있는 논리적 오류를 디버깅하는 기술은 절차지향 언어에서 사용하던 전통적인 방법을 사용한다. 따라서 프로그램의 크기가 커지고 클래스와 메소드, 그리고 문장의 수가 많아질수록 디버깅하는 시간이 기하급수적으로 증가한다.

본 논문에서는 자바 프로그램을 좀 더 쉽게 디버깅하기 위한 방법으로 알고리즈믹 디버깅 방법을 제안하였다. 알고리즈믹 디버깅 방법은 모든 함수의 호출관계를 입력 값과 출력 값으로 실행트리로 구성하고 실행트리의 상위 레벨부터 하위 레벨로 탐색을 하면서 오류를 발견한다. 자바 프로그램에서는 함수 대신 각 객체에 대한 생성자와 메소드를 가지고 실행 트리를 구성하였다. 그리고 실행 트리를 생성하기 위해 자바 소스 프로그램에 디버깅을 위한 코드를 삽입하고 컴파일하였다. 그리고 실행해서 얻은 결과 값을 기반으로 실행트리를 구성하였다. 디버거 사용자는 단지 메소드의 입력 값과 출력 값이 올바른지 아닌지를 결정함으로써 알고리즈믹 디버깅 시스템은 논리적인 오류를 자동으로 발견하였다.

그 결과 기존 전통적인 디버깅 방법보다 사용자가 실행하는 횟수를 줄였다.

향후 연구과제는 혼합디버깅 방법과 slicing 기법을 가지고 자바프로그램을 디버깅하는 방법을 연구해야 할 것이다.

참고문헌

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

[2] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing," ACM LOPLAS -- *Letters of Programming Languages and Systems.* Vol. 1, No. 4, December 1992.

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

[4] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. Journal of Functional Programming, July 1994, 4(3):337-370,

[5] N. Shahmehri, "Gneralized Algorithmic Debugging", Ph.D. Thesis No 260, Dept. of Computer and Information Science, Linkoping University, S-58183 Linkoping, Sweden, 1991

[6] E. Shapiro, "Algorithmic Program Debugging," MIT Press, May 1982

[7] E. Shapiro, "Algorithmic Program Diagnosis," ACM, 1982