제22회 한국정보처리학회 추계학술발표대회 논문집 제11권 제2호 (2004. 11)

자바 프로그램의 논리적인 오류를 찾기 위한 HDTS 시스템의 설계

고훈준
경인여자대학 컴퓨터정보기술학부
e-mail : hjkouh@kic.ac.kr

Design of HDTS System for Locating Logical Errors in Java Programs

Hoon-Joon Kouh
School of Computer Information Technology, Kyungin Women's College

요 약

이전 논문은 자바 프로그램 내에 포함된 논리적인 오류를 발견하기 위해 HDTS 기술을 제안하였다. HDTS 는 알고리즈믹 프로그램 디버깅 기술, 단계적 프로그램 디버깅 기술, 그리고 프로그램 분할을 혼합하여 효율적으로 논리적인 오류가 포함된 프로그램을 디버깅하는 기술이다. 본 논문에서는 HDTS 기술을 구현하기 위한 HDTS 시스템을 설계한다.

1. 서론

컴퓨터 프로그램은 매우 복잡한 추상화 시스템이다. 프로그램의 크기가 증가할수록 추상화 시스템은 점점 더 복잡해지고 프로그래머가 그것을 이해하기에는 더욱 어려워진다. 그리고 프로그래머가 프로그램의 일부를 이해하지 못하는 것은 심각한 오류를 가져올 수 있다. 따라서 프로그램 디버깅은 소프트웨어 개발에 있어서 매우 중요해지고 있다[1,2].

일반적으로 프로그램 내에서 발생할 수 있는 오류는 구문 오류(syntax error), 실행시간 오류(runtime error), 논리적인 오류(logical error)가 있다. 구문 오류는 프로그램을 구문 분석할 때 컴파일러에 내장된 파서를 통해서 쉽게 발견될 수 있다. 실행시간 오류는 컴파일러의 타입시스템으로 정적 분석을 통해 해결할 수 있다. 그러나 논리적인 오류는 컴파일 시간에도 실행 시간에도 발견되기 어렵고, 일반적으로 프로그램 실행 후에 실행 결과를 기반으로 오류의 원인이 되는 변수를 찾아서 프로그램을 역추적 하거나 처음부터 단계적으로 실행하면서 논리적인 오류를 포함하고 있는 원시 프로그램을 분석해야 한다. 따라서 아직까지 프로그래머에게 많은 시간과 노력을 요구한다[2].

특히, 객체지향 프로그램은 객체지향의 특징들 때문에 절차지향 프로그램에서 디버깅 작업을 하는 것이 더 어렵다. 그래서 이전 논문에서 객체지향의 개념을 가지고 있는 자바 프로그램에서 논리적인 오류를 효율적으로 발견하고 수정하기 위한 기술로 HDTS(hybrid debugging technology with slicing)를 제안하였다[3].

HDTS 는 알고리즈믹 디버깅 기술(algorithmic debugging technology)[4,5,11], 단계적 프로그램 디버깅 기술(step-wise program debugging technology)[2,10], 프로그램 분할(program slicing)[6,7,8]을 혼합하여 프로그래머가 논리적인 오류를 발견하기 위해 디버깅에 참여하는 횟수를 감소시키는 기술이다.

본 논문에서는 자바 프로그램 내에 포함된 논리적인 오류를 발견할 수 있는 HDTS 기술을 구현하기 위한 HDTS 시스템을 설계한다. 먼저, HDTS 시스템에서 사용될 Java 의 부분 언어를 정의하고, HDTS 시스템은 프로그램 변환기, 실행 트리 생성기, HDTS 디버거, 그래픽 사용자 인터페이스로 구성하여 구현한다.

2. HDTS

HDTS 는 알고리즈믹 프로그램 디버깅 기술과 단계적 프로그램 기술을 혼합하여 프로그램을 디버깅하는 HDT(hybrid debugging technique)를 사용해서 프로그램을 디버깅 할 때 프로그램 분할 기술을 사용하여 디

버깅하는 횟수를 줄일 수 있는 기술이다.

[알고리즘 1] HDTS의 알고리즘

Input: Incorrect Program P
Output: Correct Program

```
1 procedure HDTS(P)
2 begin
 let t={(o1,c1,m1,in1,out1),(o2,c2,m2,in2,out2),...,(on,cn,mn,inn,outn)}
 be the top level trace nodes of (o0,c0,m0,in0,out0), 1≤i≤n
3 repeat
4    Build an execution tree from P
5    Slicing4
6    n := (o0,c0,m0,in0,out0)  //select a start node
7    debug := False
8    i := 1
9    if(Query((o0,c0,m0,in0,out0)) = correct) then
10      debug := True
11   else
12     if ∃t then
13       Slicing1
14       while i<n do
15        if(Query((oi,ci,mi,ini,outi)) = correct) then
   //let (oj,cj,mj,inj,outj) be the right sibling node of
(oi,ci,mi,ini,outi)
16           if (oj,cj,mj,inj,outj) <> NIL then
17             i := j
18             continue
19           else
20             Slicing2
21     Statement_debug(the parent node of (oi,ci,mi,ini,outi))
22           Slicing3
23         fi
24       fi
25       i := i + 1
26     od
27     Slicing2
28     Statement_debug((oi,ci,mi,ini,outi))
29   else
30     Slicing2
31     Statement_debug((o0,c0,m0,in0,out0))
32   fi
33 fi
34 until (debug = True)
35 write(" There are no errors" )
36 end
```

[알고리즘 1]에서 실행 트리의 탐색은 실행 트리의 마지막 노드까지 탐색하여 여러 개의 오류를 발견할 수 있으며, 오류가 없을 때까지 실행 트리를 재생성하여 탐색이 가능하다. Slicing1, Slicing3, Slicing4는 실행 트리를 디버깅할 때 불필요한 노드들을 제거시키는 방법이고, Slicing2는 오류가 포함된 메소드의 원시 프로그램에서 불필요한 문장들을 제거시키는 방법이다. 함수 `Statement_debug`는 문장들을 디버깅 하기 전에 정적 분할인 `Slicing2`를 수행한 후에 단계적 프로그램 디버깅 기술을 사용한다.

3. 프로그래밍 언어의 정의

본 장에서는 HDTS 시스템을 구현하기 위한 Java를 정의하였다. [그림 1]에서 정의한 추상구문은 Java의 부분 언어로 [9]의 언어 명세를 기반으로 정의하였다.

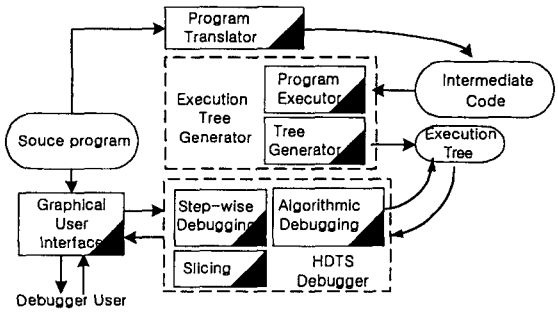
```
Program ::= Classes
Classes ::= Class Classes | ε
Class ::= ClassDecl | InterfaceDecl
ClassDecl ::= [public] class Id [extends Name] [implements
Name]{ FieldDecls }
InterfaceDecl ::= interface Id [extends Name ] { FieldDecls }
Name ::= Id | Id.Id
FieldDecls ::= FieldDecl FieldDecls | ε
FieldDecl ::= [public] MethodDecl | ConstructorDecl | VarDecl
MethodDecl ::= Type Id ( [ ParamList ] ); | Type Id ( [ ParamList ] )
{ StmntB }
ConstructorDecl ::= Id ( [ ParamList ] ) { StmntB }
VarDecl ::= [static | final] Type VarDeclarator ;
VarDeclarator ::= Var | Var, VarDeclarator
Var ::= Id [ = VarInitializer ] | Id [ ] [ = VarInitializer ]
Type ::=TypeSpecifier | TypeSpecifier [ ]
TypeSpecifier ::= boolean | int | float | char | void | Id
VarInitializer ::= Expr | { ArgList }
ParamList ::= Param | Param , ParamList
Param ::= TypeSpecifier Identifier | TypeSpecifier Identifier[ ]
ArgList ::= Expr | Expr , ArgList
StmntB ::= Stmnt StmntB | ε
Stmnt ::= VarDecl | Expr ; | { StmntB } | if ( Expr ) Stmnt [ else
Stmnt ] | while ( Expr ) Stmnt | return [ Expr] ; | break ; | continue;
Expr ::= Expr Operator Expr | Expr .Expr | Expr , Expr | Expr
        [ Expr ]
        | Operator Expr | Expr Operator | ( Expr ) | ( Type ) Expr
        | Expr ( [ ArgList ] ) | new Name ( [ ArgList ])
      | new TypeSpecifier [ Expr ] | true | false | null | super | this
        | Id | Num | Char
```

[그림 1] Java 부분 언어의 추상구문

[그림 1]의 추상구문은 Java의 완전한 언어를 포함하고 있지는 않지만, 본 논문에서 제안된 기술을 테스트하기 위한 기본형으로는 적당하다. 본 절에서 제시하는 Java의 추상 구문은 동적 결합, 상속, 그리고 인터페이스와 같은 객체 지향의 특징을 포함하고 있다. Operator는 배정 연산자, 산술 연산자, 관계 연산자, 논리 연산자, 그리고 증가·감소 연산자를 나타낸다. Id는 식별자, Num은 숫자를 나타낸다. 자료형은 정수형(int), 불린형(boolean), 실수형(float), 문자형(char)만 지원하고, 제어문에서 선택문은 if문을 지원하고, 반복문은 while문을 지원한다. switch문과 for문은 의미상으로 if문과 while문과 같기 때문에 [그림 1]의 추상구문에서 제외시켰다. 또한 접근 한정자도 제외시켰다. `main` 메소드와 메인 클래스는 public이고 나머지 클래스와 메소드는 모두 protected로 간주한다.

4. HDTS 시스템의 설계

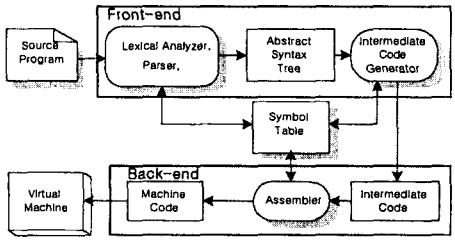
HDTS 시스템의 기능적인 구조는 [그림 2]과 같다. 전반적인 시스템 구조는 프로그램 변환기(program translator), 실행 트리 생성기(execution tree generator), HDTS 디버거, 그래픽 사용자 인터페이스(graphical user interface)로 구성된다.



[그림 2] HDTS 시스템의 기능적인 구조도

4.1 프로그램 변환기

프로그램 변환기는 [그림 3]의 구조도에서 컴파일러의 전위부(front-end)에 해당하는 부분이다.



[그림 3] 프로그램 변환기와 프로그램 실행기의 구조도

프로그램 변환기는 원시 프로그램을 입력 받아 어휘 분석과 구문 분석을 수행한다. 그리고 상속 관계와 프로그램의 자료 흐름과 제어 흐름을 기반으로 멤버 변수의 입/출력 정보를 저장하기 위해 AST(abstract syntax tree)와 이들 정보를 저장하고 있는 심볼 테이블(symbol table)을 생성한다. 그리고 AST로부터 중간 코드를 생성한다. 본 논문에서 구현한 어휘 분석과 구문 분석은 [그림 1]의 Java 추상 구문으로 WINDOWS용 Lex & Yacc의 자동화 도구를 사용하였다.

4.2 실행 트리 생성기

실행 트리 생성기는 중간 코드를 기계 코드(목적 코드)로 변환하여 프로그램을 실행하는 프로그램 실행기(program executor)와 프로그램 실행 결과로부터 노드를 구성하고 트리를 만드는 트리 생성기(tree generator)로 구분된다.

프로그램 실행기는 [그림 3]의 구조도에서 컴파일러의 후반부(back-end)와 가상 기계에 해당된다. 프로그램 실행기는 [그림 3]와 같이 중간 코드로부터 어셈블러에 의해 목적 코드(기계 코드)를 생성한다. 그리고 가상 기계에서 목적 코드를 실행한다. 목적 코드를 생성하는 어셈블러는 일반적인 two pass로 구현된다. one pass에서는 변수와 레이블의 주소 값을 레이블 테이블에 저장하고 two pass에서는 레이블 테이블을 참조하여 어셈블리 언어 코드를 목적 코드로 변환한다.

가상 기계는 스택 기반 가상 기계로 설계하였으며, 다음 기본적인 다섯 가지 전제 조건을 가지고 설계하였다.

첫째, 메모리는 두 개의 메모리로 분류하고 하나의 메모리는 부호 없는 문자형(unsigned char)을 가지는 일차원 배열로 코드 블록과 스택으로 구성한다. 그리고 다른 하나의 메모리는 힙(heap)으로 구성한다.

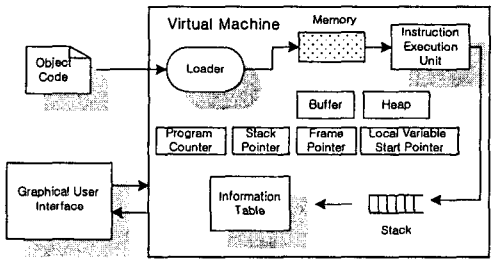
둘째, 기본 변수 타입은 문자형(char), 정수형(int), 실수형(float), 불린형(boolean)을 허용한다.

셋째, 실행 코드는 메모리 0 번지부터 저장하고, 스택을 사용할 때는 메모리의 끝에서부터 사용한다.

넷째, 가상 레지스터는 프로그램 카운터(program counter), 스택 포인터(stack pointer), 힙 포인터(heap pointer), 프레임 포인터(frame pointer)가 있고 메소드를 복귀할 때 값을 저장하는 버퍼(buffer)가 있다.

다섯째, 명령어는 1byte, 주소는 4byte, 정수는 32bit, 실수는 64bit로 구성된다.

이러한 다섯 가지 기본 전제 조건으로부터 설계된 스택 기반 가상 기계의 실행 구조는 [그림 4]와 같다. 가상 기계는 목적 코드를 메모리로 로드하고 스택을 사용하여 한번에 하나의 명령어를 수행하고, 실행 결과는 정보 테이블(information table)에 저장된다. 정보 테이블은 현재 실행 중인 객체 이름, 클래스 이름, 변수 타입, 변수 이름과 값 그리고 실행 순서 번호를 저장하고 있다.



[그림 4] 가상 기계의 실행 구조

트리 생성기는 프로그램 실행기의 가상 기계와 상호작용하며 가상 기계에서 한 개의 생성자 또는 메소드를 실행할 때마다 실행 결과를 저장하고 있는 정보 테이블을 참조해서 노드 한 개를 만들고 트리를 구성한다. 트리는 프로그램에서 생성자를 포함한 메소드의 호출에 대한 실행 순서에 따라 하향식으로 생성된다. 생성된 실행 트리는 Java 프로그램에서 main 메소드가 실행 트리의 루트 노드가 된다. 그리고 실행 순서에 따라 왼쪽부터 오른쪽으로 자식 노드와 형제 노드가 구성된다. 이와 같이 프로그램의 실행 트리는 프로그램의 실행 결과에 대한 실행 순서의 정보를 저장하고 있다.

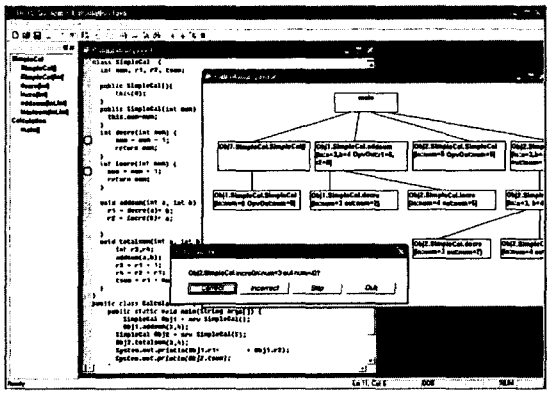
4.3 HDTS 디버거

HDTS 디버거는 알고리즘식 디버깅 추적기와 단계

적 디버깅 추적기, 그리고 프로그램 분할기로 구성된다. 이러한 HDTS 디버거는 [알고리즘 1]의 디버깅 순서에 따라 동작한다. 이 디버깅의 목적은 가능한 한 프로그래머와 디버깅 시스템과의 상호작용을 최소화하여 프로그래머가 논리적인 오류를 쉽게 발견하기 위한 것이다.

4.4 HDTS 탐색기

프로그래머가 편리하게 프로그램을 디버깅할 수 있도록 하기 위해 설계한 그래픽 사용자 인터페이스의 전체 화면은 [그림 5]와 같다. 윈도우의 전체 화면은 클래스와 메소드 정보 브라우저, 변수 정보 브라우저, 원시 프로그램을 편집하고 단계적 프로그램 디버깅을 할 수 있는 편집기, 그리고 실행 트리 브라우저, 실행 트리 탐색 메시지 윈도우로 구성된다.



[그림 5] HDTS 시스템의 그래픽 사용자 인터페이스

클래스와 메소드 정보 브라우저는 원시 프로그램의 클래스와 클래스 내에 포함된 메소드의 정보를 표시한다. 원시 프로그램을 편집하고 단계적 프로그램 디버깅을 할 수 있는 편집기는 왼쪽에 break-point를 설정할 수 있는 공간이 있으며 프로그램을 편집할 때 키워드가 파랑 색으로 표시된다. 단계적 프로그램 디버깅을 할 때는 break-point를 설정하는 부분에 화살표로 현재 실행 중인 문장을 나타내고 그 문장의 라인은 블록으로 강조된다. 이때 변수 윈도우와 함께 사용된다.

실행 트리 탐색 메시지 윈도우는 실행 트리 윈도우에서 프로그래머가 하나의 노드를 선택함으로써 시작된다. 디버깅 시스템의 메시지에 따라 프로그래머는 correct, incorrect, skip, quit를 선택할 수 있다. 현재 노드의 정보가 옳다고 생각이 들면 correct를, 아니면 incorrect를 선택한다. 결과를 예상할 수 없으면 skip을 선택하고 quit를 선택하면 디버깅을 종료한다. 디버깅 시스템은 실행 트리 탐색에 대한 프로그래머의 응답을 기반으로 오류가 포함된 메소드를 발견하고, 원시 프로그램을 편집기로 보여주고 단계적 프로그램 디버깅을 할 수 있도록 한다.

5. 결론

본 논문은 자바 프로그램에 존재하는 논리적인 오류를 쉽고 빠르게 발견하고 수정하기 위해 이전에 제안한 HDTS 기술을 구현하기 위한 시스템을 설계하였다. 우선, HDTS 시스템을 설계하기 위해 Java 의 부분 언어를 정의하였다. Java 의 추상 구문은 동적 결합, 상속, 그리고 인터페이스와 같은 객체 지향의 특징을 포함시켰고, switch 문과 for 문은 의미상으로 if 문과 while 문과 같기 때문에 추상구문에서 제외시켰다.

본 논문에서 설계한 전반적인 HDTS 시스템의 구조는 프로그램 변환기, 실행 트리 생성기, HDTS 디버거, 그리고 그래픽 사용자 인터페이스로 구성하여 구현하였다.

참고문헌

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

[2] H. J. Kouh, and W. H. Yoo, "The Efficient Debugging System for Locating Logical Errors in Java Programs", *Proceedings of International Conference on Computational Science and Its Application - ICCSA 2003*, Vol. 2667 of LNCS, pp. 684-693, Springer-Verlag, Montreal, Canada, May 18-21 2003.

[3] H. J. Kouh, K. T. Kim, S. M. Jo, W. H. Yoo, "Debugging of Java Program using HDT with Program Slicing," International Conference on Computational Scicence and Its Application-ICCSA 2004, Assisi, Italy, Springer-Verlag, LNCS 3046, pp. 524-533, 2004. 5.

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

[5] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy, "Generalized Algorithmic Debugging and Testing," *Proceeding of the 1991 ACM SIGPLAN Conference*, pp. 317-326, Toronto, Canada, June 1991., *ACM LOPLAS -- Letters of Programming Languages and Systems. Vol. 1, No. 4*, December 1992.

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

[7] L. Larsen, and M. J. Harrold, " Slicing Object-Oriented Software," Proceedings of the 18th ICSE Conference, pp. 495-505, 1996.

[8] F. Tip, " A survey of program slicing techniques," *Journal of Programming Languages, Vol. 3(3), pp.121-189,* Chapman and Hall, London, September 1995.

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

[10] 고훈준, 양창모, 유원희, "Java 프로그램을 위한 효율적인 디버깅 방법," 2002 춘계학술대회 논문집 Vol. 7, No. 1, pp. 170-176, 동양대학교, 한국산업정보학회, 2002. 6.

[11] 고훈준, 유원희 "자바 언어를 위한 알고리즈믹디버깅 기술의 설계," 정보처리학회 논문지 A 제 11-A 권 제 1 호, pp. 97-108, 한국정보처리학회, 2004. 2.