자바 프로그램을 위한 복합 디버깅 시스템의 설계
Design of Hybrid Debugging System for Java Programs

고훈준
경인여자대학 정보미디어학부

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

요약

기존 연구에서는 자바 프로그램에서 논리적인 오류를 찾기 위한 기술로 HDTS를 제안했었다. HDTS는 알고리즈믹 프로그램 디버깅 기술을 이용하여 실행트리에서 오류를 포함하고 있는 메소드를 찾고, 단계적 프로그램 디버깅 기술을 이용하여 오류를 포함하고 있는 메소드에서 오류가 있는 문장을 찾아낸다. 그리고 분할 기술을 이용하여 오류를 포함하고 있는 메소드 내의 문장 중에서 디버깅에 관련이 없는 문장과 실행트리에서 불필요한 노드를 제거하여 노드의 수를 줄임으로서 사용자가 디버깅하는 횟수를 최소화할 수 있는 프로그램 디버깅 기술이다. 본 논문은 자바 프로그램을 디버깅할 수 있는 HDTS 시스템을 구현하기 위한 설계를 한다. 먼저, 자바의 부분언어를 정의하고 자바 원시 코드를 번역할 수 있는 번역기와 실행할 수 있는 가상머신을 설계한다. 그리고 사용자가 디버깅하기 위한 사용자 그래픽 인터페이스를 설계한다.

■ 중심어 : | HDTS | 자바 | 프로그램 변환기 | 프로그램 실행기 |

Abstract

In the previous work, we presented HDTS for locating logical errors in Java programs. The HDTS 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. The technique can remove the unnecessary statements and nodes in debugging using a program slicing technique at the execution tree. So HDTS reduces the number of program debugging. In this paper, we design HDTS system for debugging java programs. We define small subset of Java language and design the translator that translates java source codes and the virtual machine that runs java programs. We design GUI(Graphical User Interface) for debugging.

■ keyword : | Hybrid Debugging Technique with Slicing | Java | Program Translator | Program Executor |

I. 서론

프로그램의 오류는 테스팅(testing)과 디버깅(debugging)을 하거나 검증(verification)을 통해 발견하고 수정할 수 있다. 다양한 검증 기술들은 사용자의 참여를 최소화하면서 자동으로 프로그램의 오류를 발견할 수 있도록 연구되고 있다. 그러나 이러한 검증 기술들은 프로그램 개발 시에 발생할 수 있는 실행 시간 오류는 쉽게 발견할 수 있지만 논리적인 오류를 발견하는 것은 어렵다. 따라서 최근까지 프로그래머는

접수번호 : #080923-004
접수일자 : 2008년 09월 23일
심사완료일 : 2008년 10월 08일
교신저자 : 고훈준, e-mail : hjkouh@kic.ac.kr

논리적인 오류를 발견하기 위해서 테스팅과 디버깅을 많이 사용하고 있다.

[1]에서는 자바 원시 코드에서 논리적인 오류를 발견하고 수정할 수 있는 복합 디버깅 기술 HDTS(Hybrid Debugging Technique with Slicing)를 제안하였다. HDTS는 알고리즈믹 프로그램 디버깅 기술[8][9]과 단계적 프로그램 디버깅 기술[2][7]을 혼합하고 프로그램 분할[4][11]의 개념을 확장하여 적용한 기술이다. HDTS는 알고리즈믹 프로그램 디버깅 기술을 이용하여 실행트리에서 오류를 포함하고 있는 메소드를 찾고, 단계적 프로그램 디버깅 기술을 이용하여 논리적인 오류를 포함하고 있는 메소드에서 그 오류가 있는 문장을 찾아낸다. 그리고 HDTS는 프로그램을 디버깅할 때 오류를 포함하고 있는 메소드 내의 문장 중에서 디버깅에 관련이 없는 문장과 실행트리에서 오류가 없는 노드를 프로그램 분할 기술을 이용하여 제거함으로써 노드의 수를 줄여 사용자가 디버깅하는 횟수를 최소화할 수 있는 프로그램 디버깅 기술이다. 이 기술은 디버깅하는 횟수를 줄여 디버깅하는 시간을 최소화하고, 그 결과 프로그램 개발 시간을 단축할 수 있다[1].

본 논문에서는 [1]에서 제안한 HDTS 기술을 기반으로 자바 프로그램을 디버깅할 수 있는 HDTS 시스템을 설계한다.

논문의 구성은 다음과 같다. II장에서는 디버깅 시스템 구현을 위해 필요한 자바 언어를 정의하고, III장에서는 제안하는 HDTS 시스템의 기능적 구조를 설명한다. IV장에서는 HDTS 탐색기의 구조를 설명하고 디버거 알고리즘을 설명한다. V장에서는 자바 프로그램을 번역하기 위한 프로그램 변환기 설계하고, VI장에서는 프로그램을 실행하고 실행트리를 생성하는 실행트리 생성기를 설계한다.

II. 프로그래밍 언어의 정의

본 논문에서는 복합 디버깅 기술 HDTS를 지원하는 디버깅 시스템을 구현하기 위해 자바 언어를 사용하였다. [그림 1]에서 정의한 추상구문은 자바의 부분 언어로 [6]의 언어 명세를 기반으로 정의하였다.

```
==========================================
Program ::= Classes
Classes ::= Class Classes | ϵ
Class ::= ClassDecl | InterfaceDecl
ClassDecl ::= [public] class Id [extends Name]
[implementsName]{ 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 ]) | newTypeSpecifier [ Expr ] | true
| false | null | super | this | Id | Num | Char
==========================================
```

그림 1. Java 부분 언어의 추상구문

[그림 1]의 추상구문은 자바의 완전한 언어를 포함

하고 있지는 않지만, 이전 논문에서 제안된 HDTS를 테스트하기 위한 기본형으로는 적당하다. [그림 1]에서 제시하는 자바의 추상 구문은 동적 결합, 상속, 그리고 인터페이스와 같은 객체지향의 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.

III. HDTS 시스템의 설계

본 장에서는 HDTS를 이용하여 자바 프로그램 내에 포함된 논리적인 오류를 발견하는 HDTS 시스템을 설계한다.

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

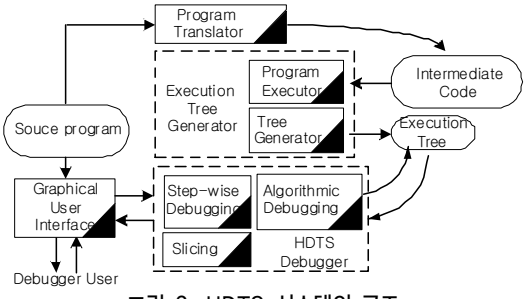

그림 2. HDTS 시스템의 구조

프로그램 변환기는 원시 프로그램을 번역하여 중간 코드를 생성한다. 실행 트리 생성기는 프로그램 실행기(program executor)와 트리 생성기(tree generator)로 구성된다. 이때 프로그램 실행기는 중간 코드를 입력 받아 가상기계에서 실행하고, 트리 생성기는 실행 결과로부터 실행 트리를 생성한다. HDTS 디버거는 HDTS 탐색기를 이용하여 생성된 실행 트리로부터 디버깅을 한다.

IV. HDTS 탐색기와 디버거

HDTS 탐색기는 프로그래머가 편리하게 프로그램을 디버깅할 수 있도록 하기 위해 설계한 그래픽 사용자 인터페이스로 전체 화면은 [그림 3]과 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.

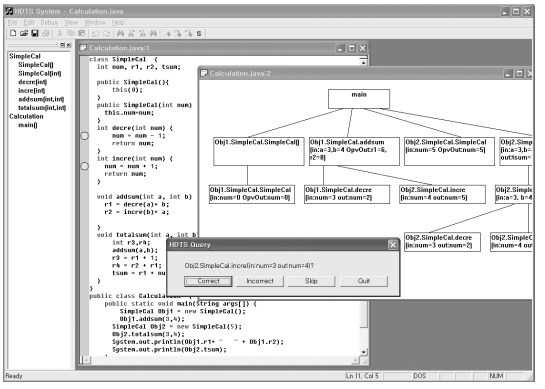

그림 3. HDTS 시스템의 그래픽 사용자 인터페이스

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 파란색으로 표시된다. 단계적 프로그램 디버깅을 할 때는 break-point를 설정하는 부분에 화살표로 현

재 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.

HDTS 디버거는 알고리즈믹 디버깅 추적기와 단계적 디버깅 추적기, 그리고 프로그램 분할기로 구성된다. HDTS 디버거는 실행 트리를 [알고리즘 1]의 디버깅 순서에 따라 동작한다.

알고리즘 1. HDTS의 알고리즘

```
==========================================
Input: Incorrect Program
1 procedure HDTS(P)
2 begin
//let t={(o1,c1,m1,in1,out1),…,(on,cn,mn,inn,outn)}
be the top level trace nodes of (o0, c0, m0, in0, out0)
3 repeat
4   Build an execution tree from t
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)<> t 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 (o, c, n, in, out)는 실행트리를 구성하기 위한 하나의 노드로 정의된다. 이때 o는 객체 이름, c는 클래스 이름, m은 메소드 이름이다. in은 입력 변수들의 집합이고, out은 출력 변수들의 집합이다. 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 재생성하여 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 함수내의 문장들을 실행하고 제어하면서 오류가 있는 문장을 발견한다[1].

이 디버깅의 목적은 가능한 한 프로그래머와 디버깅 시스템과의 상호작용을 최소화하여 프로그래머가 논리적인 오류를 빠르고 쉽게 발견하기 위한 것이다.

V. 프로그램 변환기

프로그램 변환기는 [그림 4]의 구조도에서 컴파일러의 전위부(front-end)에 해당하는 부분이며, 어휘 분석기(lexical analyzer), 파서(parser), 중간코드 생성

기(intermediate code generator)로 구성된다. 어휘 분석과 구문 분석은 WINDOWS용 Lex & Yacc[10]의 자동화 도구를 사용하여 개발하였다.

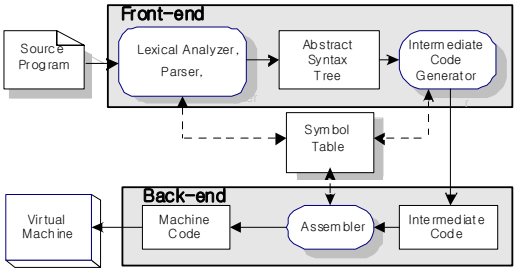

그림 4. 프로그램 변환기와 프로그램 실행기의 구조도

프로그램 변환기는 원시 프로그램을 입력 받아 어휘 분석기와 파서를 이용하여 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)[5]을 생성한다. 심볼 테이블의 중요 부분은 다음과 같다.

```
struct c_symbol {
        char *c_name;
        struct symbol *sym;
        struct c_symbol *next;
        ...
}
```

c_symbol은 클래스를 연결하는 테이블로 c_name은 클래스의 이름을 의미하고, 여러 클래스를 연결하고 각 클래스 내에는 멤버 변수, 메소드, 생성자 위한 심볼 테이블을 가지고 있다.

```
struct symbol {
        char *name;
        IDENT IDkind;
        union sym_value value;
        Exptype v_type;
        struct function_node *FNode;
        struct sym_struct *next;
        ...
}
```

IDENT은 Label, Variable, Array, Pointer, Function을 나타내고 Exptype은 Void, Char, Int, Double을 나타낸다. FNode는 심볼이 함수(메소드 또는 생성자) 이름이면 해당하는 함수 노드의 위치를 저장한다.

AST를 생성하기 위해서는 심볼 테이블과 함수 노드 테이블 그리고 트리 노드 테이블을 사용한다. 함수 노드 테이블의 중요 부분은 다음과 같다.

```
struct function_node {
        SymStruct *name;
        ExpType ReturnType;
        struct symbol *LocalSym;
        struct tree_node *treenode;
        ...
}
```

ReturnType은 함수의 리턴 타입을 나타내고 LocalSym은 각 함수 내에 지역 심볼 테이블을 나타낸다. 그리고 treenode는 각 문장들을 트리로 구성하기 위한 노드를 나타낸다.

트리 노드 테이블의 중요 부분은 다음과 같다.

```
struct tree_node {
        struct symbol SymPtr;
        struct treenode *sibling;
        struct treenode *child[4];
        ...
}
```

트리 노드는 원시 프로그램의 각 문장들을 트리로 구성하기 위한 노드이다. SymPtr은 각 노드의 변수 정보가 저장된 심볼 테이블의 위치를 나타낸다. 그리고 프로그램 변환기는 AST로부터 중간 코드를 생성한다. 중간코드는 [표 1]와 같이 현재는 59개를 정의하였다.

IADD, DADD에서 I는 정수형을 뜻하고, D는 실수형을 뜻한다. PUSHL은 현재의 FramePtr와 offset을 더해서 그 값을 스택에 PUSH하는 명령어이다. SETSP는 스택에 [length]만큼 공간을 확보하고 스택 포인터를 바꾸고, SETP는 현재의 스택 포인터를 지역

변수의 시작주소로 하는 명령어이다. CONTENTS는 스택에서 POP한 값을 주소로 하여 그 주소의 값을 스택에 저장하는 명령어이다. 또한 I2D는 스택에 있는 값을 POP해서 그 값을 실수 값으로 타입 변환 후 스택에 저장하는 명령어이고 D2I는 반대로 스택에 있는 값을 POP해서 그 값을 정수 값으로 타입 변환 후 스택에 저장하는 명령어이다. 번호는 목적 코드의 명령어 번호와 일치한다.

VI. 실행 트리 생성기

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

6.1 프로그램 실행기

프로그램 실행기는 [그림 4]의 구조도에서 컴파일러의 후위부(back-end)와 가상 기계에 해당된다. 프로그램 실행기는 [표 1]의 중간 코드 명령어로부터 구성된 중간 코드로부터 어셈블러에 의해 목적 코드를 생성한다. 목적 코드는 [표 1]의 번호로 구성하여 나타낸다. 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 Jonathan Amsterdam[3]이 제시한 가상 기계를 기반으로 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 0번지부터 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

표 1. 중간 코드 명령어

번호	명령어	번호	명령어	번호	명령어	번호	명령어
1	IADD	16	ILSSEQL	31	IPUSH variable	46	SETSP length
2	ISUB	17	INOT	32	IPOPC variable	47	SETP
3	IMUL	18	DEQUAL	33	IPOP	48	RDCHAR
4	IDIV	19	DNOTEQL	34	DPUSHC double value	49	RDINT
5	INEG	20	DGREATER	35	DPUSH variable	50	RDDOUBLE
6	DADD	21	DLESS	36	DPOPC variable	51	WRCHAR
7	DSUB	22	DGTREQL	37	DPOP	52	WRINT
8	DMUL	23	DLSSEQL	38	AREF size	53	WRDOUBLE
9	DDIV	24	DNOT	39	BRANCH address	54	CONTENTS
10	DNEG	25	PUSHC char value	40	JUMP	55	ICONTENTS
11	IEQUAL	26	PUSH variable	41	BREQL address	56	DCONTENTS
12	INOTEQL	27	PUSHL offset	42	BRLSS address	57	HALT
13	IGREATER	28	POPC variable	43	BRGTR address	58	I2D
14	ILESS	29	CPOP	44	CALL address	59	D2I
15	IGTREQL	30	IPUSHC integer value	45	RETURN		

32bit, 실수는 64bit로 구성된다.

이러한 다섯 가지 기본 전제 조건으로부터 설계된 스택 기반 가상 기계의 실행 구조는 [그림 5]와 같다. 가상 기계의 로더(loader)는 목적 코드를 메모리로 로드하고 스택을 사용하여 한 번에 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.

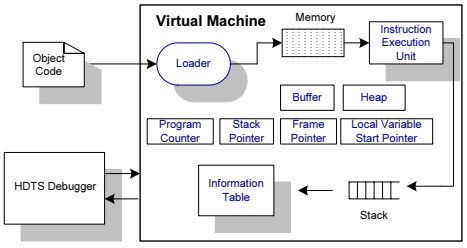

그림 5. 가상 기계의 실행 구조

6.2 트리 생성기

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 실행 트리를 구성한다. 실행트리의 개념은 [1][9]에 정의되어 있다. 다음은 하나의 노드를 구성하는 중요 부분은 다음과 같다.

```
struct ex_tree {
        char *o_name;
        char *c_name;
        char *f_name;
        struct in_node *InNode;
        struct out_node *OutNode;
        struct ex_tree *parent
       struct ex_tree *child[100];
       ...
}
```

o_name은 객체이름, c_name은 클래스이름, f_name은 메소드 또는 함수 이름을 의미하고, InNode는 매개변수의 이름과 값을 저장하는 구조체이고, OutNode는 반환되는 변수와 값을 저장하는 구조체이다.

실행 트리는 프로그램에서 생성자를 포함한 메소드의 호출에 대한 실행 순서에 따라 노드들이 하향식으로 구성된다. 생성된 실행 트리는 자바 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.

VII. 결론

디버거 사용자가 자바 프로그램을 개발할 때 빠르게 논리적인 오류를 찾고 수정하기 위해서 이전 논문에서는 HDTS 기술에 대한 알고리즘을 제안 하였다. HDTS는 알고리즈믹 프로그래밍 기술, 단계적 프로그램 기술, 프로그램 분할 기술을 혼합하여 사용자가 디버깅하는 횟수를 줄일 수 있는 기술이었다.

본 논문에서는 HDTS 기술을 구현하기 위해 HDTS 시스템을 설계하였다. 첫째, 제안된 HDTS를 테스트하기 위한 기본형 언어로 자바의 동적 결합, 상속, 그리고 인터페이스와 같은 객체지향의 특징을 포함하고 있는 자바의 부분언어를 정의하였다. 둘째, 설계한 프로그래밍 언어를 디버깅하기 위해 탐색기와 번역기 그리고 가상기계를 설계하였다. 탐색기는 프로그램을 작성하고 디버깅할 수 있는 편집기와 실행트리 브라우저, 실행트리 탐색 메시지 윈도우로 구성하였다. 번역기는 프로그램 변환기, 실행 트리 생성기, HDTS 디버거로 구성하여 설계하였다. 프로그램 변환기는 WINDOW용 Lex & Yacc을 이용하여 어휘 분석과 파서, 중간코드 생성기로 설계하였다. 그리고 59개의 중간코드를 정의하여 중간코드를 생성하고 이를 실행하는 프로그램 실행기와 실행 트리를 생성할 수 있는 트리 생성기를 포함하는 실행트리 생성기를 설계하였다.

지금가지 이루어진 연구를 바탕으로 앞으로 수행해야할 일은 완전한 자바 언어에 대해 처리할 수 있는 디버거 시스템을 개발해야 하고, HDTS 탐색기를 향상시키는 것이다. 그래픽 사용자 인터페이스는 사용자가 디버깅하는 효율에 크게 영향을 끼친다. 특히 실행 트리를 효율적으로 표현할 수 있는 방법이 요구된다.

참 고 문 헌

[1] 고훈준, "자바 원시 코드에서 논리적인 오류를 찾는 복합 디버깅 기술의 설계", 한국콘텐츠학회논문지, 제6권, 제10호, pp.114-125, 2006.

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

[3] J. Amsterdam, *A SIMPL Compiler*, BYTE, Vol.110, No.11, 1985.

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

[5] C. W. Fraser and D. R. Hanson, "A Retargetable Compiler for ANSI C," ACM SIGPLAN Notices Vol.26, No.10, pp.29-43, 1991.

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

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

[8] G. Kokai, L. Harmath, and T. Gyim'othy, "Algorithmic Debugging and Testing of Prolog Programs," P. of the 4th International Conference on Logic Programming, 8th Workshop on Logic Programming Environments Leuven-ICLP '97, pp.14-21, 1997.

[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] J. R. Levine, T. Mason, and D. Brown, *LEX & YACC*, O'Reilly & Associates, Inc., 1995.

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

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 정보미디어학부 교수

<관심분야> : 디버거, 프로그램보안, 웹서비스, 생물정보학