Journal of Korea Multimedia Society Vol. 7, No. 10, October 2004(pp. 1485-1495)

개선된 동적 객체지향 프로그램 슬라이싱에 관한 연구

박순형[†], 박만곤[††]

요 약

본 논문에서는 효율적인 동적 객체지향 프로그램 슬라이싱을 구현하기 위한 개선된 동적 객체지향 프로그램 종속 그래프 기법을 제안하였고, 이 그래프를 이용한 동적 객체지향 프로그램 슬라이싱 구현 단계를 제안하였다. 이 구현 단계는 프로그램 노드 분석 단계, 프로그램 실행이력 분석 단계, 동적 객체지향 프로그램 종속 그래프 작성 단계 그리고, 프로그램 슬라이스 작성 단계 등 모두 4 단계로 구성되어 있다. 본 논문에서 제시한 기법이 정확함을 보이기 위해 본 논문에서 제시한 알고리즘을 실제 구현하였다. 구현 프로그램은 FORTRAN과 VISUAL C++를 사용하였다. 그리고, 본 논문에서 제안한 동적 객체지향 프로그램 슬라이싱 기법이 기존의 객체지향 프로그램 슬라이싱 기법과 기존의 동적 객체지향 프로그램 슬라이싱 기법에 비해 효율적임을 보이기 위해 그래프의 복잡도를 측정하여 비교하였다. 그리고, 프로그램 슬라이스의 크기도 함께 측정하여 본 논문에서 제시한 기법이 효율적임을 증명하였다.

A Study on the Improved Dynamic Object-Oriented Program Slicing

Soon-Hyung Park[†], Man-Gon Park[††]

ABSTRACT

We propose the representation of a improved dynamic object-oriented program dependence graph so as to process the slicing of object-oriented programs that is composed of related programs in order to process certain jobs. We also propose an efficient slicing algorithm using the relations of relative tables in order to compute dynamic slices of object-oriented programs. We programmed the algorithm by using Fortran and Visual C++. The procedure that computes the dynamic object-oriented program slices using the improved dynamic object-oriented program dependence graph(IDOPDG) is divided into four steps. Consequently, the efficiency of the proposed improved dynamic object-oriented program dependence graph(IDOPDG) technique is also compared with the dependence graph techniques discussed previously. As a result, this certifies that an improved dynamic object-oriented program dependence graph is more efficient in comparison with the traditional dynamic object-oriented program dependence graph(DOPDG).

Key words: Dynamic Program Slicing(동적 프로그램 슬라이싱), Object-Oriented Program Slicing (객체지향 프로그램 슬라이싱), Program Dependence Graph(프로그램 종속 그래프)

※ 교신저자(Corresponding Author) : 박순형, 주소 : 부산 시 남구 대연3동 599-1, 전화 : 051) 620-6391, FAX : 051) 628-6155, E-mail : nepaipark@yahoo.com
접수일 : 2003년 12월 26일, 완료일 : 2004년 4월 26일
[†] 정회원, Christchurch College of Education
[††] 종신회원, 부경대학교 컴퓨터 멀티미디어 공학부 교수 (E-mail : mpark@pknu.ac.kr)

1. 서 론

프로그램 슬라이싱(program slicing)은 프로그램 P에서 어떤 포인터 p에 있는 변수 var의 값에 직 간접적으로 영향을 끼치는 모든 명령문들을 찾는 과정으로서 주어진 어떤 프로그램 중에서 관심이 있는

변수에 대해 직접적 또는 간접적으로 관련된 명령문만을 모아놓은 또 다른 프로그램을 제공해 줌으로서 디버깅 등에 대한 효율성을 높여주는 기술이다[4, 7,11,13].

프로그램 슬라이싱 기법은 프로그램 입력 값에 따른 실행이력(execution history)의 적용여부에 따라 정적 프로그램 슬라이싱(static program slicing) 기법과 동적 프로그램 슬라이싱(dynamic program slicing) 기법으로 나눈다[1,2,8]. 정적 프로그램 슬라이싱 기법은 슬라이싱 기준(criterion)에 주어진 프로그램의 한 지점에서 변수에 영향을 줄 수 있는 모든 노드들을 추출하는 방법이며 동적 프로그램 슬라이싱 기법은 프로그램의 입력에 대해 실행 이력상의 어떤 실행위치 q에서 관심 변수에 관련된 원래의 프로그램과 그것의 결과가 일치하는 프로그램 실행의 부분을 산출하는 것으로, 동적 프로그램 슬라이스(dynamic program slices)는 프로그램의 입력 자료의 값에 의해 발생되는 프로그램의 실행 이력을 추적한 다음 실행 이력 상에서 기준 변수에 실제적으로 영향을 주는 모든 명령문들로 구성된다[3,6,9,14].

객체지향 프로그램 슬라이싱(object-oriented program slicing)은 객체지향 프로그램 종속 그래프에서 객체지향 프로그램의 중심이 되는 클래스와 객체의 흐름을 추적하여 객체지향 프로그램 속성들에 대한 슬라이스를 구하는 작업으로서 정적 프로그램 슬라이싱 기법 혹은 동적 프로그램 슬라이싱 기법에 의해 슬라이스를 산출할 수 있다[12,15].

프로그램 슬라이싱 결과는 프로그램 종속 그래프(program dependence graph) 기법 혹은 시스템 종속 그래프(system dependence graph) 기법 등을 사용하여 산출할 수 있다[5].

기존의 종속 그래프 특히, 시스템 종속 그래프와 객체지향 종속 그래프들은 프로시저간의 자료전달을 표시하기 위하여 많은 정점들과 간선들이 필요하기 때문에 매우 복잡하므로 실제로 프로그래머나 테스터들이 적용하기에는 매우 어렵다[10].

본 논문에서는 기존의 프로그램 종속 그래프 기법이나 시스템 종속 그래프 기법 보다 프로그램 슬라이싱을 더 효율적으로 표현할 수 있는 동적 시스템 종속 그래프 (dynamic system dependence graph) 기법을 제안하였고, 이 기법을 바탕으로 동적 객체지향 프로그램 슬라이싱을 구현하기 위한 동적 객체지향 프로그램 종속 그래프 기법을 제안하였고, 이 그래프를 이용한 동적 객체지향 프로그램 슬라이싱 구현 단계를 제안하였다.

본 논문에서는 다음과 같은 방법으로 구성된다. 먼저 2 장에서는 기존의 프로그램 슬라이싱 기법에 대한 관련 연구들에 대해 소개하였으며 3 장에는 본 논문에서 제시한 동적 객체지향 프로그램 슬라이싱 기법을 처리하기 위해 개선된 동적 객체지향 프로그램 종속 그래프를 이용한 동적 객체지향 프로그램 슬라이스들을 구하는 구현 절차에 대해 기술하였다. 4장에서는 본 논문에서 제안한 동적 객체지향 지향 프로그램 슬라이싱 기법과 기존의 동적 객체지향 프로그램 슬라이싱 기법에 대한 특징과 장점, 그리고 단점에 대하여 기술하였다. 5장에서는 본 논문에서 제시한 슬라이싱 기법을 객체지향 프로그램 슬라이싱 논문들이 일반적으로 많이 적용하고 있는 엘리베이터 예제 프로그램에 적용한 적용사례를 기술하였고, 6장에서는 본 논문에서 제시한 개선된 기법과 기존의 동적 객체지향 프로그램 슬라이싱 기법에 대한 산출 그래프의 복잡도를 비교함으로써 두 기법에 대한 효율성을 고찰하였다.

2. 기존의 동적 객체지향 프로그램 슬라이싱

기존의 동적 객체지향 프로그램 슬라이싱 기법은 슬라이스 결과를 산출하기 위해 Agrawal이 제안한 동적 프로그램 종속 그래프 기법을 이용한 동적 객체지향 프로그램 종속 그래프(DOPDG : Dynamic Object-oriented Program Dependence Graph)를 사용한다.

객체지향 프로그램의 동적 객체지향 프로그램 종속 그래프의 구성은 해당 원시 프로그램의 제어 흐름과 자료 흐름에 대한 동적 분석에 기초하고 있다. 그리고, 이것은 procedural 프로그램에 대한 동적 종속 그래프의 구축과 유사하다. 하나의 procedural 프로그램에서의 call 명령문은 function 기능을 가진 명령문이나 procedure를 호출하는 명령문이다. 그러나, 객체지향 프로그램에서는 classes, instances, objects 그리고, 동적 바인딩을 고려해야한다. 그림 1은 프로그램 1에 대한 동적 객체지향 프로그램 종속 그래프이다.

```
CE1 :   class Elevator {
        public:
E2:     Elevator(int 1_top_floor)
S3:      { current_floor = 1;
S4:        current_direction = UP;
S5:        top_floor = 1_top_floor; }
E6:     virtual ~Elevator() {}
E7:     void up()
S8:      { current_direction = UP; }
E9:     void down()
S10:     { current_direction = DOWN; }
E11:    int which_floor()
S12:     { return current_floor; }
E13:    Direction direction()
S14:     { return current_direction; }

E15:    virtual void go (int floor)
S16:     { if (current_direction == UP)
S17:        { while ((current_floor != floor) &&
                    (current_floor <= top_floor))
C18:          add(current_floor, 1); }
           else
S19:        { while ((current_floor != floor) &&
                    (current_floor > 0))
C20:              add(current_floor, -1); }
         }

        private:
E21:     add(int &a, const int& b)
S22:      { a = a + b; };
        protected:
         int current_floor;
         Direction current_direction;
         int top_floor;
        };

CE23:   class AlarmElevator : public Elevator {
        public:
E24:     AlarmElevator(int top_floor):
S25:       Elevator(top_floor)
S26:        { alarm_on = 0; }
E27:     void set_alarm()
S28:        { alarm_on = 1; }
E29:     void reset_alarm()
S30:        { alarm_on = 0; }
E31:     void go(int floor)
S32:      { if (!alarm_on)
C33:          Elevator::go(floor)
        } ;

        protected:
          int alarm_on;
        } ;
E34:     main(int argc, char **argv) {
          Elevator *e_ptr;
S35:      if (argv[1])
S36:          e_ptr = new AlarmElevator(10);
          else
S37:          e_ptr = new Elevator(10);
C38:      e_ptr->go(3);
S39:      cout << "\n Currently on floor:" <<
           e_ptr->which_floor() << "\n";
        }
```

프로그램 1. 예제 프로그램

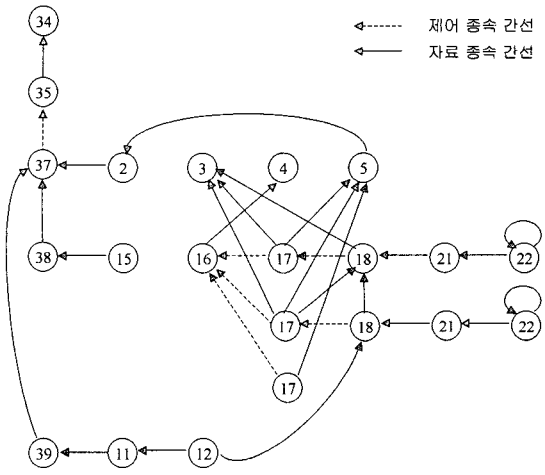


그림 1. 프로그램 1에서 C = (s39, which_floor)에 대한 DOPDG

객체지향 프로그램에 대한 슬라이싱 기준은(s, v, t, i)이다. s는 프로그램의 명령문이고, v는 s에서 사용된 변수, t는 입력 i에 대한 실행 궤도이다.

P를 객체지향 프로그램이라 하고, G = (V, A)를 P에 대한 DOPDG라 하자. 이때 V는 정점들의 집합이며, A는 간선들의 집합이다. 주어진 동적 슬라이스의 기준 (s, v, t, i)에 대한 G의 동적 슬라이스 DS_G는 G의 정점들의 subset이다. 이때, G에서 v'에서부터 v까지의 path가 존재한다면 $v' \in V$, $v' \in DS_G(s, v, t, i)$에 대해 $DS_G(s, v, t, i) \subset V$이다.

주어진 실행 궤도에 대해 DOPDG를 구성하는 즉시 기준 노드에 대해 DOPDG를 운행하기 위해 depth-first 혹은 breadth-first 그래프 운행 알고리즘을 사용함으로써 동적 슬라이스를 산출할 수 있다. 그리고, 객체지향 프로그램의 DOPDG에 대한 동적 슬라이스는 DOPDG의 정점들의 subset이다.

3. 개선된 동적 객체지향 프로그램 슬라이싱

본 논문에서 제시한 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG : Improved Dynamic Object-oriented Program Dependence Graph)를 이용한 동적 객체지향 프로그램 슬라이스를 구하는 절차는 크

게 4 단계로 나눌 수 있다.

첫째, 프로그램 노드 분석 단계
둘째, 프로그램 실행이력 분석 단계
셋째, 동적 객체지향 프로그램 종속 그래프 작성 단계
넷째, 프로그램 슬라이스 작성 단계

IDOPDG는 기존의 객체지향 프로그램 종속 그래프 개념을 기반으로 하고 동적 시스템 종속 그래프 기법을 사용하여 객체지향 프로그램을 표현할 수 있도록 하는 표현법이다. 동적 시스템 종속 그래프 기법은 동적 슬라이싱 개념을 기초로 하여 시스템 종속 개념을 표현할 수 있도록 하는 종속 그래프 기법이다[5].

3.1 프로그램 노드 분석 단계

프로그램 노드 분석 단계는 원시 프로그램에 대한 노드관련 테이블(table related nodes)을 작성하는 단계이다. 노드관련 테이블은 각 명령문에 해당되는 노드들의 구성 요소를 저장해 놓은 자료의 집합으로서 노드 번호, 노드의 종류, DEF, REF, 소속모듈번호 그리고, 상위제어노드번호로 구성된다.

(1) 노드의 종류

프로그램을 구성하는 노드들은 11가지로 구분되며 표 1에 나타나 있다.

(2) DEF

해당 노드에서 바뀌어진 값을 가진 변수의 집합

(3) REF

해당 노드에서 사용된 값을 가진 변수의 집합

(4) 소속모듈번호

상위 노드의 번호

(5) 상위제어노드번호

상위 반복 제어 노드의 번호

프로그램을 구성하는 노드 테이블이 표 1에 나타나 있고, DEF와 REF를 구성하는 식이 프로그램 2에 나타나 있다.

표 1. 프로그램을 구성하는 노드 테이블

노드의 종류	약어	
class	CE	
method	E	M
procedure		P
call	S	C
return		R
assign		A
input		I
write		W
repeat		L
select		D
constructor		N

```
DEF(class(parameter)) = {class}
REF(class(parameter)) = {parameter}

DEF(method(parameter)) = {method}
REF(method(parameter)) = {parameter}

DEF(call(parameter)) = {call}
REF(call(parameter)) = {parameter}

DEF(procedure(parameter)) = {procedure}
REF(procedure(parameter)) = {parameter}

DEF(var = expression) = {var}
REF(var = expression) = {expression}

DEF(read(var)) = {var}
REF(read(var)) = ∅

DEF(write(var)) = ∅
REF(write(var)) = {var}

DEF(predicate) = ∅
REF(predicate) = {predicate}
```

프로그램 2. DEF와 REF를 구성하는 식

3.2 프로그램 실행이력 분석 단계

소스 프로그램(source program)이 실제 실행되었을 때의 실행이력을 분석하여 실행이력 테이블(execution history table)을 작성하는 단계이다.

실행이력 테이블이란 실제 소프트웨어 프로그램이 실행되어졌을 때의 실행되는 이력에 대한 자료의 집합으로서 노드실행순번과 노드번호로 구성된다. 노드실행순번은 실행이력순번을 나타내고, 노드번호는 노드관련테이블에서의 노드번호와 같다.

3.3 동적 객체지향 프로그램 종속 그래프 작성 단계

원시 프로그램에 대한 정적 정보와 주어진 입력 자료에 의한 실행이력을 기반으로 동적 객체지향 프로그램 종속 알고리즘을 적용하여 개선된 동적 객체

지향 프로그램 종속 그래프(IDOPDG)를 작성하는 단계이다.

IDOPDG는 다음 과정을 통해 만들어진다.

(1) 실행이력에 있는 노드에 대해 원시 프로그램의 정적 정보를 사용하여 아래에 있는 종속 그래프를 그린다.

① class 제어 종속 간선
② procedure 제어 종속 간선
③ method 제어 종속 간선
④ while 문 제어 종속 간선
⑤ if 문 제어 종속 간선
⑥ call 문에 의한 inter-procedure 간선
⑦ return 제어 종속 간선
⑧ 다형성 선택 종속 간선
⑨ 다형성 호출 종속 간선
⑩ 다형성 실행 종속 간선

(2) 실행이력에 있는 기준 노드로부터 자료 종속 간 선을 구한 후 이 간선이 그래프에서 path로 존재하지 않으면 그래프에 추가한다.

(3) 실행이력에 있는 기준 노드로부터 제어 종속 간 선을 구한 후 이 간선이 그래프에서 path로 존재하지 않으면 그래프에 추가한다. 그러나, 한번만 실행되는 "if" 제어 노드는 슬라이스에서 제외되고, 해당 "if" 노드에만 종속되는 노드들은 슬라이스에서 삭제된다. 제어 종속이 시작되는 노드는 다음과 같다.

① 실행이력에서의 기준노드에서 시작하여 기준 노드의 자료종속이 완료되는 노드까지의 구간에서 두 번 이상 중복해서 발생되는 "if" 제어 노드는 제어 종속의 시작노드가 될 수 있다.

② 기준 노드에서 시작하여 기준 노드의 자료 종속이 완료되는 노드까지의 구간에서 발생되는 노드들의 상위 level에 있는 반복 제어 노드는 제어 종속의 시작 노드가 될 수 있다.

슬라이싱 기준 C<a,b>에서 a는 기준노드이고 b는 노드 a에 있는 기준변수라고 하자. <a,b>는 <m,v>와 종속관계이고 m은 n의 선행자라고 할 때, <m,v>에 대한 슬라이스 기준 B와 슬라이스된 노드들의 집합 N을 표현하는 식은 다음과 같다. 위첨자 d는 자료종속을 나타내고, 위첨자 c는 제어종속을 의미한다.

(1) $B^{d}_{C\langle a,b\rangle}\langle m,v\rangle$

$$= \begin{cases} B^{d}_{C\langle a,b\rangle}\langle n,v\rangle & \text{if } v \notin \text{def}s(n) \\ B^{d}_{C\langle a,b\rangle}\langle n,x\rangle,\ n \bigcup N^{d} & others \\ \forall\, x \in refs(n) \end{cases}$$

(2) $B^{c}_{k}\langle m,v\rangle,\ where\ k = \sum C\langle a,b\rangle \forall\, a \in N^{d}$

$= B^{c}_{k}\langle n,x\rangle\ \forall\ x \in refs(n),\ n \bigcup N^{c}$

(3) $N = N^{d} \bigcup N^{c}$

프로그램을 구성하는 명령문의 구조는 프로그램 3과 같이 표현할 수 있다. 그리고, 본 논문에서 제시한 효율적으로 동적 객체지향 프로그램 슬라이스(*Mark*)를 산출하는 알고리즘은 알고리즘 1에 나타나 있고, 프로그램을 구성하는 명령문의 구조가 프로그램 3에 나타나 있다.

```
Program → Declarations Stmt_list
Stmt_list → Stmt; Stmt_list
Stmt      → Simple_stmt | If_stmt | While_stmt
Simple_stmt → Assgn_stmt | Read_stmt |
              Write_stmt
Assign_stmt → Var := Exp
Read_stmt  → read(Var)
Write_stmt → write(Var)
If_stmt    → if (Predicate_exp) then Stmt_list
             else Stmt_list
             end if
While_stmt → while (Predicate_exp)
                Stmt_list
             end while
Predicate_exp → Exp
Exp        → Exp Binary_op Exp | Unary_op Exp
             | Var | Const
```

프로그램 3. 프로그램을 구성하는 명령문의 구조

```
IDOPDG(<prevhist | Mark>) =
    SetVar' = Ins(Criterion, SetVar)
    DependCheck(Criterion, 1, 1)
    while k = 1, n
      DependCheck(IfCriterion, lastnum, RepeatUpper,
      CheckObject, 2)
    end while
    while k = 1, m
      SubSist(k, CheckObject, sist)
      if (sist = 1)
      then DependCheck(RepeatCriterion, 1, RepeatUpper,
                       CheckObject, 2)
      end if
    end while
  in (Criterion, IfCriterion, RepeatCriterion, RepeatUpper,
      CheckObject,sist, Mark', SetVar')
```

```
DependCheck(Criterion, lastnum, RepeatUpper, CheckObject, init) =
  let startnum = Criterion
      while  k = startnum, lastnum, -1
        if (NodeType(num) = "R")
        then Return (Ref, num, SetVar, sist, Mark, last),
        end if

        if (NodeType(num) = "A")
        then Assign(Mark, num, Def, Ref, SetVar, last),
        end if

        if (NodeType(num) = "I")
        then Input(Mark, num, Def, SetVar, last),
        end if

        if (NodeType(num) = "CE" or NodeType(num) = "M" or
          NodeType(num) = "P" or NodeType(num) = "C" or
          NodeType(num) = "N")
        then Ins(Mark(i), num),
             last = i
        end if

        if (init = 1 and RepeatUpper(num) not = " ")
        then CheckObject(x) = num
        end if

        if (init = 1 and NodeType(num) = "D")
           Criterion(n) = (Ref, num),
           Select(Mark, num, Ref, SetVar, sist)
        end if

        if (init = 1 and NodeType(num) = "L")
           Criterion(m) = (Ref, num),
           Repeat(Mark, num, Ref, SetVar)
        end if
      end while
      lastnum = last
  in (Criterion', lastnum', RepeatUpper, CheckObject', init)

Return (Ref, num, SetVar, sist, Mark, last)
        SubSist(Ref(num), SetVar)
        if (sist = 1)
        then Ins(Mark(i) , num),
             last = I
        end if
  in (Ref, num, SetVar', sist, Mark', last')

Assign(Mark, num, Def, Ref, SetVar, last)
      Ins(Mark(i), num),
      last = i,
      Del(Def(num), SetVar)
      if (Ref(num) not = " ")
      then Ins(Ref(num), SetVar)
      end if
  in (Mark', num, Def, Ref, SetVar', last')

Input(Mark, num, Def, SetVar, last)
      Ins(Mark(i), num),
      last = i,
      Del(Def(num) , SetVar)
  in (Mark', num, Def, SetVar', last')

Repeat(Mark, num, Ref, SetVar)
     SubSist(Ref(num), SetVar)
     if (sist = 1)
     then Ins(Mark(i), num),
          Ins(Ref(num), SetVar)
     end if
  in (Mark', num, Ref, SetVar')

Select(Mark, num, Ref, SetVar, sist)
     SubSist(Ref(num), SetVar)
     if (sist = 1)
     then Ins(Mark(i) , num),
          Ins(Ref(num), SetVar)
     end if
  in (Mark', num, Ref, SetVar', sist')

Ins(Var, SetVar) =
     Ins(Var, SetVar') = ∪x∈D Var( x ) ∪ ∪x∈D SetVar( x )
  in (Var, SetVar')

Del(Var, SetVar) =
     Del(Var, SetVar') = ∪x∈D SetVar( x ) - ∪x∈D Var( x )
  in (Var, SetVar')

SubSist(Var(x), SetVar) =
  let sist = 0
     if ( ∪x∈D Var( x ) ∩ Ux∈D SetVar( x ))
     then sist = 1
      end if
  in (Var, SetVar, sist')
```

알고리즘 1. 동적 객체지향 프로그램 슬라이스를 산출하는 알고리즘

3.4 프로그램 슬라이스 작성 단계

기준 변수에 대한 동적 슬라이스 추출을 위해 3.3에서 작성한 IDOPDG에서 기준 노드를 중심으로 역운행(back tracking)하여 해당되는 슬라이스를 추출하는 단계이다. 슬라이스된 프로그램은 주어진 입력에 대해 실행 가능한 완벽한 프로그램이다.

4. 프로그램 슬라이싱 기법의 비교

본 논문에서 제시한 개선된 동적 객체지향 프로그램 슬라이싱 기법과 동적 객체지향 프로그램 슬라이싱 기법 그리고 객체지향 프로그램 슬라이싱 기법들의 특성을 비교하면 다음과 같다.

4.1 기존의 동적 객체지향 프로그램 슬라이싱 기법

(1) 특징

기존의 동적 프로그램 종속 그래프 기법을 기반으로 하여 해당하는 객체지향 프로그램의 동적 객체지향 프로그램 종속 그래프를 만든 후 이 그래프를 depth-first 혹은 breadth-first 그래프 운행 알고리즘을 사용하여 동적 객체지향 프로그램 슬라이스를

산출한다.

(2) 장점

동적 객체지향 프로그램 종속 그래프를 사용하여 동적 객체지향 프로그램 슬라이스를 간단히 산출할 수 있다.

(3) 단점

프로그램에서 반복 제어 횟수가 많은 반복문이 포함되면 그래프의 복잡도가 그 횟수에 비례해서 복잡해진다.

4.2 개선된 동적 객체지향 프로그램 슬라이싱 기법

(1) 특징

기존의 기법에 비해 클래스간 인터페이스와 다형성 호출 등을 효율적으로 표현할 수 있는 동적 객체지향 프로그램 종속 그래프를 통해 동적 프로그램 슬라이스를 산출한다.

(2) 장점

기존의 객체지향 프로그램 기법들에 비해 프로그램 슬라이스의 크기가 작다. 그리고, 기존의 동적 객체지향 프로그램 슬라이싱 기법이 종속 그래프에서 depth-first 혹은 breadth-first 방식의 탐색 방법을 사용하였기 때문에 그래프가 복잡해지면 탐색이 복잡해질 수 있으나 개선된 동적 객체지향 프로그램 슬라이싱 기법은 전통적인 back tracking 방법을 통해 종속 그래프를 탐색함으로써 그래프의 복잡성과는 상관없이 탐색을 손쉽게 할 수 있다. 그리고, 사용되는 동적 객체지향 프로그램 종속 그래프는 복잡도의 크기가 기존의 기법에 비해 작아졌을 뿐만 아니라 특히 반복과 같은 제어문의 표현에 효율적이다.

5. 적용 사례

프로그램 1의 예제 프로그램에서 argv[1] = 3이고, 슬라이싱 기준이 (H, 39_{22}, which_floor)일 때, 동적 객체지향 슬라이스를 구하기 위해 본 논문에서 제안한 동적 객체지향 프로그램 슬라이싱 알고리즘에 적용시킨다.

5.1 프로그램 노드 분석 단계

프로그램 1의 예제 프로그램을 구성하고 있는 노드들의 자료 분석 테이블이 표 2에 나타나 있다.

표 2. 예제 프로그램의 노드 관련 자료

노드 번호	노드 type	DEF	REF	소속 모듈 번호	상위 제어 노드
1	CE	Elevator		1	
2	M	Elevator	1_top_floor	1	
3	A	current_floor		2	
4	A	current_direction		2	
5	A	top_floor	1_top_floor	2	
6	M	~Elevator		1	
7	M	up		1	
8	A	current_direction		7	
9	M	down		1	
10	A	current_direction		9	
11	M	which_floor		1	
12	R		current_floor	11	
13	M	direction		1	
14	R		current_direction	13	
15	M	go	floor	1	
16	D		current_direction	15	
17	L		current_floor, floor, top_floor	15	
18	C	add	current_floor	15	17
19	L		current_floor, floor	15	
20	C	add	current_floor	15	17
21	M	add	a, b	18	
22	A	a	a, b	21	
23	CE	AlarmElevator		23	
24	M	AlarmElevator	top_floor	23	
25	C	Elevator	top_floor	23	
26	A	alarm_on		23	
27	M	set_alarm		23	
28	A	alarm_on		27	
29	M	reset_alarm		23	
30	A	alarm_on		29	
31	M	go	floor	23	
32	D		alarm_on	31	
33	C	go	floor	31	
34	P	main	argc, **argv	34	
35	D		argv[1]	34	
36	N	AlarmElevator		34	
37	N	Elevator		34	
38	C	go		34	
39	W		which_floor	34	

5.2 프로그램 실행이력 분석 단계

프로그램 1의 예제 프로그램에서 argv[1] = 3일 때 실행 이력은 {34, 35, 37, 2, 3, 4, 5, 38, 15, 16, 17, 18, 21, 22, 17, 18, 21, 22, 17, 39, 11, 12}가 된다.

5.3 동적 객체지향 프로그램 종속 그래프 작성 단계

(H, 39_{22}, which_floor)를 슬라이싱 기준으로 했을 때 즉, 기준 노드 39번에 있는 슬라이싱 기준 변수 which_floor를 슬라이싱 기준 변수로 했을 때 IDOPDG를 생성한다.

(1) IDOPDG에서 class 제어 종속 간선은 (1→2), (1→11), (1→15)이고, procedure 제어 종속 간선은 (34→35), (34→38), (34→39)이다. 그리고, method 제어 종속 간선은 (2→3), (2→4), (2→5), (11→12), (15→16), (21→22)이고, while 문 제어 종속 간선은 (17→18) 그리고, if 문 제어 종속 간선은 (16→17), (35→37)이다. inter-procedure 제어 종속 간선은 (18→21), (37→2)이고, return 제어 종속 간선은 (3→37), (5→37), (12→11), (22→18)이다. 그리고, 다형성 선택 종속 간선은 (37→ϒ)이다. 다형성 호출 종속 간선은 (38→γ)이고 다형성 실행 종속 간선은 (γ→15)이다.

(2) 추가자료 종속 간선은 (18→12), (11→39)이다.

(3) 간선 (15→16)과 (16→17)은 (15→17)로 변경되고, 간선 (2→4)는 삭제된다. 그리고, 간선 (34→36)과 (36→37)은 (34→37)로 변경된다.

프로그램 1의 예제 프로그램에 대한 IDOPDG가 그림 2에 나타나 있다.

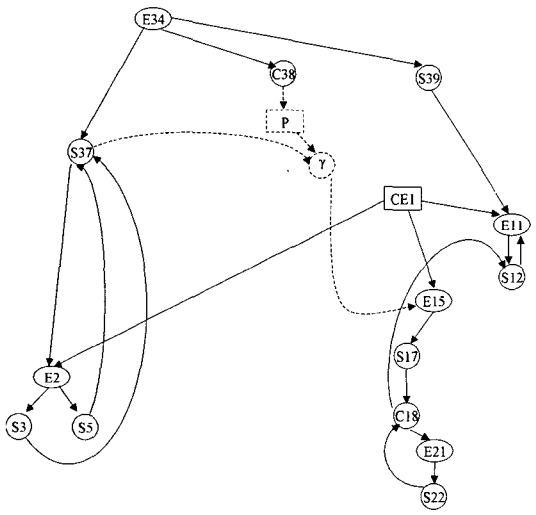


그림 2. 기준노드 39에 대한 개선된 동적 객체지향 프로그램 종속 그래프

5.4 프로그램 슬라이스 작성 단계

(H, 39_{22}, *which_floor*)를 슬라이싱 기준으로 했을 때 즉, 기준 노드 39번에 있는 슬라이싱 기준 변수 which_floor에 대한 동적 슬라이스를 구하기 위해 그림 2에 있는 IDOPDG를 역 운행한 결과 산출된 슬라이스 노드는 {1, 2, 3, 5, 11, 12, 15, 17, 18, 21, 22, 34, 37, 38, 39}가 된다. 이것은 구현 프로그램에 적용시킨 결과에서도 슬라이스 결과가 같음을 확인할 수 있다. 구현 프로그램을 통해 적용시킨 슬라이싱 결과가 그림 3에 나타나 있다.

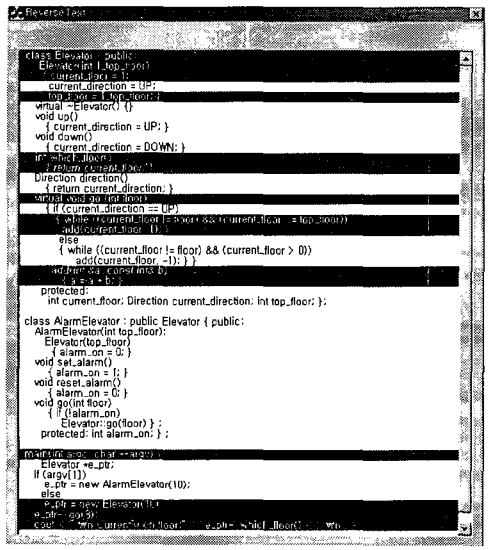


그림 3. 슬라이싱 결과

6. 효율성 분석

기존의 객체지향 프로그램 종속 그래프와 본 논문에서 제시한 동적 객체지향 프로그램 종속 그래프의 복잡도를 측정하면 다음과 같다. 그리고, 복잡도를 구하는 변수의 테이블이 표 3에 나타나 있다.

표 3. 복잡도를 구성하는 변수 테이블

변수 명	변수의 내용
p	procedure
pp	procedure의 parameter
pv	procedure 내에서의 일반 vertex
pc	procedure 내에서의 call
pcp	procedure 내에서의 call 문의 parameter
sgc	class 생성 호출
scv	class 생성 호출에 따른 변수
s	class
m	class 멤버인 method
mp	class 멤버인 method의 parameter
sc	class 내에서의 call
scp	class 내에서의 call 문의 parameter
sv	class 내에서의 일반 vertex

6.1 동적 객체지향 프로그램 종속 그래프의 복잡도

프로그램 1의 예제 프로그램에 대해 기존의 동적 객체지향 프로그램 종속 그래프 기법을 이용해 슬라이스를 구하면 프로그램 4와 같다. 기존의 동적 객체지향 프로그램 종속 그래프에 대한 복잡도ϕ가 표 4에 나타나 있다.

```
class Elevator {
public:
Elevator(int 1_top_floor)
 { current_floor = 1;
   current_direction = UP;
   top_floor = 1_top_floor; }
int which_floor()
 { return current_floor; }

virtual void go (int floor)
 { if (current_direction == UP)
    { while ((current_floor != floor) &&
             (current_floor <= top_floor))
      add(current_floor, 1); }
 }

private:
 add(int &a, const int& b)
  { a = a + b; };
protected:
 int current_floor;
 Direction current_direction;
 int top_floor;
};

 main(int argc, char **argv) {
  Elevator *e_ptr;
  if (argv[1])
     e_ptr = new AlarmElevator(10);
  else
     e_ptr = new Elevator(10);
  e_ptr->go(3);
  cout << "\n Currently on floor:"
  << e_ptr->which_floor() << "\n";
 }
```

프로그램 4. DOPDG를 이용한 슬라이스된 프로그램

표 4. 기존의 동적 객체지향 프로그램 종속 그래프의 복잡도

	동적 객체지향 프로그램 종속그래프 복잡도
procedure 종속	p+pc+sgc+3*pv
class 종속	s+sc+m+3*sv

Σ복잡도ϕ(DOPDG)
= 복잡도ϕ(procedure 종속)+복잡도ϕ(class 종속)
= p+pc+sgc+3*pv+s+sc+m+3*sv

6.2 개선된 동적 객체지향 프로그램 종속 그래프의 복잡도

프로그램 1의 예제 프로그램에 대해 본 논문에서 제시한 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG)의 복잡도 ϕ가 표 5에 나타나 있다.

표 5. 개선된 동적 객체지향 프로그램 종속 그래프의 복잡도

	IDOPDG 복잡도
procedure 종속	pv+p+2*(pc+sgc)
class 종속	s+sv+sc+m

Σ복잡도ϕ(IDOPDG)
= 복잡도ϕ(procedure 종속)+복잡도ϕ(class 종속)
= pv+p+2*(pc+sgc)+s+sv+sc+m

6.3 효율성 비교

프로그램 1의 예제 프로그램에 대하여 명령문 39번의 which_floor()를 기준으로 했을 때, 기존의 OPDG와 DOPDG 그리고, 본 논문에서 제시한 IDOPDG의 슬라이스의 크기를 비교하는 테이블이 표 6에 나타나 있고, 복잡도를 비교하는 테이블이 표 7에 나타나 있다.

(1) 슬라이스의 크기

기존의 OPDG 방식에서 산출한 프로그램 슬라이스의 결과는 {1, 2, 3, 4, 5, 11, 12, 15, 16, 17, 18, 21, 22, 34, 35, 37, 38, 39}이다. 그러므로 슬라이스의 크기는 18이다. 그러나, 본 논문에서 제시한 DOPDG 방식에 의한 프로그램 슬라이스의 결과는 {1, 2, 3, 5, 11, 12, 15, 17, 18, 21, 22, 34, 37, 38, 39}이다. 그러므로 슬라이스의 크기는 15이다.

표 6. 슬라이스의 크기를 비교하는 테이블

	슬라이스의 크기
동적 객체지향 프로그램 종속 그래프 (DOPDG)	18
개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG)	15

(2) 복잡도 비교

표 4에 있는 기존의 동적 객체지향 프로그램 종속

표 7. 복잡도를 비교하는 테이블

	최대 복잡도	실제 복잡도
동적 객체지향 프로그램 종속 그래프 (DOPDG)	73	22
개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG)	42	17

그래프의 복잡도를 구하는 공식과 표 5에 있는 개선된 동적 객체지향 프로그램 종속 그래프의 복잡도를 구하는 공식에 의해 최대 복잡도를 구하고, 그림 1과 그림 2에서 나타난 기존의 동적 객체지향 프로그램 종속 그래프와 개선된 동적 객체지향 프로그램 종속 그래프의 실제 복잡도를 구하면 다음과 같다.

기존의 객체지향 프로그램 종속 그래프에서 최대 복잡도와 실제 복잡도가 차이가 나는 이유는 호출에서 매개변수의 전부 변경되지 않을 수도 있기 때문이다. DOPDG에서는 while 문에 대한 반복 노드의 수 때문에 최대 복잡도와 실제 복잡도가 차이가 나며, 반복 노드의 수 때문에 IDOPDG에 비해 복잡도가 증가한다.

7. 결 론

정적 슬라이스는 주어진 기준변수에 영향을 끼치는 모든 노드이고, 동적 슬라이스는 주어진 프로그램 입력에 대해 발생된 변수 값에 실제적으로 영향을 주는 명령문들로 구성되어 있다. 그러므로 어떤 시험 사례를 통해 프로그램을 분석하는 디버깅 분야에서는 동적 슬라이싱이 정적 슬라이싱 보다 더 유용하게 사용될 수 있다. 본 논문에서는 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG)를 이용한 동적 슬라이싱 기법을 제안하였다. 기존의 DOPDG 기법과 본 논문에서 제시한 IDOPDG 기법을 사용하여 프로그램 1의 예제 프로그램을 본 논문에서 제시한 복잡도 측정 공식에 적용한 결과 IDOPDG의 복잡도가 42이고, 기존의 DOPDG의 복잡도는 69임을 알 수 있었다. 그리고, 이론 수치가 아닌 실제 그래프에 나타난 IDOPDG와 DOPDG의 복잡도 수치는 각각 17과 22로 나타났다. 그러므로 본 논문에서 제시한 IDOPDG 기법이 기존의 DOPDG 기법에 비해 복잡성을 줄여주는 효율적인 표현법임을 알 수 있었다.

우리가 실험한 특징들로 인하여 본 논문에서 제시한 IDOPDG 기법이 기존의 OPDG 기법 과 DOPDG 기법 보다 더 간략하고 프로그램 슬라이스의 크기도 작으므로 더 효율적임을 알 수 있었고, 이러한 장점을 더 잘 이해하고 정리하기 위해 더 많은 연구와 실험이 필요하다. 표현된 특징들의 유용성과 적합성을 결정할 수 있는 더 큰 프로그램에서 이를 시험해 보고 계속적으로 객체지향 프로그램 종속 그래프의 간략화 그리고 크기의 감소를 위해 꾸준히 연구해야 할 것이다.

참 고 문 헌

[1] Hiralal. Agrawal and J. R. Horgan. "Dynamic Program Slicing", Proc. ACM SIGPLAN'90 Conf. Programming Lang. Design and Implementaion, pp.246-256, 1990.

[2] B. Korel and J. Laski, "Dynamic Slicing in Computer Programs", The Journal of System and Software Engineering, vol.23, No.1, pp. 17-34, 1997.

[3] B. Korel and J. Laski, "Dynamic Program slicing", Information Proceeding Letters, vol. 29, No.3, pp.155-163, 1998.

[4] Park, S. H. and M. G. Park, "An efficient dynamic program slicing algorithm and its application.", Proc. of the IASTED International Conference, Pittsburgh, Pennsylvania, pp.459-465, May 1998.

[5] 박순형, 박만곤, "Dynamic Slicing using Dynamic Dependence Graph.", 한국멀티미디어학회 논문지, 제5권 제3호, pp.331-341, 2002. 6.

[6] M. Jean and C. Ning, "Reuse-Driven Interprocedural Slicing", Proceeding of the 20th International Conference on Software Engineering, pp.74-83, 1998. 4.

[7] Bodan Korel, "Computation on Dynamic Program Slices for Unstructured Programs", IEEE Trans. on Software Engineering, vol. 23, No. 1, pp.17-34, January 1997.

[8] Cheng J, "Slicing Concurrent Programs.", Proc. First Int'l Workshop Automated and

Algorithmic Debugging, Linkoping, Sweden, pp. 244-261, 1993.

[9] Loren D. Larsen and Mary Jean Harrold, "Slicing Object-Oriented Software.", Technical Report 95-103, Department of Computer Science, Clemson University, March 1995.

[10] Mangala Gowri Nanda, S. Ramesh, "Slicing concurrent programs.", International Symposium on Software Testing and Analysis (ISSTA), Portland, OR, USA, pp.180-190, 2000.

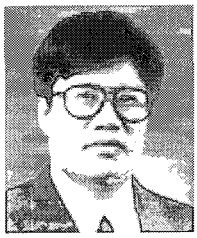
[11] Margaret Ann Francel, Spencer Rugaber, "The value of slicing while debugging.", Science of Computer Programming, Volume 40, Number 2-3, pp.151-169, July 2001.

[12] Zhengqiang Chen, Xu Baowen, "Slicing Object-Oriented Java Programs.", SIGPLAN Notices, Volume 36, 2001, Volume 36, Number 4, pp.33-40, April 2001.

[13] Robert M. Hierons, Mark Harman, Sebastian Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants.", Software Testing, Verification & Reliability (STVR), Volume 9, pp.233-262, 2000.

[14] Raghavan Komondoor, Susan Horwitz, "Using Slicing to Identify Duplication in Source Code.", Static Analysis (WSA/SAS), Paris, France, pp.40-56, 2001.

[15] J. Zhao, "Dynamic Slicing of Object-Oriented Programs.", Technical-Report SE-98-119, pp. 17-23, Information Processing Society of Japan (IPSJ), May 1998.



박 순 형

1981년 울산대학교 공과대학 전자계산학과(학사)
1985년 숭실대학교 대학원 전자계산학과(석사)
2003년 부경대학교 대학원 전자계산학과(박사)
1981년~1983년 현대미포조선(주) 전산실 근무
1987년~2000년 동의공업대학 전자계산과 교수
관심분야 : 소프트웨어공학 및 재공학, 소프트웨어 테스팅, 비지니스 프로세스 재공학, 정보시스템 분석 및 설계, 비주얼 프로그래밍 기법, 멀티미디어 정보시스템 개발방법론

박 만 곤

1976년 경북대학교 수학교육학과 졸업(이학사)
1987년 경북대학교 대학원 전산통계학과(이학박사)
1980년~1981년 경남정보대학 전자계산학과 교수
1990년~1991년 영국 리버풀대학교 전자계산학과 객원교수
1992년~1993년 미국 캔사스대학교 컴퓨터공학과 교환교수
1996년 호주 사우스 오스트레일리아대학교 컴퓨터 및 정보과학부 객원교수
1995년 몽골 컴퓨터매핑 전문가로 외무부 파견
1997년 중국 산동성 정부 정보시스템구축 전문가로 외무부에 의해서 파견
1997년~2002년 콜롬보플랜 기술자교육대학(Colmbo Plan Staff College, 필리핀 마닐라에 본부) 정보기술 및 통신학처장으로 정부파견
1981년~현재 부경대학교 컴퓨터 멀티미디어 공학부 교수
2002년~현재 콜롬보플랜 기술자교육대학(Colmbo Plan Staff College, 필리핀 마닐라에 본부) 총재
관심분야 : 소프트웨어공학 및 재공학, 소프트웨어 신뢰성 및 안전성공학, 비지니스 프로세스 재공학, 멀티미디어 정보시스템, 소프트웨어품질공학, 소프트웨어 메트릭스, 소프트웨어 테스팅 및 감사, 결함허용 소프트웨어 시스템