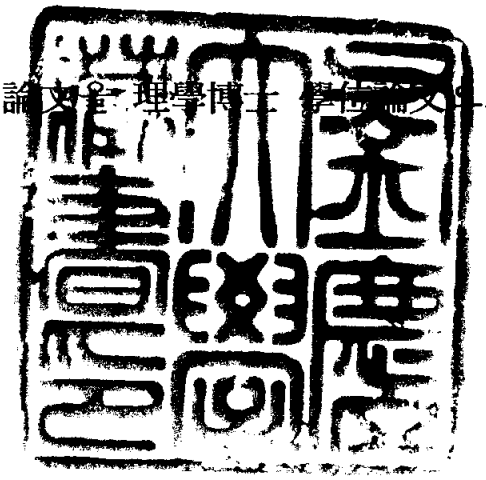
理學博士 學位論文

동적 객체지향 프로그램 슬라이싱에 관한 연구

指導敎授 朴 萬 坤



이 論文을 理學博士 學位論文으로 提出함

2003年 2月

釜 慶 大 學 敎 大 學 院

電 子 計 算 學 科

朴 淳 亨

朴淳亨의 理學博士 學位論文을 認准함

2002年 12月 日

主 審 工學博士 金 榮 鳳 

副 審 理學博士 朴 萬 坤 

委 員 理學博士 金 泰 和 

委 員 工學博士 余 定 模 

委 員 工學博士 梁 海 述 

목 차

그림 목차

표 목차

프로그램 목차

알고리즘 목차

A Study on the Dynamic Object-Oriented Program Slicing

Soon-Hyung Park

Department of Computer Science Graduate School of
PuKyong National University

ABSTRACT

Program slicing is a progress of finding all statements in a program P that may directly or indirectly affect the value of a variable var at a point p. Accordingly, program slicing is a useful technique with other applications in program debugging by providing other programs that gather statements relating to an interested variable in a program.

The slicing technique is classified by the two criteria. Firstly, it can be divided into static slicing and dynamic slicing by existence of execution history. Secondly, it can be divided into program slicing, system slicing and object-oriented program slicing by the number of programs that are objects of slicing.

Static slices are a set of nodes that affect criterion variables. Dynamic slices are a set of nodes that affect actually the values of variables tracing on the test case. Therefore we can use usefully a dynamic concept in the field of the debugging through a test case. Object-oriented program slicing is working to get slices of object-oriented program by tracing the flow of classes that is the core of object-oriented program and objects. Generally it is important that in the object-oriented program slicing we present polymorphism, dynamic binding, class inheritance, etc.

Traditional program slicing techniques often use graphs as a process of slicing to generate correct slices. But traditional dependence graphs especially object-oriented dependence graphs and dynamic object-oriented dependence graphs are complicated because that it need many vertexes and edges to represent data transmission inter procedures. So it is very difficult that programmer and tester use them to debug source programs. And size of the slices is larger.

We propose the representation of a 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. And 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.

Firstly, a step of the nodes analysis in program

Secondly, a step of the program execution history analysis

Thirdly, a step of the dynamic object-oriented program dependence graph generation

Finally, a step of the sliced program generation

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 is certifying that an improved dynamic object-oriented program dependence graph is more efficient in comparison with the traditional object-oriented dependence graphs(OPDG) and dynamic object-oriented program dependence graph(DOPDG).

An Improved Dynamic Object-oriented Program Dependence Graph(IDOPDG) proposed in this paper is similar to the Program Dependence Graph(PDG) in the respect that the graphs represent the control dependence information by the control dependence edges and the data dependence information by the data dependence edges at the

statements vertexes. The traditional object-oriented program dependence graphs is added the member variable edges, the call edges for construction of objects, the polymorphic call edges, the method call edges, etc. However, IDOPDG only is added the polymorphic call edges.

We find that the complexities of the IDOPDG is 42, the traditional complexities of the OPDG is 271 and the traditional complexities of the DOPDG is 69 with a result that we apply an example program to the formulas of the complexities using the traditional OPDG technique, the traditional DOPDG technique and the IDOPDG technique. As the result, the values of the actual complexities of the IDOPDG, OPDG and DOPDG are 17, 89 and 22 respectively. The size of the slices of the IDOPDG, OPDG and DOPDG are 15, 28 and 18 respectively.

제1장. 서론

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 [11][18][39]. 프로그램 슬라이싱 기법은 Mark Weiser에 의해 처음 제안되었고 프로그램 디버깅, 테스팅, 프로그램의 이해 그리고, 소프트웨어 유지보수에 유용하다[27][31][36][52].

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[28][57]. 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[1][14][21][22].

정적 프로그램 슬라이싱은 모든 가능한 프로그램 실행을 포함한다. 그러나, 디버깅에서 우리는 근본적으로 어떤 특별한 잘못된 실행을 처리하고 그 결과 그 실행의 오류의 원인이 있는 위치에 관심이 있다. 그러므로 입력 전체보다는 오히려 어떤 한 개의 주어진 입력에 대한 프로그램의 행위를 보존하는 슬라이스에 관심이 있다. 이러한 슬라이싱은 동적 슬라이싱과 관련이 있다. 동적 프로그램 슬라이싱은 소프트웨어 디버깅, 소프트웨어 유지보수, 프로그램 이해 그리고, 소프트웨어 테스팅에 유용하다 [1][5][7][13][56].

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. 일반적으로 객체지향 프로그램 종속 그래프에서는 다형성(polymorphism), 동적 바인딩(dynamic binding), 클래스 상속(class inheritance)등을 표현하는 것이 중요하다.

프로그램 슬라이싱의 적용 분야 중 프로그램 디버깅(program debugging) 분야에서의 프로그램 슬라이싱에 대한 필요성을 살펴본다.

프로그램 상의 오류를 일반적으로 fault 혹은 bug라고 한다. 디버깅(debugging)은 이러한 오류의 위치를 찾아 수정하는 작업이다. 그리고, 프로그램에서 fault에 관련된 부분만을 골라서 집약화시키는 기술은 프로그래머에게 오류의 발생 가능성에 대한 위치 정보를 제공해 줄 수 있는 주요한 기술이다. 이것을 일반적으로 프로그램 슬라이싱 기술이라고 한다.

그리고, 프로그램 오류를 발생시키는 error-revealing test case를 만들 수 있다면 error의 존재를 보다 쉽게 발견할 수 있을 것이다. 이것은 동적 프로그램 슬라이싱 기법과 관련이 있다.

3 개의 삼각형에 대한 세 변의 값을 입력으로 받아 각각의 넓이와 삼각형의 종류를 출력하는 프로그램이 (프로그램 1-1)에 나타나 있다. (프로그램 1-1)의 예제 프로그램에서 7번 명령문 AREA := (S * (S - E1) * (S - E2) * (S - E3)) ** 0.5 이 AREA := (S * (S - E1) * (S - E2) * (S - E1)) ** 0.5 로 잘못 코딩되어 있다. 만일, 세 변의 입력 값이 각각 (5, 5, 5), (5, 8, 5), (5, 8, 10)으로 주어졌다고 하자. 첫 번째 입력 값을 처리할 경우에는 AREA 값이 10.8로 프로그램 상에서 오류를 발견할 수 없게된다. 그리고, 두 번째 입력 값을 처리할 경우에도 AREA 값이 12.0으로 역시 프로그램 상에서 오류를 발견할 수 없게된다. 그러나, 세 번째 입력 값을 처리할 경우에는 AREA 값이 19.8로 나타나 실제 값 41.2와 다르게 나타나 프로그램 상의 오류가 발생하였음을 알 수 있다. 그러므로 17번의 write 명령문에 있는 변수 AREA에 관련된 명령문 즉, 오류를 발생시킬 가능성이 있는 명령문들만 따로 모아서 실행 가능한 프로그램을 만든다면 디버깅하는데 매우 유용할 것이다. 이와 같이 자료 입력 값에 대응하는 동적 프로그램 슬라이싱은 실제 프로그램 개발 환경에서 매우 적합한 적용기법이라고 할 수 있다.

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

기존의 종속 그래프 특히, 시스템 종속 그래프와 객체지향 종속 그

래프들은 프로시저간의 자료전달을 표시하기 위하여 많은 정점들과 간선들이 필요하기 때문에 매우 복잡하므로 실제로 프로그래머나 테스터들이 적용하기에는 매우 어렵다.

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. 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.

본 논문에서는 다음과 같은 방법으로 구성된다. 먼저 2 장에서는 기존의 프로그램 슬라이싱 기법에 대한 관련 연구들에 대해 소개하였으며 3 장에는 본 논문에서 제안한 동적 객체지향 프로그램 슬라이싱 기법의 기초 이론을 제공하기 위해 동적 시스템 슬라이싱을 위한 동적 시스템 종속

그래프를 제안하였으며 기존의 시스템 종속 그래프 기법과 비교하여 기술하였다. 4 장에서는 본 논문에서 제안한 동적 객체지향 프로그램 슬라이싱을 위한 동적 객체지향 프로그램 종속 그래프를 제안하였으며, 5 장에서는 본 논문에서 제시한 동적 객체지향 프로그램 슬라이싱을 위한 슬라이싱 구현 절차에 대해 기술하였고, 본 논문에서 제시한 기법과 기존의 객체지향 프로그램 슬라이싱 기법들의 특징과 장점, 단점에 대해 기술하였다. 마지막으로 6 장에서는 본 논문에서 제안한 동적 객체지향 지향 프로그램 슬라이싱 기법과 기존의 동적 및 정적 객체지향 프로그램 슬라이싱 기법과 비교한 다음 그 효율성에 대해 고찰하였다.

```
1:  begin
2:      I := 1;
3:      while (I <= 3)
        do
4:          I := I + 1;
5:          read (E1, E2, E3);
6:          S := 0.5 * (E1 + E2 + E3);
7:          AREA := (S * (S - E1) * (S - E2) * (S - E1)) ** 0.5;
8:          if (E1 == E2)
            then
9:              if (E1 == E3)
                then
10:                 TYPE := 'Equilateral';
                else
11:                 TYPE := 'Isosceles';
                end_if;
            else
12:             if (E1 == E3)
                then
13:                 TYPE := 'Isosceles';
                else
14:                 if (E2 == E3)
                    then
15:                     TYPE := 'Isosceles';
                    else
16:                     TYPE := 'Scalene';
                    end_if;
                end_if;
            end_if;
17:         write (AREA, TYPE);
        end_while;
    end.
```

(프로그램 1-1) 삼각형에 대한 예제 프로그램

(Program 1-1) Example program for triangle

제2장. 관련 연구

2.1 프로그램 슬라이싱의 정의

프로그램 슬라이싱(program slicing)은 프로그램에서 주어진 변수에 직접 혹은 간접적으로 영향을 미치는 명령문들의 집합인 프로그램 슬라이스(program slices)를 구하는 과정으로서 관련된 부분들에 대해 관련 범위를 좁혀주기 위한 효과적인 기술이다[53][54]. 프로그램 슬라이싱 기법은 다음과 같은 2 가지 기준에 의해 분류할 수 있다. 첫째, 입력 값에 따른 실행이력의 적용여부에 따라 정적 프로그램 슬라이싱과 동적 프로그램 슬라이싱으로 나눈다. 실행 이력이란 주어진 시험사례를 실행하는 동안 방문된 순서에 의한 일련의 정점들인 $\{v_1, v_2, \ldots\ldots, v_n\}$의 집합이다. 둘째, 슬라이싱 대상 프로그램의 수와 프로그램의 구성요소에 따라 프로그램 슬라이싱과 시스템 슬라이싱 그리고 객체지향 프로그램 슬라이싱으로 나눈다. 프로그램 슬라이싱은 시스템 슬라이싱의 개념을 포함할 수도 있으며, 특히 단일 프로그램을 의미할 때는 프로시저 슬라이싱(procedure slicing)이라는 용어를 쓰기도 한다.

프로그램 슬라이싱(일반적으로 정적 프로그램 슬라이싱)은 모든 가능한 프로그램 실행들을 포함한다. 그러나 디버깅 실행에서 우리는 근본적으로 어떤 특별한 잘못된 실행을 처리하고 그 결과로 나타나는 오류의 원인이 되는 위치에 관심이 있다. 그러므로 입력의 집합보다는 오히려 어떤 주어진 프로그램 입력에 대하여 해당 프로그램의 동작을 보존하는 슬라이

스에 더 관심이 있다. 이러한 슬라이싱 형태는 동적 프로그램 슬라이싱과 관련이 있다[23][50][58].

정적 프로그램 슬라이싱(static program slicing)은 어떤 입력에 대한 변수 발생의 값에 영향을 끼치는 모든 명령문들과 관련이 있기 때문에, 현재 입력 값에 영향을 끼치는 모든 명령문들에 관련된 동적 프로그램 슬라이싱(dynamic program slicing)이 실제 테스팅과 디버깅에 더 적합하다고 할 것이다. 결과적으로 보면 정적 프로그램 슬라이싱은 실행과 직접 관련이 없는 노드를 보유하게 된다[9][12].

정적 프로그램 슬라이싱은 제일 처음 Mark Weiser가 자료 흐름 분석과 제어 흐름 분석을 이용한 정적 프로그램 슬라이스를 산출하는 기법을 소개하였고[30][31], Ottenstein and Ottenstein은 프로그램 종속 그래프(program dependence graph)로 내부 절차적 슬라이스를 구축하는 알고리즘을 소개하였다[48][51]. 그리고, Horwitz는 시스템 종속 그래프(system dependence graph)를 사용하여 정적 프로그램 슬라이스를 구축하는 알고리즘을 소개하였다[22].

동적 슬라이싱은 Korel & Laski가 프로그램의 자료 종속과 제어 종속 개념을 이용한 동적 프로그램 슬라이싱 기법을 소개하였으며[11][13][60], Agrawal 과 Horgan이 프로그램 종속 그래프를 사용하여 보다 효율적인 동적 프로그램 슬라이싱 기법을 소개하였다[9][14][15].

객체지향 프로그램 슬라이싱은 Anand Krishunaswamy가 기존의 프로그램 종속 그래프 개념에 클래스 계층 구조 서브 그래프를 추가하여 객체지향 개념을 표현하였으며[5], Loren D. Larsen과 Mary Jean Harrold가 기존의 시스템 종속 그래프를 확장하여 객체지향 개념을 표현하였다

[2][40][41]. 그러나, 이들 객체지향 프로그램 슬라이싱 기법들은 호출 문맥을 표현하기 위한 매개변수 정점들의 수와 제어 종속을 표현하기 위한 영역 정점들의 수가 크게 증가하여 그래프가 복잡하다는 단점이 있다. 그리고, Jianjun Zhao는 동적 객체지향 종속 그래프 기법을 이용하여 객체지향 프로그램에 대한 동적 슬라이싱 방법을 제시하였다[24]. 그러나, 반복 제어의 횟수가 증가하면 그래프의 복잡도가 증가하는 단점이 있다.

2.2 정적 프로그램 슬라이싱

정적 프로그램 슬라이싱이란 슬라이싱 기준에서 주어진 프로그램의 변수 값에 영향을 줄 수 있는 모든 가능한 노드들 즉, 프로그램 슬라이스를 추출하는 동작이다.

2.2.1 Mark Weiser 기법

슬라이싱의 기준(criterion)을 C라 하면 C는 <V, i> 즉, 기준 변수 V와 V를 포함하는 명령문 i로 구성된다. 그리고, 명령문 n에서 슬라이싱 기준 C와 관련이 있는 변수 v의 집합을 $R^0_c(n)$이라고 하자. 이 때, 상위첨자 0은 관련성에 대한 간접성의 정도를 나타낸다.

$R^0_c(n)$를 구하는 방법은 첫 번째, n 과 i 가 같은 값을 가진다면 V는 v에 포함된다. 두 번째, 노드 n이 m의 선행자(immediate predecessor) 이고 어떤 변수가 DEF(n)과 $R^0_c(m)$에 동시에 존재한다면 v는 REF(n) 에 존재한다. 만일 v가 DEF(n)에 있지 않으면 v는 $R^0_c(m)$에 존재하게 된다.

$R^0_c(n)$에 의해 슬라이스에 포함된 명령문은 $S^0_c(n)$이라 하면 $S^0_c(n)$은 DEF(n)이 $R^0_c(m)$에 존재할 때이며, 이들 명령문의 집합이 슬라이스의 집합 즉, S^0_c가 된다.

$$S^{0}_{c} = all\ nodes\ n\ s.t.\ R^{0}_{c}(n+1) \bigcap DEF(n) \neq 0$$

그러나 이 때, S^0_c는 간접 영향을 끼치는 명령문은 포함되어 있지 않다. 간접적인 영향을 끼칠 수 있는 제어문과 같은 branch 명령문을 b 그리고, 이것을 보완하기 위해 INFL(b)는 b로부터 가장 가까운 역 지배자(inverse dominator) d까지의 경로 p에 존재하는 명령문들의 집합이라 하고, <b, REF(b)>로 정의된 branch 명령문의 기준을 BC(b)라고 한다면 슬라이스를 구하는 공식은 다음과 같다[16][31].

모든 $i \geq 0$ 에 대해서

$$R_C^{i+1}(n) = R_C^i \bigcup_{b \in B_C^i} R_{BC(b)}^0(n)$$

$$B_C^{i+1} = \bigcup_{n \in S_C^{i+1}} INFL(n)$$

$S_C^{i+1} = n \in B_C^i$ 또는 $R_C^{i+1}(n+1) \bigcap DEF(n) \neq 0$ 인 모든 노드 n

N은 모든 노드들의 집합이고 n_e는 exit 노드라고 하자. 그리고, m과 n은 N에 있는 2 개의 노드이고 m이 n으로부터 n_e까지의 모든 경로에 존재한다면 m은 n을 역 지배한다고 할 수 있다.

2.2.2 프로그램 종속 그래프 기법

프로그램 종속 그래프(PDG : Program Dependence Graph) 기법은 원시 프로그램을 자료 흐름 분석과 제어 흐름 분석을 통하여 프로그램 종속 그래프로 변환하여 슬라이스를 산출하는 방법이다. 프로그램 종속 그래프는 몇 가지 종류의 간선들에 의해 연결된 정점들의 방향성 그래프이다[51].

정점(vertex)은 간단한 명령문 (예를 들어 assignment, read, write)에 대해 한 개의 노드를 가지며, 각각의 제어 술부 표현(control predicate expression : 예를 들어 if-then-else, while-do내에서의 조건 표현)에 대해서도 한 개의 노드를 갖는다. 그리고, 노드들은 간선(edge)들로 연결된다. 간선은 프로그램 요소들간의 종속을 의미하며, 제어 종속 간선(control dependence edge) 과 자료 종속 간선(data dependence edge)으로 나눌 수 있다.

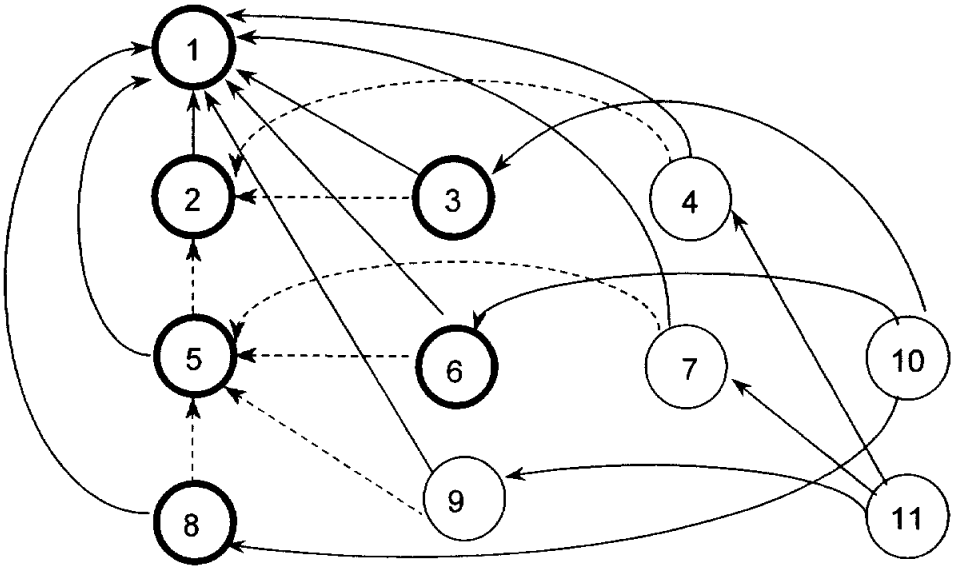
정점 v_i에서 정점 v_j까지의 제어 종속 간선은 정점 v_i에서 정점 v_j에 이르는 두 개의 path가 존재할 때이다. 정점 v_i에서 정점 v_j까지의 자료 종속 간선은 정점 v_j에서 실행된 결과가 v_i에 직접적으로 종속이 됨을 의미한다. 즉, 정점 v_i에서는 정점 v_j에서 정의된 변수를 사용한다.

(그림 2-1)은 (프로그램 2-1)의 예제 프로그램 2에 대한 프로그램 종속 그래프이다. 실선 간선은 자료 종속을 나타내고 점선 간선은 제어 종속을 나타낸다. 그리고 굵은 실선으로 표시된 노드는 예제 프로그램 2에서 명령문 10번에 있는 변수 Y를 슬라이스 기준으로 했을 때의 정적 슬라이스를 나타낸다.

```
        begin
S1:         read(X);
S2:         if (X < 0);
            then
S3:             Y := f1(X);
S4:             Z := g1(X);
            else
S5:             if (X = 0);
                then
S6:                 Y := f2(X);
S7:                 Z := g2(X);
                else
S8:                 Y := f3(X);
S9:                 Z := g3(X);
                end_if;
            end_if;
S10:        write(Y);
S11:        write(Z);
        end.
```

(프로그램 2-1) 예제 프로그램 2

(Program 2-1) Example Program 2



(그림 2-1) 예제 프로그램 2의 프로그램 종속 그래프

(Figure 2-1) PDG of Example Program 2

2.2.3 시스템 종속 그래프 기법

프로그램 종속 그래프는 하나의 프로시저 내에 있는 명령문 혹은 서술적 표현식에 대해 정점을 가지는 그래프이지만 시스템 종속 그래프(SDG : System Dependence Graph)는 각 프로시저를 위한 프로그램 종속 그래프의 집합이다[43].

전체 프로그램에 대한 일반적인 슬라이스 방법은 프로시저 호출의 경계(boundaries)들을 가로질러 얻어질 수 있다. 그러나, 이행적 종속 동작이 호출된 프로시저의 호출문맥을 정확히 설명하지 못하는 경우 명확한 슬라이스를 얻지 못하는 문제가 있다. 프로시저간의 슬라이싱 문제를 해결하기 위하여 프로시저들을 결합시키는 명백한 종속적 표현들을 가진 시스템 종속 그래프를 사용한다. 프로시저간의 슬라이싱은 프로시저 호출의 경계들을 연결하는 슬라이스들을 포함한다[4].

시스템 종속 그래프는 프로그램 종속 그래프에 비해 5 가지의 새로운 종류의 정점들을 사용함으로써 표현된다. 호출문은 *call-site* 라는 정점을 사용한다. 정보전달은 4가지 종류의 *인수* 정점을 사용함으로 표현된다. 호출 정점에서 볼 때, 정보전달은 *actual-in* 과 *actual -out* 이라 불리는 정점들의 집합에 의하여 표현되어진다. 유사하게, 호출된 프로시저에서의 정보전달은 *formal-in* 과 *formal-out* 이라 불리는 정점들의 집합에 의해서 표현된다. 이러한 모형을 사용한, 프로시저간의 자료종속은 *actual-in* 정점으로부터 *formal-in* 정점까지 그리고, *formal-out* 정점으로부터 *actual-out* 정점까지의 종속으로 제한되어진다.

시스템 종속 그래프에서의 간선은 프로그램 종속 그래프에서의 간선에

서 3가지 종류의 새로운 간선들을 추가한 것이다. : (1) *call* 간선은 각각의 호출문 정점에서 그것과 일치된 프로시저의 entry 정점까지 연결된다. (2) *parameter-in* 간선은 호출문에서의 각각의 *actual-in* 정점으로부터 그와 일치된 호출된 프로시저에서의 *formal-in* 정점까지 연결된다. (3) *parameter-out* 간선은 호출된 프로시저에서의 각각의 *formal-out* 정점으로부터 그와 일치된 호출문에서의 *actual-out* 정점까지 연결된다. 프로그램 종속 그래프와 비교하여 *call* 간선들은 새로운 종류의 제어종속선이다. 그리고, *parameter-in* 과 *parameter-out* 간선들은 새로운 종류의 자료종속선이다[22][24].

그러나, 매개변수 개수의 2 배만큼의 정점들이 추가되기 때문에 전체 정점들의 수가 증가되고, 또 그들간에 간선으로 연결되기 때문에 그래프가 복잡해질 수 있다.

2.3 동적 프로그램 슬라이싱

동적 프로그램 슬라이싱(dynamic program slicing)은 프로그램의 입력 자료의 값에 의해 발생되는 프로그램의 실행 이력을 추적한 다음 추적궤도(trajectory) 상에서 실행위치 q에서의 기준 변수에 실제적으로 영향을 주는 모든 명령문들을 산출하는 동작이다[29][55]. 그러므로 동적 슬라이싱을 하기 전에 실행이력 H에 대해 실행된 프로그램 P의 동적 슬라이싱 기준 $C = (H, I_q, V)$를 먼저 정의한다. I는 실행 이력 H에서 position q의 명령문이다. V는 I_q에 있는 변수의 집합이다. 슬라이싱 기준 C에 대한 프로그램 P의 동적 슬라이스는 0 개 이상의 명령문을 삭제함으로써 P

로 부터 얻어지는 정확하고 실행 가능한 프로그램 P'이다[7][10][59].

입력 x에 대한 실행 이력 H_x 내에 있는 position p에서 노드 Y 즉, $H_x(p) = Y$는 Y_P로 표현되며 하나의 명령에 대한 실행동작(action)을 의미한다. (프로그램 2-2)의 예제 프로그램 3에서 N = 2일 때 실행 이력 H_x에 대해 $H_x(9) = 5$, $H_x(14) = 9$로 나타낼 수 있다. 그리고, 상위 첨자를 사용함으로써 실행 이력에서 같은 노드의 다중 발생을 구별할 수 있다. 따라서, 실행이력 H_x는 $\{1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 5^2, 6^2, 7^2, 8^2, 5^3, 9^1\}$이 된다. 예를 들어 (프로그램 2-2)의 실행 이력 H_x에서 5^1와 5^2 그리고 5^3은 같은 노드 5를 포함하는 세 개의 실행동작(action)이다.

```
        begin
S1:         read(N);
S2:         Z := 0;
S3:         Y := 0;
S4:         I := 1;
S5:         while (I <= N)
            do
S6:             Z := f1(Z, Y);
S7:             Y := f2(Y);
S8:             I := I + 1;
            end_while;
S9:         write(Z);
        end.
```

(프로그램 2-2) 예제 프로그램 3

(Program 2-2) Example Program 3

2.3.1 Korel & Laski 기법

하나의 슬라이싱 기준에 대한 동적 슬라이스의 계산을 위한 Korel & Laski 동적 슬라이싱 알고리즘이 (알고리즘 2-1)에 나타나 있다. 알고리즘의 처리순서는 다음과 같다. 맨 처음 이 알고리즘에서 모든 동작들은 맨 처음 마크되지 않고 방문되지 않은 상태에서 초기화된 후 기준변수 I_q의 가장 최근 정의된 노드를 찾아 마크한다. 그 다음 while-loop의 각 반복에 대해 마크만 되고 방문되지 않는 하나의 동작 X^k를 선택한다. 그리고, 방문되어졌음을 표시한 후 X^k에서 사용된 모든 변수들의 최근 정의를 확인한 후 마크한다. 그 다음 단계는 동작들 사이에서 자료종속(Z^t)을 발견하는데 대응한다. 그리고, X^k에 대해 제어종속이 존재하는 모든 동작들은 마크한다. 그 다음 노드 X를 마크한다. 그리고, while-loop에 의해 마크된 모든 동작들이 전부 방문될 때까지 위 과정을 반복한다. 그리고, 위 과정의 반복이 종결된 후 마크된 노드들의 집합이 동적 슬라이스의 결과이다[8][10][13].

1: 입력 x에 대한 프로그램 P를 실행하여 위치 q까지의 실행 이력 T_x를 기록한다.
2: T_x 에 있는 모든 동작들을 마크되지 않고 방문되지 않은 상태로 초기화시킨다.
3: I^q의 가장 최근 정의 Y^v를 찾은 후 Y^v를 마크한다.
4: **while** T_x에 마크는 되었으나 방문되지 않는 동작이 존재한다.
do
5: T_x 에서 마크는 되었으나 방문되지 않는 한 개의 동작 X^k를 선택한다.
6: X^k는 방문되었음을 표시한다.
7: $v \in U(X^k)$를 만족하는 모든 변수들에 대해 v의 최근 정의 (last definition) Y^v를 찾은 후 마크한다.
8: Z^t 와 X^k 사이에서 하나의 제어 종속이 존재하는 그러한 모든 동작 Z^t를 마크한다.
9: 노드 X에 대한 모든 동작들을 마크한다.
10: **end while**
11: T_x에 있는 마크되지 않는 모든 동작들의 노드를 제거한 후 P로 부터 구축된 동적 슬라이스를 보여준다.

(알고리즘 2-1) Korel & Laski 동적 슬라이싱 알고리즘

(Algorithm 2-1) Korel & Laski's dynamic slicing algorithm

2.3.2 Agrawal & Hogan 기법

주어진 실행이력에 대한 변수에 관해 동적 슬라이스를 구하기 위해 Agrawal 과 Horgan이 제안한 2 가지 기법을 동적 종속 그래프를 통해 설명한다[9][14][15].

(1) 기법 1

실행 이력에서 어떤 한 명령문의 각 발생들에 대해서 해당 명령문에 종속성 간선을 가진 단지 한 개의 명령문만을 새로운 노드로 만든다.

(2) 기법 2

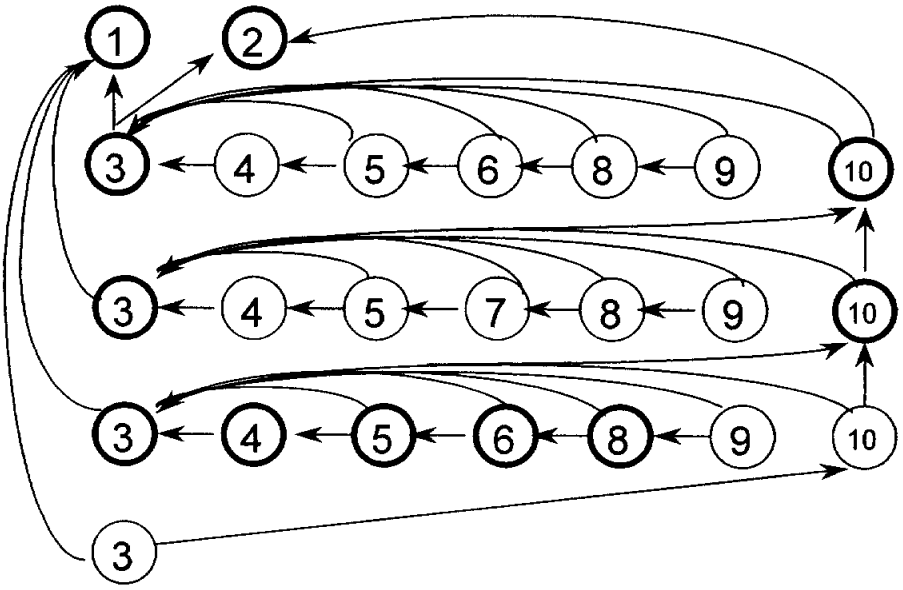
실행 이력에서 명령문의 모든 발생에 대해 새로운 노드를 만드는 대신 동일한 종속성을 가진 또 다른 노드가 존재하지 않을 경우에 한해서 새로운 노드를 만든다.

(프로그램 2-3)의 예제 프로그램 4의 시험 사례로서 N = 3 이고 X = {-4, 3, -2} 일 때 변수 Z에 대한 동적 슬라이스를 산출한다. 이 때, 실행 이력은 {1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 8^1, 9^1, 10^1, 3^2, 4^2, 5^2, 7^1, 8^2, 9^2, 10^2, 3^3, 4^3, 5^3, 6^2, 8^3, 9^3, 10^3, 3^4}가 된다. 기법 1을 적용시킨 결과는 (그림 2-2)에 나타나 있다. 그리고, 기법 2를 적용시킨 결과는 (그림 2-3)에 나타나 있고, 특히 이 그래프를 축소 동적 종속 그래프(RDDG : Reduced Dynamic Dependence Graph)라고 한다.

```
        begin
1:          read(N);
2:          I := 1;
3:          while (I <= N)
            do
4:              read(X);
5:              if (X < 0)
                then
6:                  Y := f1(X);
                else
7:                  Y := f2(X);
                end_if
8:              Z := f3(Y);
9:              write(Z);
10:             I := I + 1;
            end_while;
        end.
```

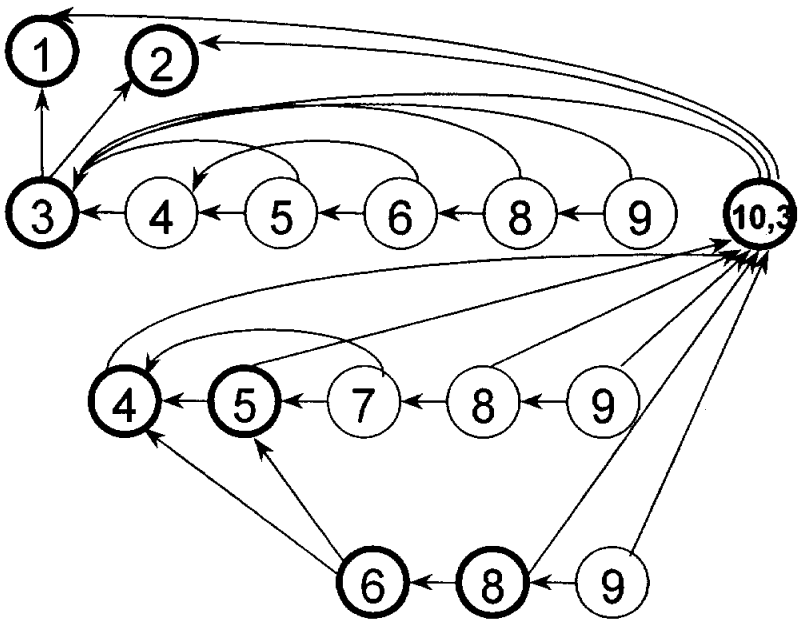
(프로그램 2-3) 예제 프로그램 4

(Program 2-3) Example Program 4



(그림 2-2) 예제 프로그램 4의 동적 종속 그래프

(Figure 2-2) DDG of Example Program 4



(그림 2-3) 예제 프로그램 4의 축소 동적 종속 그래프

(Figure 2-3) RDDG of Example Program 4

2.4 객체지향 프로그램 슬라이싱

2.4.1 객체지향 프로그램을 위한 시스템 종속 그래프

객체지향 프로그램을 위한 시스템 종속 그래프 기법은 기존의 시스템 종속 그래프를 객체지향 개념을 표현할 수 있게 확장한 기법으로 시스템에서의 각 클래스를 표시하기 위한 클래스 종속 그래프(CLDG : CLass Dependence Graph)와 호출 환경을 위한 프로시저 종속 그래프(procedure dependence graph)로 구성된다.

클래스 종속 그래프(CLDG)에서 각 메소드는 메소드 진입을 위한 메소드 진입 정점(method entry vertex)을 가지며, 클래스와 메소드를 연결하는 클래스 멤버 간선(class member edge)과 클래스 진입을 위한 클래스 진입 정점(class entry vertex)을 가진다. 그리고, 메소드에 있는 각 formal parameter를 위한 formal-in 정점과 메소드에 의해 수정될 각 formal reference parameter를 위한 formal-out 정점을 가진다. 그리고, 메소드에 의해 참조될 gobal 변수는 class의 instance 변수와 함께 formal-in parameter와 formal-out parameter를 가진다.

클래스 종속 그래프에서의 함수 호출(function call)은 호출 정점으로 표시되며, 이 함수 호출은 return 명령문의 값에 대해 자료에 종속되며, return 명령문은 parameter-out 간선을 사용하여 호출 정점과 연결된다 [46].

다형성 호출(polymorphic call)은 클래스에 대해 간접적인 참조가 만

들어 질 때 그리고 참조되어지고 있는 오브젝트 타입이 컴파일 시간에 알려지지 않을 때 발생한다. CLDG는 다형성 호출의 목적지를 표현하기 위해 다형성 선택 정점을 사용한다. 다형성 호출의 호출 정점은 다형성 선택 정점에 대해 간선을 가진다. 다형성 선택 정점은 다형성 호출의 모든 가능한 목적지에 대해 호출을 표현하는 서브 그래프에 대해 간선을 가진다.

호출 정점을 메소드 진입 정점에 그리고 actual-in 정점을 formal-in 정점에 또한 formal-out 정점을 actual-out 정점에 연결을 함으로써 완전한 시스템 종속 그래프를 구축할 수 있다.

기존의 시스템 종속 그래프 알고리즘에서의 슬라이싱 기준은 <p, x>이며, 위치 p에서 정의되거나 사용된 명령문 p에 대한 변수 x를 의미한다. 그러나, 객체지향 시스템 종속 그래프 기법에서의 슬라이싱 기준 <p, x>는 명령문 p에서의 변수 혹은 메소드 호출 x를 의미한다. 만일 x가 변수라면 그것은 p에서 정의되거나 사용되어야 하고, 만일 메소드 호출이라면 그것은 위치 p에서 호출되어야 한다.

다음에 설명하는 두 단계를 통해 슬라이스의 결과를 구할 수 있다. 첫 번째 단계는 슬라이스 기준 정점으로부터 parameter-out 간선을 제외한 모든 간선을 역방향으로 추적하여 도달된 모든 정점들을 찾아 표시한다. 두 번째 단계는 첫 번째 단계에서 표시된 정점들을 고려하여 call과 parameter-in 간선을 제외한 모든 간선을 역방향으로 추적하여 도달된 모든 정점들을 표시한다. 슬라이스는 첫 번째 단계와 두 번째 단계 동안에 표시된 정점들의 합이다[41].

2.4.2 객체지향 프로그램 종속 그래프

객체지향 프로그램 종속 그래프(OPDG : Object-oriented Program Dependence Graph) 기법은 기존의 프로그램 종속 그래프의 개념을 기초로 하여 객체지향 프로그램을 표현한 기법으로 클래스 계층 서브그래프(CHS : Class Hierarchy Subgraph) 와 제어 종속 서브그래프(CDS : Control Dependence Subgraph), 그리고 자료 종속 서브그래프(DDS : Data Dependence Subgraph)로 구성된다[5].

CHS는 기초적인 표현으로 클래스와 클래스 정의 안에 있는 메소드들 사이에서 상속관계의 표현이다. 각 CHS는 각 클래스를 위한 클래스 헤더 정점과 클래스에 정의된 각 메소드를 위한 메소드 헤더 정점을 가진다. CHS에서 간선은 각 클래스 헤더와 상속 관계를 가진 다른 클래스의 헤더를 연결한다. 메소드는 그들이 정의된 클래스를 위한 클래스 헤더에 연결된 그들의 헤더에 의해 표현된다.

CDS는 각 메소드를 위한 구현을 위한 표현을 포함한다. DDS는 객체에 대해 표현을 하며 특히 동적인 분석을 위해 객체에서 특정 메소드에 대한 동적으로 바운딩하는 메시지에 대해 표현한다.

제3장. 동적 시스템 종속 그래프의 제안

객체지향 프로그램 슬라이싱을 위한 객체지향 프로그램 종속 그래프는 2.2.3에서 설명한 시스템 종속 그래프(SDG : System Dependence Graph) 개념을 바탕으로 하고 있다. 그러나 기존의 시스템 종속 그래프는 매우 복잡한 구조로 되어있으므로 기존의 시스템 종속 그래프 개념보다 훨씬 복잡성이 줄어들어 보다 개선된 동적 시스템 종속 그래프(DSDG : Dynamic System Dependence Graph) 기법을 제안한다. 본 논문에서 제안하는 동적 시스템 종속 그래프 기법은 동적 슬라이싱 개념을 기초로 하여 시스템 종속 개념을 표현할 수 있도록 하는 종속 그래프 기법이다. 이 새로운 기법은 다음 장에서 제시할 동적 객체지향 프로그램 슬라이싱(dynamic object-oriented program slicing)을 위한 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG : Improved Dynamic Object-oriented Program Dependence Graph)에 적용된다.

3.1 기존의 시스템 종속 그래프

기존의 시스템 종속 그래프는 시스템의 주 프로그램을 표현한 프로그램 종속 그래프와, 시스템의 보조 프로시저를 표현한 프로시저 종속 그래프, 그리고 약간의 추가 간선들을 포함한다. 추가 간선들은 두 가지로 분류된다. (1) 호출문과 호출된 프로시저 사이의 직접 종속을 표현한 간선. (2) 호출에 따른 이행적 종속을 표현한 간선.

그리고, 기존의 시스템 종속 그래프에서 정점은 크게 프로시저 노드(procedure node)와 매개변수 노드(parameter node)로 나눌 수 있고 간선은 제어 간선, 자료 간선, 프로시저 내부 흐름(intra-procedure flow) 간선 그리고 프로시저 간 흐름(inter-procedure flow) 간선으로 나눌 수 있다. 기존 SDG의 정점 표기법과 간선 표기법이 (표 3-1)과 (표 3-2)에 나타나 있다.

procedure node	procedure name vertex	pv	
	general vertex	gv	
	call vertex	cv	
parameter node	call parameter	cp	
	procedure parameter	pp	

(표 3-1) 기존 SDG의 정점 표기법

(Table 3-1) Notation of vertexes of established SDG

control, data	
intra-procedure flow	
inter-procedure flow	

(표 3-2) 기존 SDG의 간선 표기법

(Table 3-2) Notation of edges of established SDG

그러나, 매개변수 개수의 2배만큼의 정점들이 추가되기 때문에 전체 정점들의 수가 증가되고, 또 그들간에 간선으로 연결되기 때문에 그래프가 복잡해지는 단점이 있다. (프로그램 3-1)의 예제 프로그램 5에 대한 시스템 종속 그래프가 (그림 3-1)에 나타나 있다.

```
1:  program Main
2:    x=1
3:    y=1
4:    i=1
5:    while i<=2 do
6:        call A(x,y,z,i)
      end while
7:    write z
    end
```

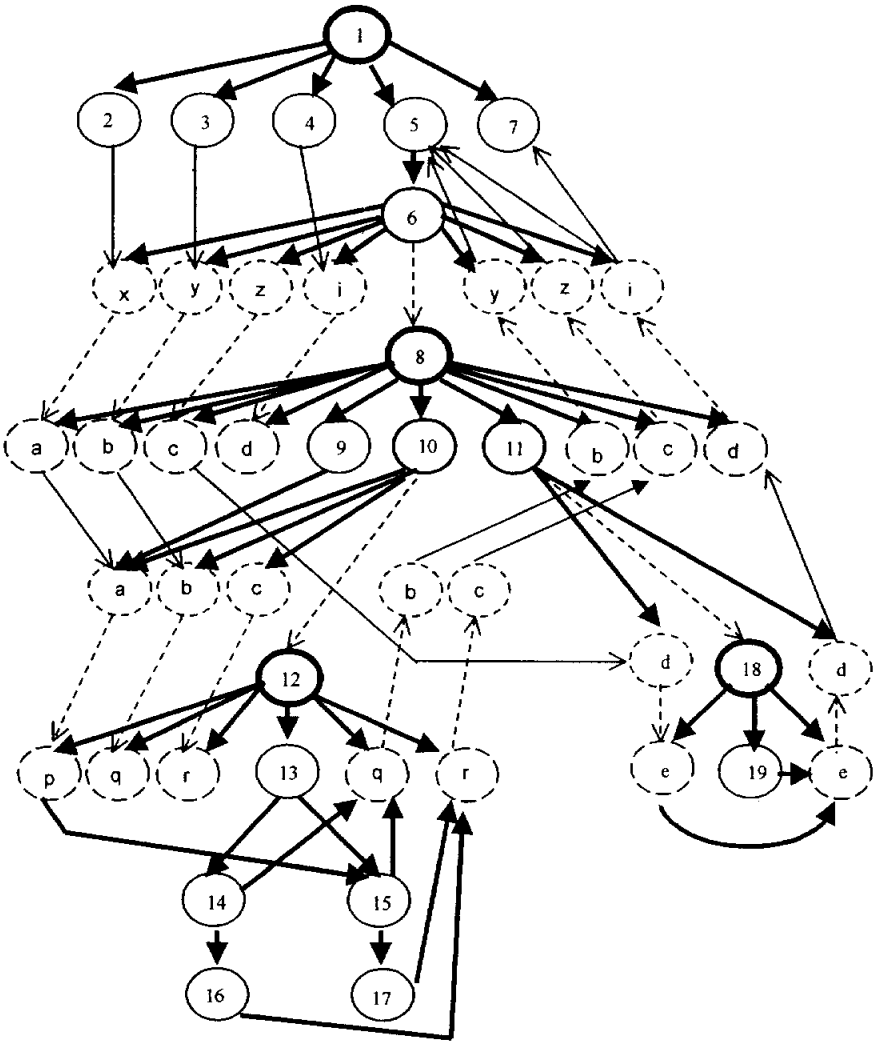
```
8:  procedure A(a,b,c,d)
9:    read a
10:   call IFF(a,b,c)
11:   call Increment(d)
    end
```

```
12: procedure IFF(p,q,r)
13:   if p<0 then
14:     q=f1(p)
15:     r=f2(q)
      else
16:     q=f3(p)
17:     r=f4(q)
      end if
    end
```

```
18: procedure Increment(e)
19:   e=e+1
    end
```

(프로그램 3-1) 예제 프로그램 5

(Program 3-1) Example Program 5



(그림 3-1) 예제 프로그램 5의 시스템 종속 그래프

(Figure 3-1) SDG of Example Program 5

3.2 동적 시스템 종속 그래프

본 논문에서 제안하는 동적 시스템 종속 그래프(DSDG : Dynamic System Dependence Graph)는 프로그램의 정적 정보 와 동적 정보를 이용하여 기존의 시스템 종속 그래프 보다 간결하게 그래프를 표현하는 기법이다.

DSDG는 다음 과정을 통해 만들어진다.

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

① procedure 제어 종속 간선
② while 문 제어 종속 간선
③ if 문 제어 종속 간선
④ call 문에 의한 inter-procedure 간선

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

(3) 실행이력에 있는 기준 노드로부터 제어 종속 간선을 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 시

작되는 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.

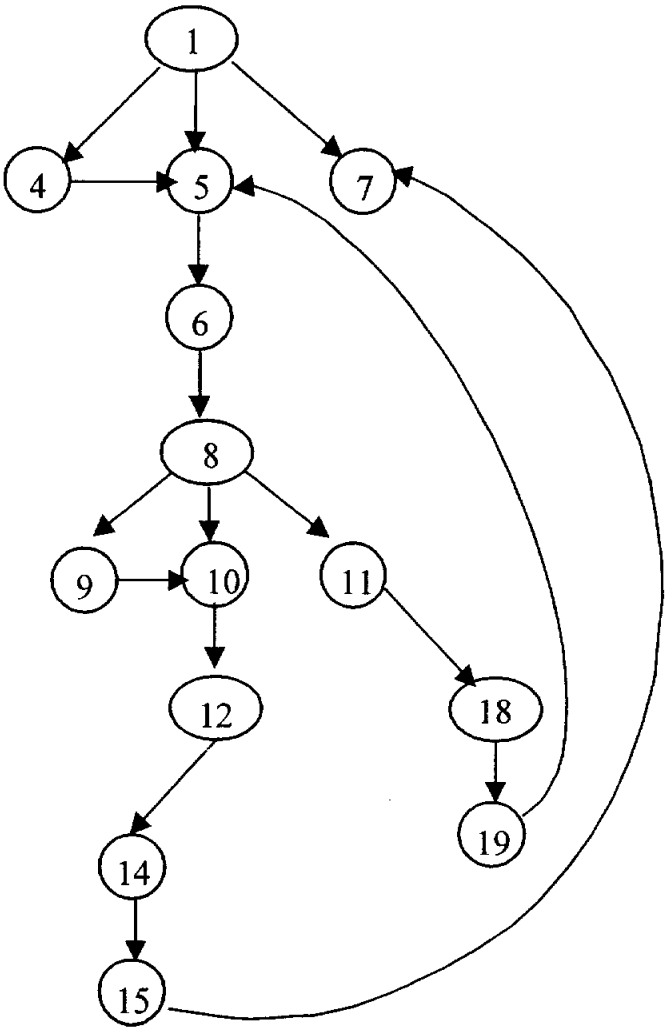
(프로그램 3-1)의 예제 프로그램 5에서 입력 값을 a = {3, -2}로 주었을 때 실행이력은 {1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 13, 16, 17, 11, 18, 19, 5, 6, 8, 9, 10, 12, 13, 14, 15, 11, 18, 19, 5, 7}이 된다.

슬라이싱 기준(criterion)은 (H, I_q, v)로 표시한다. 여기에서, H는 실행이력을 나타내고, I_q는 실행이력 q번째 노드를 의미한다. 그리고, v는 노드 I_q에 있는 변수로서 I_q의 부분 집합이다. 기준 노드를 (H, 7_{30}, z)라고 하자. 즉, 실행이력에서 30번째 있는 노드 7의 z를 기준 변수라고 했을 때 DSDG를 구해보자.

DSDG에서 procedure 제어 종속 간선은 (1→4), (1→5), (1→7), (8→9), (8→10), (8→11), (12→14), (18→19) 이고, while 문 제어 종속 간선은 (5→6) 그리고, inter-procedure 제어 종속 간선은 (6→8), (10→12), (11→18)이다. 자료 종속 간선은 (9→10), (14→15), (15→7) 이고 제어 종속 간선은 (4→5) 와 (19→5) 이다.

위 기준 변수를 실행이력 상에서 위의 조건을 만족하는 동적 시스템

종속 그래프를 그리면 (그림 3-2)와 같다. 앞에서 정의한 슬라이싱 기준을 (그림 3-2)에 있는 동적 시스템 종속 그래프에 적용하면 {1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 18, 19}를 슬라이스 결과로 얻을 수 있다. (프로그램 3-2)에는 슬라이스 기준 (H, 7_{30}, z)을 (프로그램 3-1)의 예제 프로그램 5에 적용했을 때 그 결과를 얻을 수 있는 슬라이스된 프로그램이 나타나 있다.



(그림 3-2) 예제 프로그램 5의 동적 시스템 종속 그래프

(Figure 3-2) DSDG of Example Program 5

```
program Main
  i=1
  while i<=2 do
     call A(x,y,z,i)
  end while
  write z
end

procedure A(a,b,c,d)
  read a
  call IFF(a,b,c)
  call Increment(d)
end

procedure  IFF(p,q,r)
  q=f1(p)
  r=f2(q)
end

procedure Increment(e)
  e=e+1
end
```

(프로그램 3-2) 슬라이스 기준 (H, 7_{30}, z)에 대한 슬라이스된 프로그램

(Program 3-2) Sliced program of criterion (H, 7_{30}, z)

3.3 시스템 종속 그래프의 복잡도

기존의 시스템 종속 그래프(SDG)의 정점들에 대한 복잡도 φ는 다음과 같다.

$\cdot \sum \varphi(SDG)$

$= pv + gv + cv + (cp * 2) + (pp * 2)$

본 논문에서 제시한 동적 시스템 종속 기법(DSDG)에 대한 종속 그래프의 복잡도 φ는 다음과 같다.

$\cdot \sum \varphi(DSDG)$

$= pv + gv + cv$

복잡도 산출을 위한 변수들의 종류가 (표 3-3)에 나타나있다.

변수 명	변수의 내용
pv	procedure vertex의 수
pp	procedure의 parameter의 수
gv	procedure 내에서의 일반 vertex의 수
cv	procedure 내에서의 call vertex의 수
cp	procedure 내에서의 call 문의 parameter의 수

(표 3-3) 복잡도 산출을 위한 변수 테이블

(Table 3-3) Table of variables for computation of complexities

(그림 3-1)의 기존의 시스템 종속 그래프와 (그림 3-2)에 있는 본 논문에서 제시한 동적 시스템 종속 그래프의 복잡도를 측정하기 위해 (프로그램 3-1)의 예제 프로그램 5에 복잡도 공식을 대입하면 (표 3-4)와 같은 복잡도 테이블이 산출된다.

(표 3-4)에서 보는 바와 같이 이론상의 기존 시스템 종속 그래프의 복잡도는 51이나 실제로는 47이다. 실제 복잡도가 이론상의 복잡도 보다 작은 이유는 호출에서 매개변수의 값이 전부 변경되지 않을 수도 있기 때문이다. 본 논문에서 제시한 동적 시스템 종속 그래프의 복잡도는 이론상으로는 19이나 실제로는 14이다. 이 이유는 실제 슬라이스에 포함되지 않는 프로그램의 명령문 노드가 발생되기 때문이다. 그리고, 슬라이싱 후에

얻을 수 있는 슬라이스의 크기도 시스템 종속 그래프 기법의 결과로 얻어진 슬라이스의 크기가 19인데 비해 동적 시스템 종속 그래프 기법으로 추출된 슬라이스의 크기는 14이다. SDG와 DSDG의 복잡도와 슬라이스의 크기가 (표 3-4) 와 (표 3-5)에 나타나있다.

프로그램순번	pv	gv	cv	pp	cp
1	1	5	1	0	4*2
2	1	1	2	4*2	4*2
3	1	5	0	3*2	0
4	1	1	0	1*2	0

(표 3-4) 예제 프로그램 5의 복잡도

(Table 3-4) Complexities of example program 5

	최대 복잡도	실제 복잡도	슬라이스의 크기
시스템 종속 그래프(SDG)	51	47	19
동적 시스템 종속 그래프(DSDG)	19	14	14

(표 3-5) SDG와 DSDG의 복잡도와 슬라이스의 크기

(Table 3-5) Complexities and size of slices of SDG와 DSDG

제4장. 개선된 동적 객체지향 프로그램 종속 그래프의 제안

제안하는 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG : Improved Dynamic Object-oriented Program Dependence Graph)는 기존의 객체지향 프로그램 종속 그래프 개념을 기반으로 하고 3 장에서 제안한 동적 시스템 종속 그래프 기법을 사용하여 객체지향 프로그램을 표현할 수 있도록 하는 표현법이다. IDOPDG는 호출환경을 위한 프로시저 종속 그래프와 각 클래스를 위한 클래스 종속 그래프로 구성되며 클래스와 메인 프로그램간의 상호작용, 다형성 등을 표현한다.

객체지향 프로그램에서는 객체는 new 연산자, 선언 등을 통해서 생성된다. 예를 들면, 만일 o1이 클래스 c1의 객체라면, 객체 o1의 생성 지점에서 클래스 c1의 생성자(constructor) 메소드로의 호출이 존재한다. 이러한 호출은 객체 o1의 생성 정점에서 기저 클래스 c1의 클래스 헤드 정점으로 호출 간선을 연결하여 표현한다. 만일, 클래스 c2의 임의 메소드 m2에서 클래스 c1의 공용 영역에 있는 메소드 m1으로 호출 문장이 있다면, 클래스 c2의 메소드 m2의 호출 정점에서 메소드 m1과 멤버 간선으로 연결된 클래스 c1의 헤드 정점으로 호출 간선을 연결하여 표현한다. 이때 호출 정점과 클래스 c1과 멤버 간선으로 연결된 메소드인 m1 사이에 암시적 호출 문맥이 표현된다.

4.1 IDOPDG 표기법

4.1.1 흐름 그래프

프로그램 P의 흐름 그래프(flow-graph)는 (V, A, En, Ex)로 구성된다. V는 assign, read 그리고 write 문과 같은 일반적인 명령문과 조건문과 반복문의 제어조건을 나타내는 서술식(predicate statement) 이다. A는 정점들 사이에서의 방향성 간선(directed edges)을 나타낸다. En과 Ex는 V에서의 진입노드(entry node)와 출구노드(exit node)를 각각 나타낸다. 클래스를 CL 그리고, 호출이나 메소드 정점을 CM이라고 할 때, 다음은 클래스(CLass), 호출(call), 메소드(method), 할당문(assign statement), 조건문 그리고 반복문에 대한 흐름 그래프에 대한 프로그래밍 언어 표기를 나타낸다. 본 논문에서 기술한 프로그래밍 언어의 구조는 프로그래밍 언어 ML에서 유사 구조로 채택된 let-in구조를 사용한다[49].

let <declarations> in <expression>

<declarations> 는 <expression>에서 사용되어지는 변수들과 그 값으로 구성되어 있다. <expression> 실행 후 그 결과는 let으로 return된다.

Flow(CL; S) =

let Flow(S) = (V, A, En, Ex),

$V' = CL \cup V$,

$A' = A \cup \{(CL, En)\}$

in (V', A', CL, Ex)

Flow(CM; S) =

let Flow(S) = (V, A, En, Ex),

$V' = CM \cup V$,

$A' = A \cup \{(CM, En)\}$

in (V', A', CM, En)

$Flow(S_1; S_2)$ =

let $Flow(S_1) = (V_1, A_1, En_1, Ex_1)$,

let $Flow(S_2) = (V_2, A_2, En_2, Ex_2)$,

$V' = V_1 \cup V_2$,

$A' = A_1 \cup A_2 \cup \{(Ex_1, En_2)\}$

in (V', A', En_1, Ex_2)

Flow(if P S_1 else S_2) =

let Flow(S_1) = (V_1, A_1, En_1, Ex_1),

let Flow(S_2) = (V_2, A_2, En_2, Ex_2),

$V' = V_1 \cup V_2 \cup P$,

$A' = A_1 \cup A_2 \cup \{(P, En_1), (P, En_2)\}$

in (V', A', P, Ex)

Flow(while P S) =

let Flow(S) = (V, A, En, Ex),

$V' = V \cup P$,

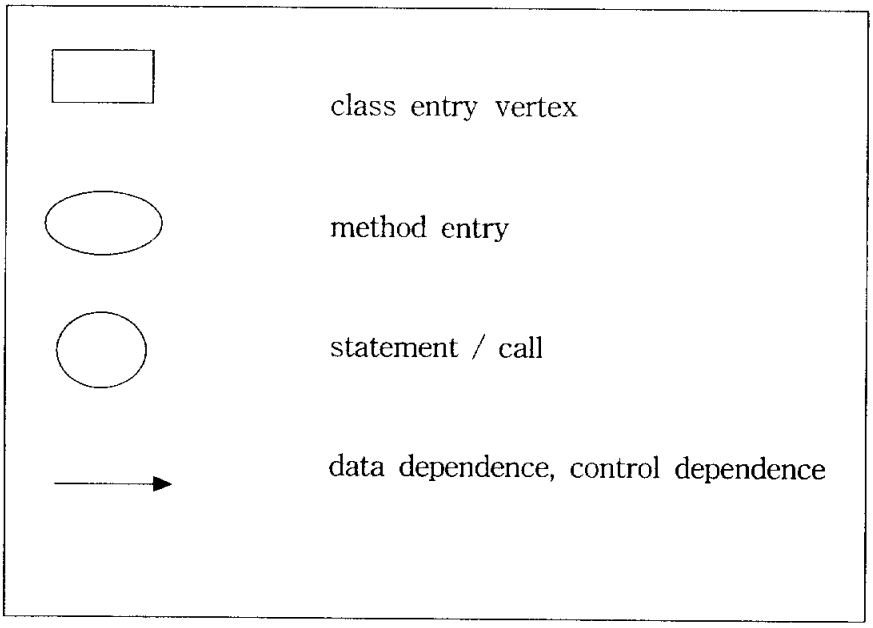
$A' = A \cup \{(P, En), (Ex, P)\}$

in (V', A', P, P)

4.1.2 흐름 그래프의 기호

클래스 진입 정점은 사각형으로 표기하며, 메소드 진입 정점은 타원형으로 프로그램의 각 명령문과 호출문 정점은 원으로 표기하였다. 클래스 멤버 간선과 자료 종속 그리고, 제어 종속 간선은 실선으로 표기하였다.

IDOPDG의 표현기호에 대한 정의가 (표 4-1)에 나타나 있다.



(표 4-1) IDOPDG의 표현기호에 대한 정의

(Table 4-1) Definition about representation symbol of IDOPDG

4.2 종속 그래프

4.2.1 클래스 종속 그래프

본 논문에서 제안하는 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG) 기법은 객체지향 프로그램의 이해와 분석을 쉽게 하기 위하여 우선 클래스를 식별하고, 각 클래스는 클래스 종속 그래프(CLDG)로 표현할 수 있다.

(그림 4-1)은 (프로그램 4-1)의 C++ class Elevator 와 class Elevator로부터 상속된 C++ class AlarmElevator에 대한 클래스 종속 그래프를 보여준다.

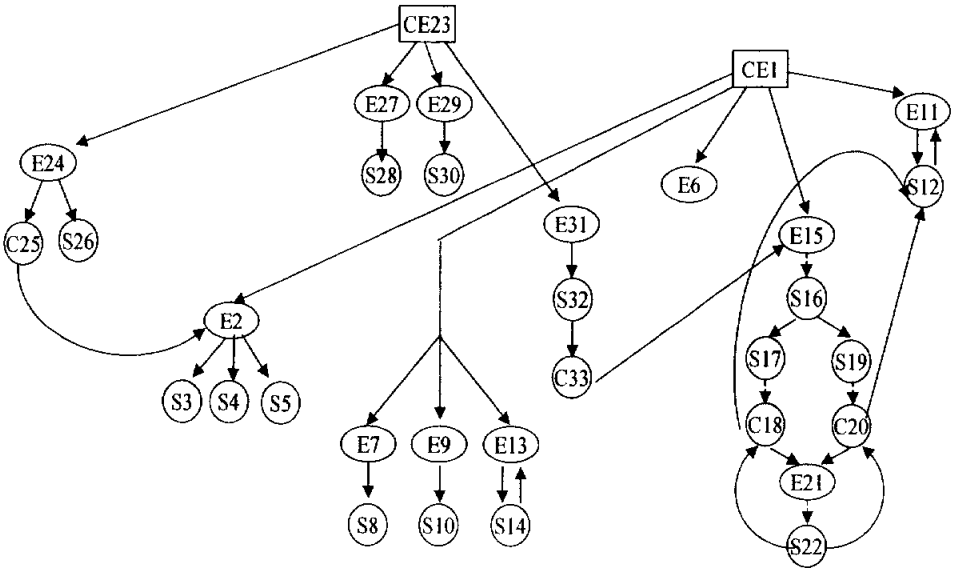
```
CE1 :   class Elevator {
        public:
E2:     Elevator(int 1_top_floor)
S3:      { current_floor = 1;
S4:        current_direction = UP;
S5:        top_floor = 1_top_floor; }
E6:     virtual ~Elevator() {}
E7:     void up()
S8:      { current_direction = UP; }
E9:     void down()
S10:      { current_direction = 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;
    } ;
```

(프로그램 4-1) C++ Elevator class 와 class Elevator로부터 상속된 C++ class AlarmElevator

(Program 4-1) the C++ Elevator class and the C++ class AlarmElevator inherited from the class Elevator



(그림 4-1) class Elevator 와 class Elevator로부터 상속된 class AlarmElevator를 위한 CLDG

(Figure 4-1) CLDG for the class Elevator and the class AlarmElevator inherited from class Elevator

4.2.2 다형성 종속 그래프

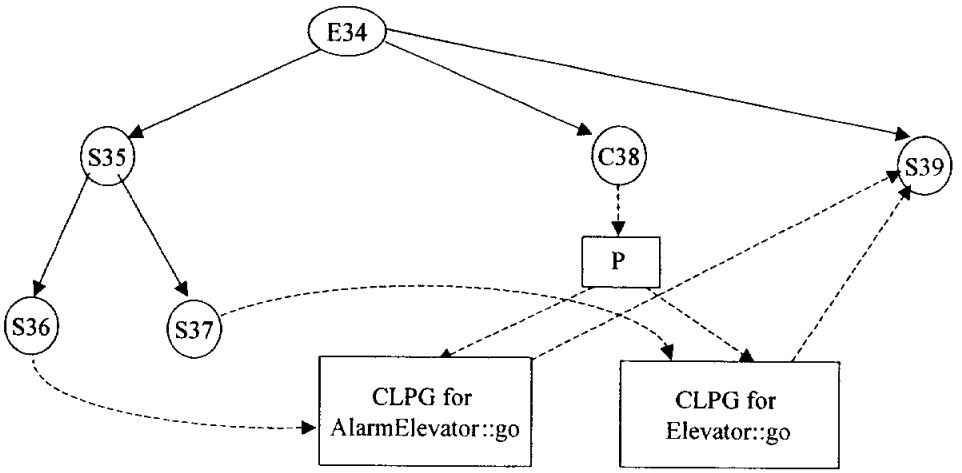
다형성 호출(polymorphic call)은 class에 대해 간접적 참조가 만들어질 때, 그리고 참조되어지고 있는 오브젝트 타입이 컴파일시간에 알려지지 않을 때 발생한다. 그런 경우에, 컴파일러는 pointer 참조타입을 동적으로 해결하기 위한 코드를 포함한다. 다형성 호출의 가능한 목적지를 표현하기 위해서 다형성 선택(polymorphic choice) 정점을 사용한다. 다형성 호출의 call 정점은 다형성 선택 정점에 대해 간선을 가진다. 다형성 선택 정점은 다형성 호출의 모든 가능한 목적지에 대해 호출을 표현하는 sub-graph에 대해 즉각적인 간선을 가진다.

(프로그램 4-2)에는 class Elevator와 class AlarmElevator를 호출하는 main()이 나타나 있고, (그림 4-2)에는 다형성 호출에 대해 나타나 있다. C38에서 다형성 선택 정점에 이르는 call 간선이 있고, 다형성 선택 정점에서 가능한 호출 목적지 각각에 대한 호출을 표현하는 call 정점에 이르는 call 간선이 있다. 이러한 call 정점은 호출되어 질 수 있는 methods를 위한 entry 정점과 관계되어 덧붙여진다. 이 경우에, 가능한 methods는 좌측에 보여지는 AlarmElevator :: go()이며 우측에 보여지는 Elevator :: go()이다.

```
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 4-2) C++ class Elevator와 class AlarmElevator를 호출하는 main()

(Program 4-2) main() calling the C++ class Elevator and class AlarmElevator



(그림 4-2) polymorphic 호출

(Figure 4-2) polymorphic call

4.2.3 개선된 동적 객체지향 프로그램 종속 그래프

개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG)는 제어 종속 정보를 각 문장 정점에서 제어 종속 간선으로 표현하고 각 변수들에 대한 자료 종속 정보를 자료 종속 간선으로 표현한다는 점에서는 기존의 절차적 언어의 단일 프로시저를 표현하는 PDG와 유사하다. 그러나 기존의 객체지향 프로그램 종속 기법은 클래스 정의에 대한 객체 생성을 표현하기 위해 객체 변수에 대하여 객체 정점에 멤버 변수 간선으로 연결하고 클래스 헤드 정점과의 객체 생성 호출 간선, 다형성 호출을 표현하기 위한 다형성 호출 간선, 클래스 메소드 호출을 표현하기 위해 호출 정점과 해당 클래스의 클래스 헤드 정점간을 연결하는 메소드 호출 간선 등이 추가된다. 그러나, 본 논문에서 제시하는 IDOPDG 기법은 다형성 호출을 위한 다형성 호출 간선만 추가된다. 그러므로 기존의 객체지향 프로그램 종속 기법 보다 복잡성이 줄어든다.

본 논문에서 제안하는 IDOPDG는 프로그램의 정적 정보 와 동적 정보를 이용하여 기존의 OPDG 와 DOPDG 보다 간결하게 그래프를 표현하는 기법이다.

(그림 4-3)은 (프로그램 4-1)과 (프로그램 4-2)에 대한 IDOPDG이다.

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 간선을 구한 후 이 간선이 그래프에서 path로 존재하지 않으면 그래프에 추가한다.

(3) 실행이력에 있는 기준 노드로부터 제어 종속 간선을 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.

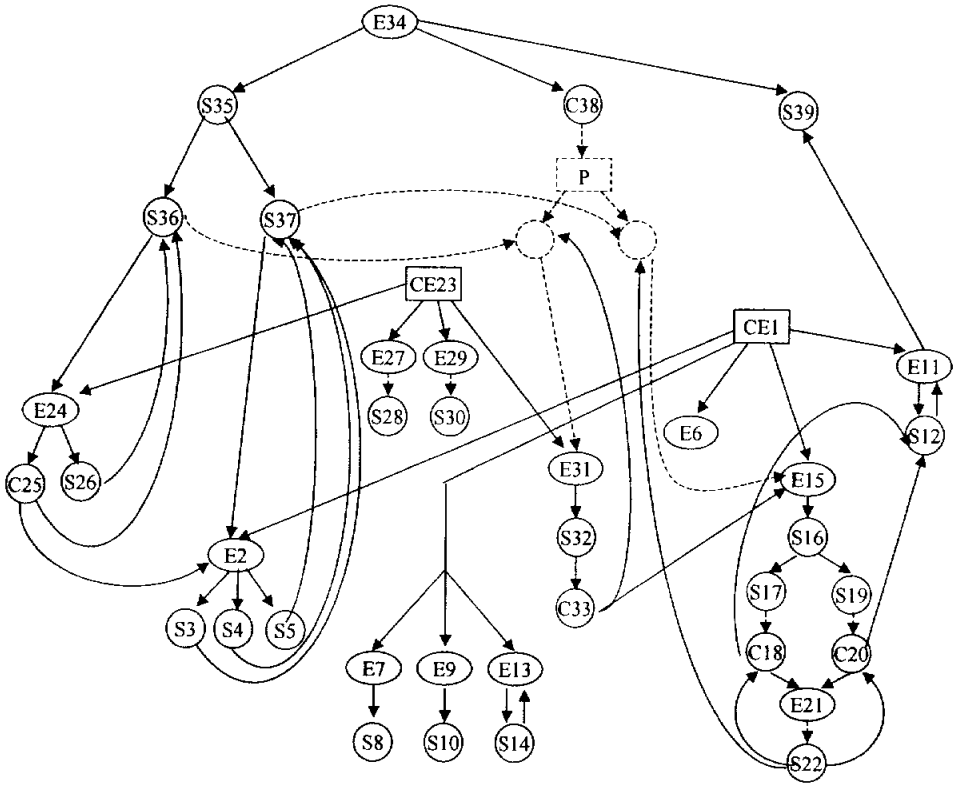
(1) $B^{d}_{C\langle a,b\rangle}\langle m,v\rangle$

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

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

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

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



(그림 4-3) main()에 대한 IDOPDG

(Figure 4-3) IDOPDG for main()

제5장. 동적 객체지향 프로그램 슬라이싱

5.1 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.

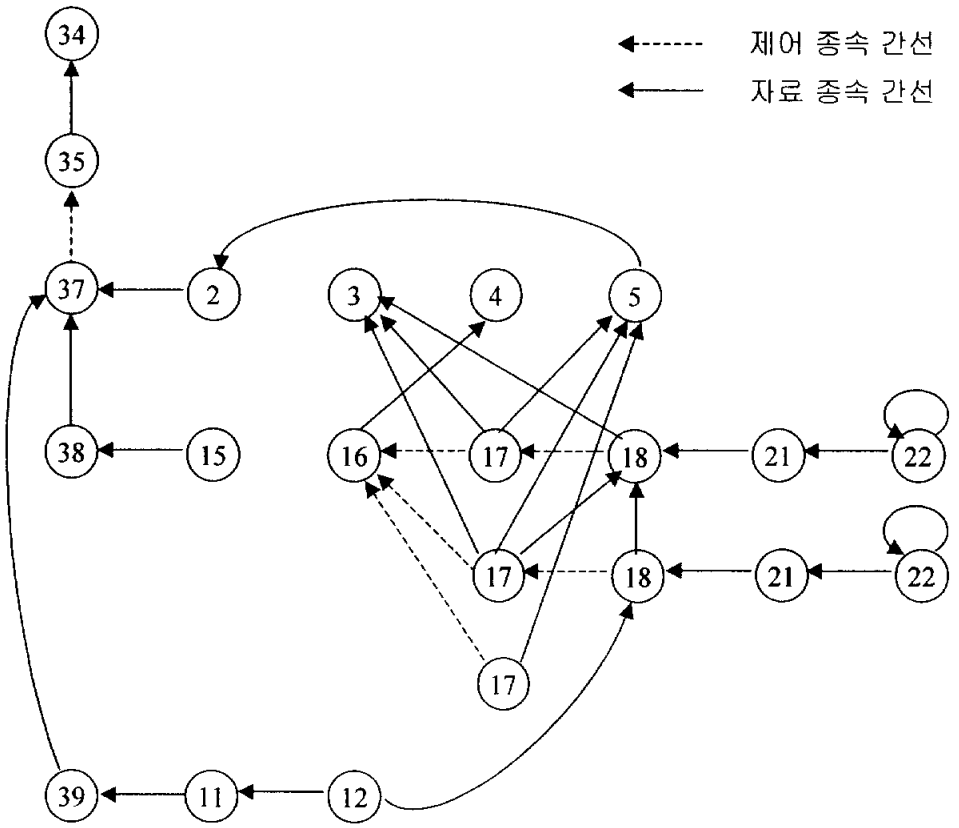
객체지향 프로그램의 동적 객체지향 프로그램 종속 그래프의 구성은 해당 원시 프로그램의 제어 흐름과 자료 흐름에 대한 동적 분석에 기초하고 있다. 그리고, 이것은 procedural 프로그램에 대한 동적 종속 그래프의 구축과 유사하다. 하나의 procedural 프로그램에서의 call 명령문은 function 기능을 가진 명령문이나 procedure를 호출하는 명령문이다. 그러나, 객체지향 프로그램에서는 classes, instances, objects 그리고, 동적 바인딩을 고려해야한다. (그림 5-1)은 (프로그램 4-1)과 (프로그램 4-2)에 대한 동적 객체지향 프로그램 종속 그래프이다.

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 정점

들의 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) $\subset$ V이다.

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, 객체 지향 프로그램의 DOPDG에 대한 동적 슬라이스는 DOPDG의 정점들의 subset이다.



(그림 5-1) (프로그램 4-1)과 (프로그램 4-2)에서 C = (s39, which_floor)에 대한 DOPDG

(Figure 5-1) DOPDG on C = (s39, which_floor) of (Program 4-1) and (Program 4-2)

5.2 개선된 동적 객체지향 프로그램 슬라이싱

본 논문에서 제시한 개선된 동적 객체지향 프로그램 종속 그래프를 이용한 동적 객체지향 프로그램 슬라이스를 구하는 절차는 크게 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

5.2.1 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 노드번호, 노드type, DEF, REF, 소속모듈번호 그리고, 상위제어노드번호로 구성된다.

(1) 노드의 종류

프로그램을 구성하는 노드들은 11가지 종류로 구분되며 (표 5-1)에 나타나 있다.

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

(표 5-1) 프로그램을 구성하는 노드 테이블

(Table 5-1) Node table that constitutes programs

(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

DEF와 REF를 구성하는 식이 (프로그램 5-1)에 나타나 있다.

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 5-1) DEF와 REF를 구성하는 식

(Program 5-1) Expressions that constitute DEFs and REFs

5.2.2 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 구

성된다. 노드실행순번은 실행이력순번을 나타내고, 노드번호는 노드관련테이블에서의 노드번호와 같다.

5.2.3 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 객체지향 프로그램 종속 그래프(IDOPDG)를 작성하는 단계이다.

프로그램을 구성하는 명령문의 구조는 (프로그램 5-2)와 같이 표현할 수 있다. 그리고, 본 논문에서 제시한 효율적으로 동적 객체지향 프로그램 슬라이스(*Mark*)를 산출하는 알고리즘은 (알고리즘 5-1)에 나타나 있다.

5.2.4 프로그램 슬라이스 작성 단계

기준 변수에 대한 동적 슬라이스 추출을 위해 5.2.3에서 작성한 IDOPDG에서 기준 노드를 중심으로 역 운행(back tracking)하여 해당되는 슬라이스를 추출하는 단계이다. 슬라이스된 프로그램은 주어진 입력에 대해 실행 가능한 완벽한 프로그램이다.

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 5-2) 프로그램을 구성하는 명령문의 구조

(Program 5-2) Structure of statements that constitutes programs

```
IDOPDG(<prevhist | Mark>) =
        SetVar' = Ins(Criterion, SetVar)
        DependCheck(Criterion, 1, 1)
        while k = 1, n
            DependCheck(IfCriterion, lastnum, 2)
        end while
        while k = 1, m
            if SubSist(k, CheckObject)
            then DependCheck(RepeatCriterion, 1, 2)
            end if
        end while
    in (Criterion, IfCriterion, RepeatCriterion, CheckObject,
        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'*) = $\cup_{x \in D}$ *Var*(*x*) $\cup$ $\cup_{x \in D}$ *SetVar*(*x*)

in (*Var*, *SetVar'*)

Del(*Var*, *SetVar*) =

Del(*Var*, *SetVar'*) = $\cup_{x \in D}$ *SetVar*(*x*) - $\cup_{x \in D}$ *Var*(*x*)

in (*Var*, *SetVar'*)

SubSist(*Var*(*x*), *SetVar*) =

let sist = 0

if ($\cup_{x \in D}$ *Var*(*x*) $\cap$ $\cup_{x \in D}$ *SetVar*(*x*))

then sist = 1

end if

in (*Var*, *SetVar*, *sist'*)

(알고리즘 5-1) 동적 객체지향 프로그램 슬라이스를 산출하는 알고리즘

(Algorithm 5-1) Algorithm that computes dynamic object-oriented program slices

5.3 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.

5.3.1 기존의 객체지향 프로그램 슬라이싱 기법

(1) 특징

기존의 프로그램 종속 그래프 혹은 시스템 종속 그래프 표현에 클래스 종속 그래프, 클래스 계층구조 그래프, 클래스 호출 그래프, 클래스 제어 흐름 그래프, 클래스간 호출 그래프, 클래스간 제어 흐름 그래프, 클래스간 종속 그래프 등의 그래프를 추가하여 객체지향 프로그램의 특징을 표현

(2) 장점

클래스간의 상호작용, 다형성, 상속성 등을 잘 표현할 수 있다.

(3) 단점

메소드의 진입 정점에 매개변수의 표현과 파생 클래스의 객체 변수의 표현 그리고 메소드의 진입 정점의 매개변수의 표현을 위해 formal_in 정점, formal_out 정점, actual_in 정점, actual_out 정점들을

표현함으로써 그래프의 복잡도가 증가한다.

5.3.2 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.

5.3.3 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.

제6장. 적용 사례 및 평가

6.1 적용 사례 1

(프로그램 4-1)과 (프로그램 4-2)의 예제 프로그램에서 argv[1] = 3 이고, 슬라이싱 기준이 (H, 39_{22}, which_floor)일 때, 동적 객체지향 슬라이스를 구하기 위해 본 논문에서 제안한 동적 객체지향 프로그램 슬라이싱 알고리즘에 적용시킨다.

6.1.1 프로그램 노드 분석 단계

(프로그램 4-1) 과 (프로그램 4-2)의 예제 프로그램을 구성하고 있는 노드들의 자료 분석 테이블이 (표 6-1)에 나타나 있다.

노드 번호	노드 type	DEF	REF	소속모듈번호	상위제어노드
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	P	add	a, b	18	

노드 번호	노드 type	DEF	REF	소속모듈번호	상위제어노드
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	M	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	

(표 6-1) 예제 프로그램의 노드 관련 자료

(Table 6-1) The data of related nodes for the Example Program

6.1.2 프로그램 실행이력 분석 단계

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

6.1.3 동적 객체지향 프로그램 종속 그래프 작성 단계

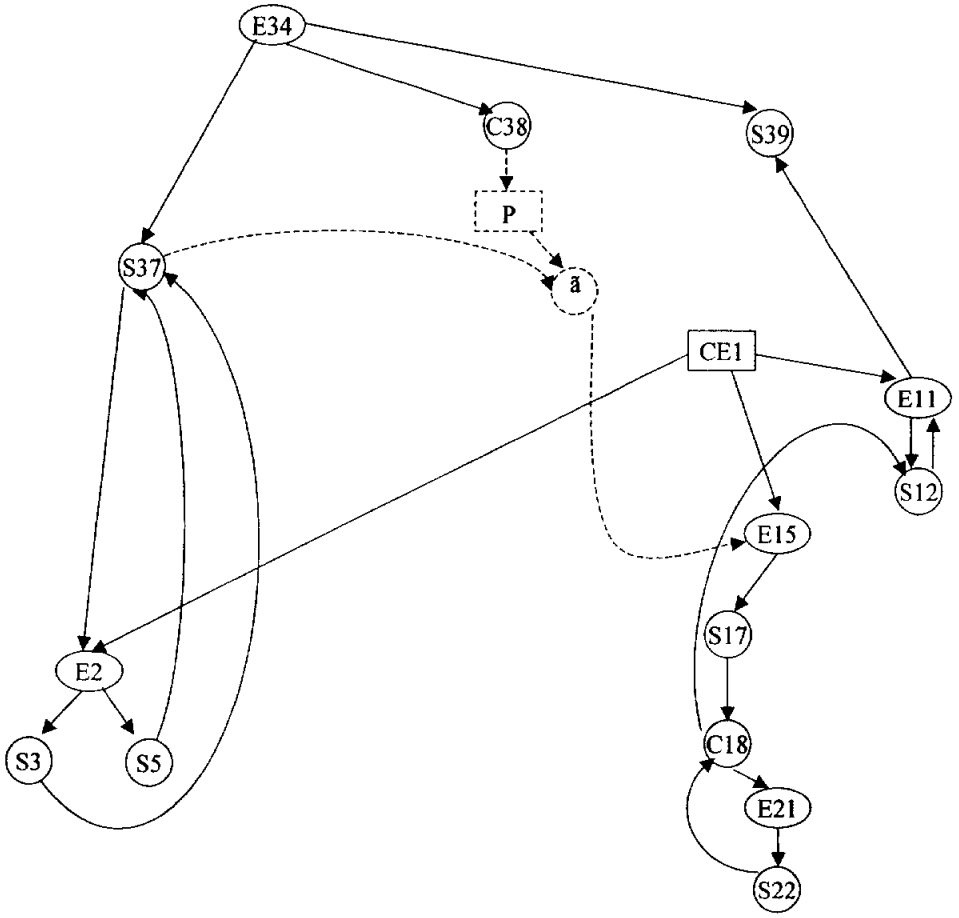
(H, 39_{22}, which_floor)를 슬라이싱 기준으로 했을 때 즉, 기준 노드 39번에 있는 슬라이싱 기준 변수 which_floor를 슬라이싱 기준 변수로 했을 때 IDOPDG를 4.2.3에 있는 IDOPDG 만드는 절차에 의해 생성한다.

(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→γ)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 4-1) 과 (프로그램 4-2)의 예제 프로그램에 대한 IDOPDG가 (그림 6-1)에 나타나 있다.

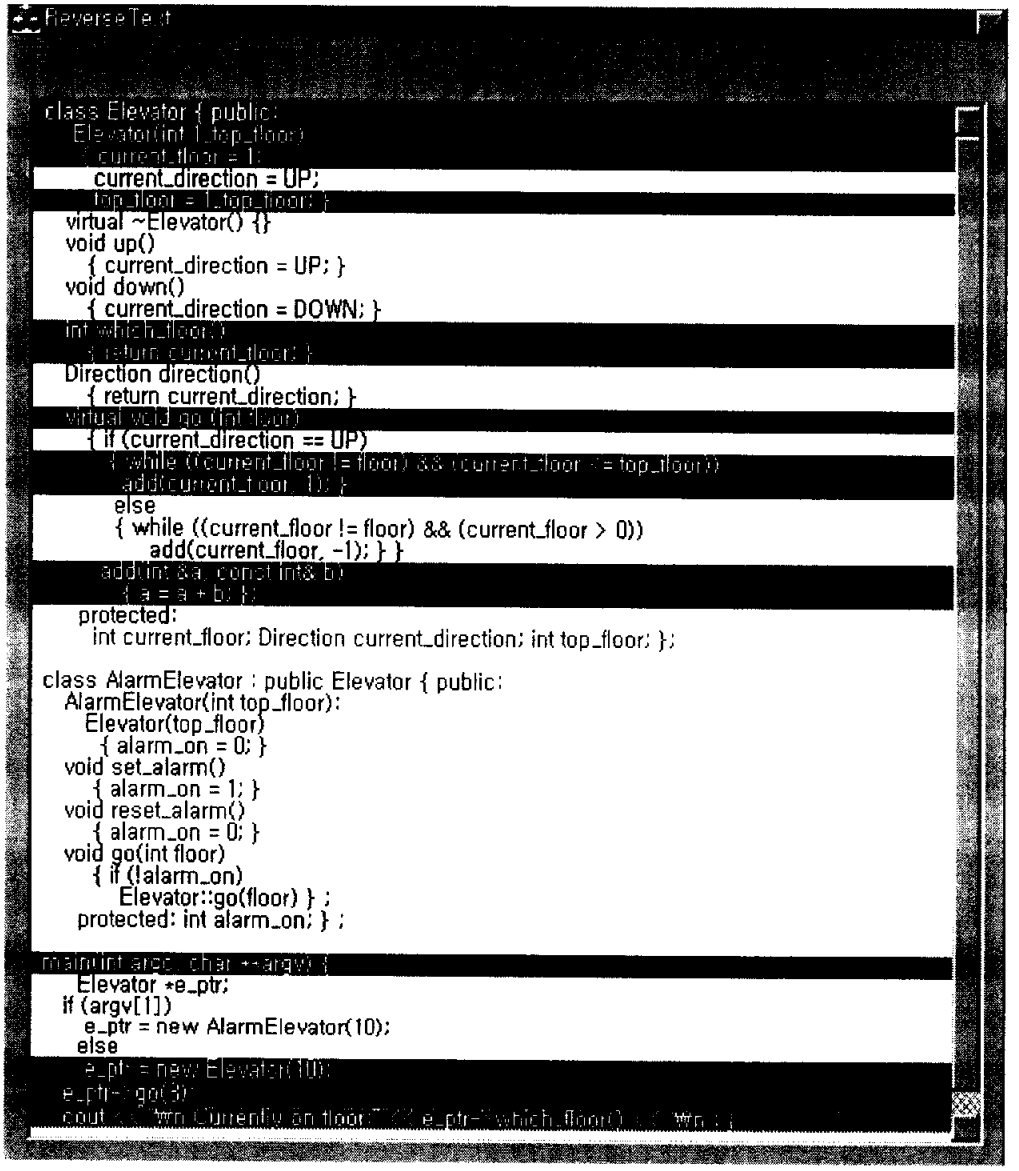


(그림 6-1) 기준노드 39에 대한 개선된 동적 객체지향 프로그램 종속 그래프

(Figure 6-1) IDOPDG for criterion node 39

6.1.4 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 6-1)에 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 6-2)에 나타나 있다.



(그림 6-2) 슬라이싱 결과

(Figure 6-2) Slicing results

6.2 적용 사례 2

(프로그램 4-1) 과 (프로그램 4-2)의 예제 프로그램에서 argv[1] = 3 일 때 실행이력의 16번째에 있는 노드번호 18의 current_floor 즉, (H, 18_{16}, current_floor)를 슬라이싱 기준으로 하는 동적 객체지향 프로그램 슬라이싱을 위해 본 논문에서 제안한 동적 객체지향 프로그램 슬라이싱 알고리즘에 적용시킨다.

슬라이싱을 위한 네 단계 중 앞의 두 단계 즉, 프로그램 노드 분석 단계와 프로그램 실행 이력 분석 단계는 동일하다. 그러므로 이 두 단계의 실행 결과를 바탕으로 세 번째 단계인 동적 객체지향 프로그램 종속 그래프 작성 단계와 네 번째 단계인 프로그램 슬라이스 작성 단계를 통해 슬라이스를 추출한다.

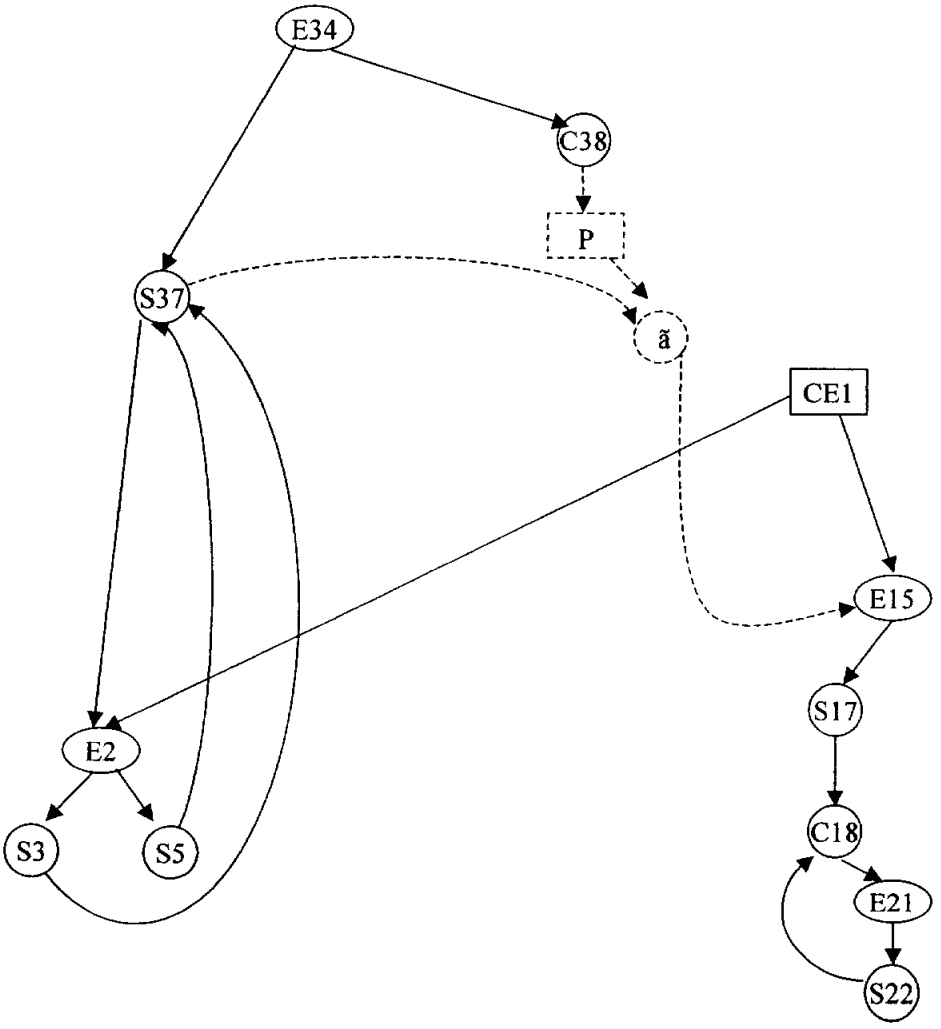
6.2.1 동적 객체지향 프로그램 종속 그래프 작성 단계

(H, 18_{16}, current_floor)를 슬라이싱 기준으로 했을 때 (프로그램 4-1) 과 (프로그램 4-2)의 예제 프로그램에 대한 IDOPDG가 (그림 6-3)에 나타나 있다.

6.2.2 프로그램 슬라이스 작성 단계

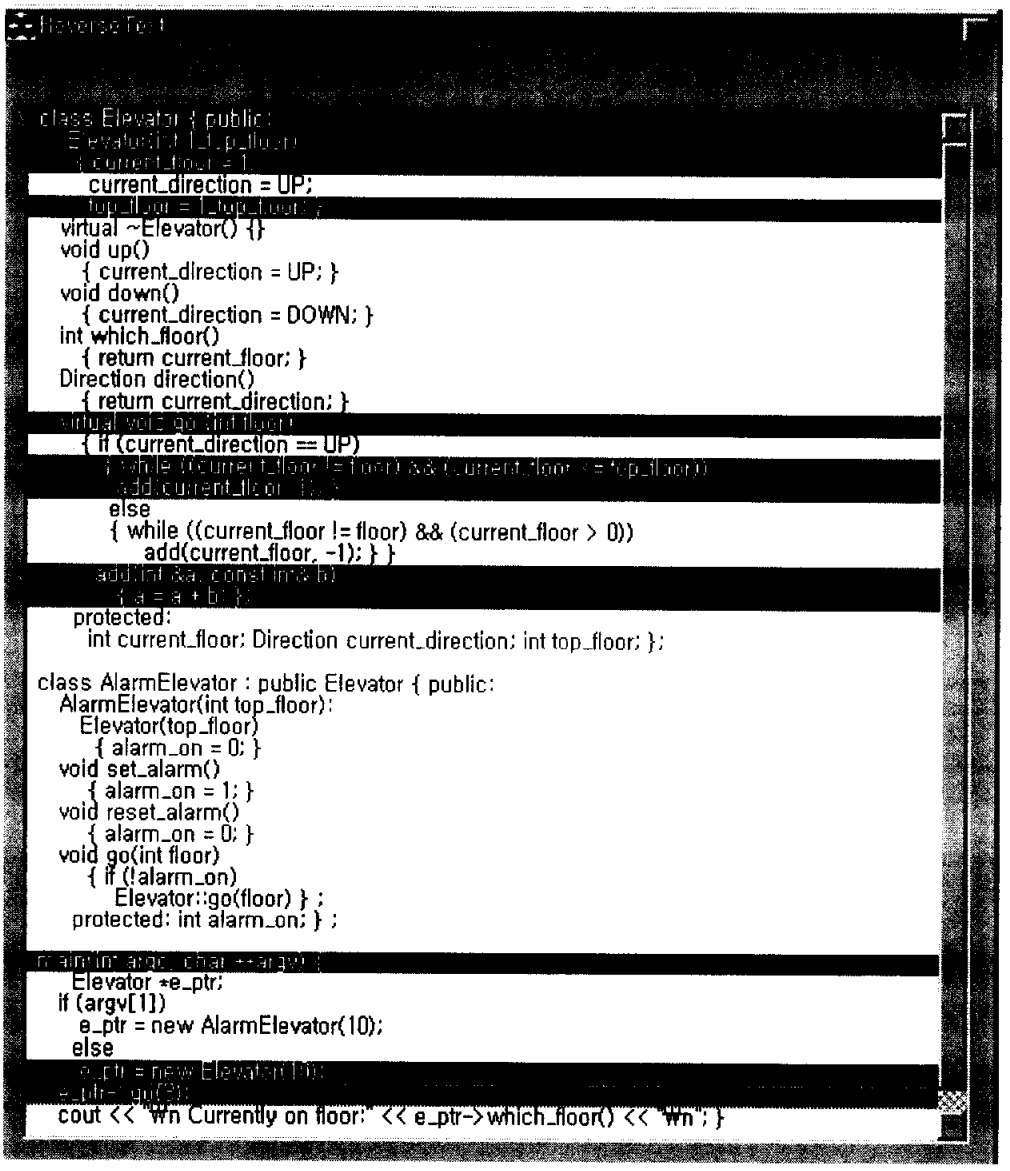
(H, 18_{16}, current_floor)를 슬라이싱 기준으로 했을 때 즉, 기준 노드

18번에 있는 슬라이싱 기준 변수 current_floor에 대한 동적 슬라이스를 구하기 위해 (그림 6-3)에 있는 IDOPDG를 역 운행한 결과 산출된 슬라이스 노드는 {1, 2, 3, 5, 15, 17, 18, 21, 22, 34, 37, 38}이 된다. 이것은 구현 프로그램에 적용시킨 결과에서도 슬라이스 결과가 같음을 확인할 수 있다. 구현 프로그램을 통해 적용시킨 슬라이싱 결과가 (그림 6-4)에 나타나 있다.



(그림 6-3) (H, 18_{16}, current_floor)에 대한 동적 객체지향 프로그램 종속 그래프

(Figure 6-3) IDOPDG for (H, 18_{16}, current_floor)



```
class Elevator { public:
    Elevator(int t_p_floor)
      { current_floor = 1;
        current_direction = UP;
        top_floor = t_top_floor; }
    virtual ~Elevator() {}
    void up()
      { current_direction = UP; }
    void down()
      { current_direction = DOWN; }
    int which_floor()
      { return current_floor; }
    Direction direction()
      { return current_direction; }
    virtual void go(int floor)
      { if (current_direction == UP)
          { while ((current_floor != floor) && (current_floor <= top_floor))
              add(current_floor, 1); }
        else
          { while ((current_floor != floor) && (current_floor > 0))
              add(current_floor, -1); } }
      add(int &a, const int& b)
        { a = a + b; }
    protected:
      int current_floor; Direction current_direction; int top_floor; };

class AlarmElevator : public Elevator { public:
    AlarmElevator(int top_floor):
      Elevator(top_floor)
        { alarm_on = 0; }
    void set_alarm()
      { alarm_on = 1; }
    void reset_alarm()
      { alarm_on = 0; }
    void go(int floor)
      { if (!alarm_on)
          Elevator::go(floor) } ;
    protected: int alarm_on; } ;

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(5);
    cout << "\n Currently on floor:" << e_ptr->which_floor() << "\n"; }
```

(그림 6-4) 슬라이스싱 결과

(Figure 6-4) Slicing results

6.3 효율성 분석

기존의 객체지향 프로그램 종속 그래프와 본 논문에서 제시한 동적 객체지향 프로그램 종속 그래프의 복잡도를 측정하면 다음과 같다.

6.3.1 객체지향 프로그램 종속 그래프의 복잡도

(프로그램 4-1) 와 (프로그램 4-2)의 예제 프로그램에 대해 기존의 객체지향 프로그램 종속 그래프 기법을 이용해 슬라이스를 구하면 (프로그램 6-1)과 같다. 그리고, 기존의 객체지향 프로그램 종속 그래프의 복잡도 φ 가 (표 6-2)에 나타나 있다. 그리고, 복잡도를 구성하는 변수들의 종류가 (표 6-3)에 나타나 있다.

```
class Elevator {
public:
Elevator(int l_top_floor)
 { current_floor = 1;
   current_direction = UP;
   top_floor = l_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); }
   else
    { while ((current_floor != floor) &&
             (current_floor > 0))
          add(current_floor, -1); }
 }

private:
 add(int &a, const int& b)
  { a = a + b; };
```

```
protected:
 int current_floor;
 Direction current_direction;
 int top_floor;
};

class AlarmElevator : public Elevator {
public:
 AlarmElevator(int top_floor):
   Elevator(top_floor)
    { alarm_on = 0; }
 void go(int floor)
  { if (!alarm_on)
      Elevator::go(floor)
  } ;

protected:
  int alarm_on;
} ;

 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 6-1) OPDG를 이용한 슬라이스된 프로그램

(Program 6-1) A sliced program using OPDG

종류	기존 객체지향 프로그램 종속그래프 복잡도
procedure 종속	pv + pc * (1 + pcp * 2) + p + 2 * pp + sc * (1 + scv * 2)
class 종속	s + (sv + sc) + m * 2 * (mp + scp)

(표 6-2) 기존의 객체지향 프로그램 종속 그래프의 복잡도

(Table 6-2) Complexities of a traditional object-oriented program dependence graph

변수 명	변수의 내용
p	procedure
pp	procedure 의 parameter
pv	procedure 내에서의 일반 vertex
pc	procedure 내에서의 call
pcp	procedure 내에서의 call 문의 parameter
sc	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

(표 6-3) 복잡도를 구성하는 변수 테이블

(Table 6-3) Table of variables that constitute complexities

Σ복잡도φ(기존OPDG)

= 복잡도φ(procedure 종속) + 복잡도φ(class 종속)

= pv + pc * (1 + pcp * 2) + p + 2 * pp + sc * (1 + scv * 2)

+ s + sv + sc + m * 2 * (mp + scp)

6.3.2 동적 객체지향 프로그램 종속 그래프의 복잡도

(프로그램 4-1) 과 (프로그램 4-2)의 예제 프로그램에 대해 기존의 동적 객체지향 프로그램 종속 그래프 기법을 이용해 슬라이스를 구하면 (프로그램 6-2)와 같다. 기존의 동적 객체지향 프로그램 종속 그래프에 대한 복잡도φ가 (표 6-4)에 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 6-2) DOPDG를 이용한 슬라이스된 프로그램

(Program 6-2) A sliced program using DOPDG

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

(표 6-4) 기존의 동적 객체지향 프로그램 종속 그래프의 복잡도

(Table 6-4) Complexities of a traditional dynamic object-oriented program dependence graph

Σ복잡도 φ(DOPDG)

= 복잡도 φ(procedure 종속) + 복잡도 φ(class 종속)

= p + pc + 3 * pv + s + sc + m + 3 * sv

6.3.3 개선된 동적 객체지향 프로그램 종속 그래프의 복잡도

(프로그램 4-1) 와 (프로그램 4-2)의 예제 프로그램에 대해 본 논문에서 제시한 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG)의 복잡도 φ가 (표 6-5)에 나타나 있다.

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

(표 6-5) 개선된 동적 객체지향 프로그램 종속 그래프의 복잡도

(Table 6-5) Complexities of a Improved Dynamic Object-oriented Program Dependence Graph

Σ복잡도 φ(IDOPDG)

= 복잡도 φ(procedure 종속) + 복잡도 φ(class 종속)

= pv + p + 2 * (pc + sc) + s + sv + sc + m

6.3.4 효율성 비교

(프로그램 4-1)과 (프로그램 4-2)의 예제 프로그램에 대하여 명령문 39번의 which_floor()를 기준으로 했을 때, 기존의 OPDG와 DOPDG 그리고, 본 논문에서 제시한 IDOPDG의 슬라이스의 크기를 비교하는 테이블이 (표 6-6)에 나타나 있고, 복잡도를 비교하는 테이블이 (표 6-7)에 나타나 있다.

(1) 슬라이스의 크기

	슬라이스의 크기
객체지향 프로그램 종속 그래프 (OPDG)	28
동적 객체지향 프로그램 종속 그래프 (DOPDG)	18
개선된 동적 객체지향 프로그램 종속 그래프 (IDOPDG)	15

(표 6-6) 슬라이스의 크기를 비교하는 테이블

(Table 6-6) Table that compares sizes of slices

(2) 복잡도 비교

	최대 복잡도	실제 복잡도
객체지향 프로그램 종속 그래프 (OPDG)	271	89
동적 객체지향 프로그램 종속 그래프 (DOPDG)	69	22
개선된 동적 객체지향 프로그램 종속 그래프 (IDOPDG)	42	17

(표 6-7) 복잡도를 비교하는 테이블

(Table 6-7) Table that compares complexities

기존의 객체지향 프로그램 종속 그래프에서 최대 복잡도와 실제 복잡도가 차이가 나는 이유는 호출에서 매개변수의 전부 변경되지 않을 수도 있기 때문이다. 위 표에서 보는 바와 같이 동적 객체지향 프로그램 종속 그래프의 복잡도는 기존의 객체지향 프로그램 종속 그래프 보다 훨씬 줄어듦을 알 수 있다. 그 이유는 실 매개변수 전달을 위해 필요한 실 매개변수와 형식 매개 변수들의 표현을 위해 formal-in 정점, formal-out 정점과 actual-in 정점, actual-out 정점이 제외되기 때문이다. 그리고, class 계층 그래프 또한 동적 객체지향 그래프에서는 필요가 없다. 그리고, DOPDG에서는 while 문에 대한 반복 노드의 수 때문에 최대 복잡도와

실제 복잡도가 차이가 나며, 반복 노드의 수 때문에 IDOPDG에 비해 복잡도가 증가한다. 그리고, 명령문 39번의 which_floor()를 기준으로 했을 때의 본 논문에서 제안한 IDOPDG의 복잡도는 17이 되며, 명령문 17번의 current_floor를 기준 변수로 했을 때 실제 동적 객체지향 종속 그래프의 복잡도는 14가 된다. 이 이유는 동적 슬라이스에 포함되지 않는 정점들이 제외되어 그래프의 복잡도가 줄어들기 때문이다.

제7장. 결론

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 동적 슬라이싱 개념을 객체지향 프로그램 슬라이싱에 적용하기 위해 먼저 동적 시스템 슬라이싱 기법을 제안하였다. 그리고, 동적 시스템 종속 그래프를 이용하여 동적 슬라이스를 산출하였다. 그리고, 기존의 시스템 종속 그래프(SDG)와 본 논문에서 제시한 동적 시스템 종속 그래프(DSDG)를 (프로그램 3-1)의 예제 프로그램에 적용해본 결과 기존의 SDG의 복잡도가 47이었고, DSDG의 복잡도는 14였다. 그리고, DSDG의 기법을 바탕으로 본 논문에서는 개선된 동적 객체지향 프로그램 종속 그래프(IDOPDG)를 이용한 동적 슬라이싱 기법을 제안하였다. 기존의 OPDG 기법과 DOPDG 기법 그리고, 본 논문에서 제시한 IDOPDG 기법을 사용하여 (프로그램 4-1)과 (프로그램 4-2)의 예제 프로그램을 본 논문에서 제시한 복잡도 측정 공식에 적용한 결과 IDOPDG의 복잡도가 42이고, 기존의 OPDG의 복잡도가 271 그리고, 기존의 DOPDG의 복잡도는 69임을 알 수 있었다. 그리고, 이론 수치가 아닌 실제 그래프에 나타난 IDOPDG와 OPDG 그리고, DOPDG의 복잡도 수치는 각각 17과 89 그리고, 22로 나타났다. 그러므로 본 논문에서 제시한 IDOPDG 기법이 기존의 OPDG 기법과 DOPDG 기법

에 비해 복잡성을 줄여주는 효율적인 표현법임을 알 수 있었다. 복잡도가 줄어든 이유는 기존의 OPDG 기법에서는 정점과 간선이 추가적으로 발생하기 때문이다. 구체적으로 살펴보면 클래스에서 기존의 표현 방법에서는 기저 클래스에서의 클래스 정점은 멤버 변수와는 멤버 변수 간선으로 연결되고, 메소드와는 클래스 멤버 간선으로 연결된다. 그리고, 파생 클래스에서는 클래스 정점이 기저 클래스의 헤드와 상속 간선으로 연결되고, 기저 클래스의 메소드와 상속된 간선으로 연결된다. 그러므로, 멤버 변수 간선과 파생 클래스와 기저 클래스간의 상속 간선이 추가적으로 발생한다. 그리고, 메소드와 프로시저는 기존의 기법에서 실 매개변수와 1 : 1 대응되는 형식 매개변수는 서로 제어 종속 간선으로 연결된다. 그러므로 매개변수간의 간선이 추가적으로 발생한다. 그리고, 클래스의 자료 속성과 메소드의 변수를 연결하는 자료 종속 간선도 추가적으로 발생한다. 그리고, 기존의 기법에서 매개변수를 표현할 때 호출이 발생하면 실 매개변수와 형식 매개 변수를 표현하는 정점인 매개변수 정점과 클래스의 객체 생성시 각 객체의 변수를 정의하는 객체변수 정점이 추가적으로 발생한다. 그리고, 객체 변수에 대하여 객체 정점에 멤버 변수 간선으로 연결하고 클래스 헤드 정점과의 객체 생성 호출 간선, 다형성 호출을 표현하기 위한 다형성 호출 간선, 클래스 메소드 호출을 표현하기 위해 호출 정점과 해당 클래스의 클래스 헤드 정점간을 연결하는 메소드 호출 간선 등이 추가된다. 그리고, DOPDG에서는 while 문에 대한 반복 노드가 발생하므로 복잡도가 그만큼 증가한다.

그리고, IDOPDG에서 노드 39번에 있는 which_floor를 슬라이싱 기준 변수로 했을 때 슬라이스의 크기는 15가 되며 노드 번호 18에 있는

current_floor를 기준 변수로 했을 때는 슬라이스의 크기가 12 이었다. 그러나, DOPED에서는 슬라이스의 크기가 각각 18과 15였다.

우리가 실험한 특징들로 인하여 본 논문에서 제시한 DSDG 기법과 IDOPDG 기법이 기존의 SDG 기법이나 OPDG 기법 그리고, 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 연구해야할 것이다.

참 고 문 헌

[1] 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.

[2] M. J. Harrold and G. Rothermel. "A coherent family of analyzable graph representations for object-oriented software.", Technical Report OSU-CISRC-11/96-TR60, The Ohio State University, November 1996.

[3] Frank Tip, T. B. Dinesh, "A slicing-based approach for locating type errors.", ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 10, Number 1, pp.5-55, January 2001.

[4] Mary Jean Harrold and Brian Malloy, "A unified interprocedural program representation for a maintenance environment.", IEEE Transactions on Software Engineering, Vol. 19, No. 6, pp.584-593, June 1993.

[5] Brian A. Malloy, John D. Mcgregor, Anand Krishnaswamy and

Murali Medikonda, "An Extensible Progarm Representation for Object-Oriented Software.", ACM SIGPLAN Notice., Vol. 29, No. 12, pp.38-47, December 1994.

[6] Wamberto Weber Vasconcelos, "A Flexible Framework for Dynamic and Static Slicing of Logic Programs.", Practical Aspects of Declarative Languages (PADL), San Antonio, Texas, USA, pp.259-274, 1999.

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

[8] B. Korel, "Computation of dynamic slices for unstructured programs.", IEEE Transactions on Software Engineering, vol.23, No.1, pp.17-34, 1997.

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

[10] 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.

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

[12] R. Gopal, "Dynamic program slicing based on dependence relations.", Proc. of the Conf. on Software Maintenance, pp.299-308, 1992.

[13] B. Korel, J. Laski, "Dynamic slicing in computer programs.", The Journal of System and Software, vol.13, No.3, pp.187-195, 1990.

[14] Agrawal, H., Demillo, R., and Spafford, E., "Dynamic slicing in the presence of unconstrained pointers.", In Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification(TAV4) (1991), pp.60-73. Also Purdue University technical report SERC-TR-93-P.

[15] Agrawal, H., Demillo, R., and Spafford, E., "Debugging with dynamic slicing and backtracking.", software - practice and Experience 23, 6, pp.589-616, 1993.

[16] J. Lyle, M. Weiser, "Experiments on slicing-based debugging

tools.", Proc. of the 1st Conf. on Empirical Studies of Programming, pp.187-197, 1986.

[17] Tibor Gyimothy, Arpad Beszedes, Istvan Forgacs, "An Efficient Relevant Slicing Method for Debugging.", SIGSOFT FSE, Toulouse, France, pp.303-321, 1999.

[18] Korel B. and S. Yalamanchili, "Forward Derivation of Dynamic Slices.", Proc. Int'l Symp. Software Testing and Analysis, Seattle, pp.66-79, 1994.

[19] Ralf Kollmann, Martin Gogolla, "Capturing Dynamic Program Behaviour with UML Collaboration Diagrams.", Conference on Software Maintenance and Reengineering (CSMR), Lisbon, Portugal, pp.58-67, 2001.

[20] Arpad Beszedes, Tamas Gergely, Zsolt Mihaly Szabo, Janos Csirik, Tibor Gyimothy, "Dynamic Slicing Method for Maintenance of Large C Programs.", Conference on Software Maintenance and Reengineering (CSMR), Lisbon, Portugal, pp.105-113, 2001.

[21] Susan Horwitz, "Identifying the semantic and textual differences

between two versions of a program.", Proc. of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation pp.234-245, 1990.

[22] Susan Horwitz, T. Reps and David Binkley, "Interprocedural Slicing using Dependence Graph.", ACM Tran. on Programming Languages and Systems, vol. 12, no. January 1990.

[23] Kamkar, M., " Interprocedural Dynamic Slicing with Applications to Debugging and Testing." , PhD thesis, Linkoping Univ., 1993.

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

[25] Gagan Agrawal, Liang Guo, "Evaluating explicitly context-sensitive program slicing.", Workshop on Program Analysis For Software Tools and Engineering (PASTE), Snowbird, Utah, USA, pp.6-12, 2001.

[26] Saurabh Sinha, Mary Jean Harrold, Gregg Rothermel, "Interprocedural control dependence.", ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 10,

Number 1, pp.209-254, April 2001.

[27] Raghavan Komondoor, Susan Horwitz, "Tool Demonstration: Finding Duplicated Code Using Program Dependences.", European Symposium on Programming (ESOP), Genova, Italy, pp.383-386, 2001.

[28] Susan Horwitz, Jan Prins, and Thomas Reps., "Integrating noninterfering versions of programs.", ACM Trans. on Programming Langusges and Systems, pp.345-387, July 1989.

[29] Agrawal, H., "On slicing programs with jump statements.", In Proceedings of the ACM SIGPLAN'94 Conference of Programming Language Design and Implementation, Florida, 1994.

[30] Mark Weiser, "Programmers use slices when debugging.", Communications of the ACM, pp.446-452, July 1982.

[31] Mark Weiser, "Program slicing.", IEEE Trans. on Software Engineering, pp.352-357, July 1984.

[32] Lynette I. Millett, Tim Teitelbaum, "Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding,

and Simulation.", International Journal on Software Tools for Technology Transfer (STTT), Volume 2, Number 4, pp.343-349, 2000.

[33] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, Tim Teitelbaum, "Program Slicing of Hardware Description Languages.", Correct Hardware Design and Verification Methods (CHARME), Bad Herrenalp, Germany, pp.298-312, 1999.

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

[35] Gyongyi Szilagyi, Tibor Gyimothy, Jan Maluszynski, "Slicing of Constraint Logic Programs.", Automated and Algorithmic Debugging (AADEBUG), Munich, Germany, 2000.

[36] Mark Weiser, "Programmers use slices when debugging.", Communications of the ACM, pp.446-452, July 1982.

[37] Wamberto Weber Vasconcelos, Marcelo A. T. Aragao, "Slicing knowledge-based systems: techniques and applications.",

Knowledge Based Systems, Volume 13, Number 4, pp.177-198, June 2000.

[38] G. Rothermel, Mary Jean Harrold. "Regression Test Selection for C++ Software.", Technical Report 99-06-01, Computer Science Department, Oregon State University, January 1999.

[39] Cheng J, "Slicing Concurrent Programs.", Proc. First Int'l Workshop Automated and Algorithmic Debugging, Linkoping, Sweden, pp.244-261, 1993.

[40] Loren D. Larsen and Mary Jean Harrold, "Slicing Object-Oriented Software.", Proceedings of ICSE-18. pp. 495-505, March 1996.

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

[42] John Hatcliff, Matthew B. Dwyer, Hongjun Zheng, "Slicing Software for Model Construction.", Higher-Order and Symbolic Computation, Volume 13, Number 4, pp.315-353, December 2000.

[43] Saurabh Sinha, Mary Jean Harrold, Gregg Rothermel,

"System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow.", International Conference on Software Engineering (ICSE), Los Angeles, CA, USA, pp.432-441, 1999.

[44] Zhengqiang Chen, Xu Baowen, Hongji Yang, "Slicing Tagged Objects in Ada.", Ada-Europe, Leuven, Belgium, pp.100-112, 2001.

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

[46] Rothermel G. and Harrold M. J., "Selecting tests and identifying test coverage requirements for modified software.", Proceedings of the ACM International Symposium on Software Testing and Analysis, pp.169-184, August 1994.

[47] Jeanne Ferrante, Karl J. Ottentein, and Joe D. Warren. "The program dependence graph and its uses in optimization.", ACM Trans. on Programming Languages and Systems, pp.319-349, July 1987.

[48] Karl J. Ottentein and Linda M. Ottentein. "The program

dependence graph in a software development environment.", Proc. of the ACM SIG SOFT/SIGPLAN Symposium on Practical Software Development Environments, Pittaburgh, Pennsilvania, April 1984.

[49] Robin Milner, Mads Tofte, and Robert Harper. "The Definition of Standard ML.", The MIT Press, Cambridge, MA, 1990.

[50] Choi, J.-D., Miller, B., and Netzer, R. "Techniques for debugging parallel programs with flowback analysis.", ACM Transactions on Programming Languages and Syatems 13, 4 (1991), pp.491-530, 1991.

[51] Ferrante J., Ottenstein K. J., and Warren J. D., "The program dependence graph and its use in optimization.", ACM Transaction on Programming Languages and Systems, pp.319-349, July 1987.

[52] K. Gallagher, J. Lyle, "Using program slicing in software maintenance.", IEEE Tran. on Software Engineering, vol.17, No.8, pp.751-761, 1991.

[53] Park, S. H. and Park, M. G., "An efficient dynamic program slicing algorithm and its Application.", Proc. of the IASTED

International Conference, Pittsburgh, Pennsylvania, pp.459-465, May 1998.

[54] 박순형, 박만곤, "소프트웨어 테스팅을 위한 동적 프로그램 슬라이싱 알고리즘의 효율성 비교.", 정보처리학회논문지 제5권 제9호, pp.2323-2334, 1998.

[55] 박순형, 박만곤, "다중 기준변수를 사용한 동적 프로그램 슬라이싱 알고리즘의 효율성 비교.", 한국정보처리학회 논문지, 제6권, 제9호, pp.2384-2392, 1999. 9.

[56] 박순형, 정은이, 박만곤, "동적 제어 정보를 이용한 효율적인 프로그램 슬라이싱 알고리즘.", 한국멀티미디어학회 논문지, 제3권 제1호, pp.92-99, 2000. 1.

[57] 박순형, 정은이, 박만곤, "동적 슬라이싱 기법을 이용한 프로그램 버전들의 통합 알고리즘.", 한국정보처리학회 논문지, 제7권, 제3호, pp.831-841, 2000. 3.

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

[59] 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.

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

감사의 글

먼저 정체되어 있지않고 동적인 지식인으로서 항상 다른 사람 보다 한발 앞서 나가셔서 이끌어 주시는 지도교수이신 박만곤 교수님께 감사를 드립니다. 그리고, 자상한 가르침을 주신 김영봉 교수님, 멋진 삶의 철학이 가득하신 김태화 교수님, 심사 과정에서 많은 가르침을 주신 여정모 교수님, 바쁘신 와중에서도 많은 조언과 격려를 해주신 양해술 교수님께도 깊은 감사를 드립니다. 그리고, 평소 많은 학생들로부터 존경을 받으시는 이경현 교수님과 김창수 교수님께도 깊은 감사의 말씀을 드립니다.

소프트웨어공학 및 멀티미디어 정보처리 연구실을 졸업한 정은이 선생님과 수료한 최원석 선생님과 이소영 선생님께도 많은 도움에 대한 감사의 말씀을 전합니다. 그리고, 연구실에서 같이 세미나를 했던 많은 졸업생들과 재학생들에게 대한 깊고 아름다운 추억은 영원히 잊지 못할 것입니다.

오늘이 있기까지 항상 따뜻한 사랑과 격려를 해주시는 진해시 문화원장이신 아버님과 항상 무엇인가 베풀어 주실려고만 하시는 어머님께도 감사의 말씀을 전합니다. 항상 멀리서 지켜봐 주시며 편안하게 대해주시는 장인과 장모님께도 깊은 감사를 드립니다. 그리고, 동생 소영이와 인재 그리고 내조를 아끼지 않는 아내와 사랑하는 딸 예진이와 이 기쁨을 함께 하고자 합니다.