선택 프로그램 슬라이싱을 이용한 소프트웨어 분석
(Software Analysis through Selection Program Slicing)

박 수 희 †
(Suehee Pak)

요 약 시스템이 진화와 노화를 거듭하는 경우 프로그램을 이해하는데 있어서 가장 정확한 문서는 기존의 시스템 자체에서 추출되어진 정보이다. 프로그램의 이해를 돕기 위한 분석기법중의 하나인 프로그램 슬라이싱은 1984년 Weiser에 의해서 소개되어진 이후 다양한 방향으로 연구가 진행되어져 왔다. 특히, 생성되어지는 프로그램 슬라이스가 사이즈가 크고 충분히 집약적이지 못하다는 문제점과 슬라이스의 계산방법이 효율적이지 못하다는 문제점을 극복하기 위하여 여러 가지 방법들을 시도되어 왔다. 본 논문은 고전적인 슬라이싱 기준에 새로운 파라미터를 추가함으로써 *제외*개념을 이용하여 보다 함수적으로 응집력있고, 사이즈가 작은 슬라이스를 생성하는 선택 슬라이스를 제시하며 이 *선택*슬라이스를 계산하기 위한 효율적인 방법을 기술한다. 데이타와 제어의 흐름을 이용하여 선택 슬라이싱을 정의하고, 선택 슬라이싱을 위한 종속 그래프를 정의하고 이를 사용하여 선형적 시간 내에 선택 슬라이스를 생성하는 알고리즘을 기술한다.

Abstract For the system that has evolved over years, the best document for program comprehension is the information to extract and to recover from the existing system. Program slicing, introduced by Weiser, has been studied since 1984, but most of works still have the problem that the resulted program slice is not manageable in size and in cohesiveness and the problem that the algorithm to calculate the slice is not efficient enough. Focusing on overcoming these problems, we extend the classical program slicing to an advanced slicing technique, *Selection Slicing*. The effective method to calculate the *Selection Slice* is also presented. Another parameter is added to the conventional slicing criterion so that the concept of *exclusiveness* makes the resulted slice smaller and functionally cohesive. This paper also proposes both the definition of the Selection slicing and the representation of a dependence graph, on which an effective algorithm to produce the correct Selection slice in linear time is described.

1. 서 론

1970년 이후 소프트웨어는 빠른 속도로 진화, 노화해 왔다. 소프트웨어 유지보수에 드는 비용이 전체 소프트웨어 생명주기에 투자되는 비용의 70%이상을 차지한다는 통계[1]도 있다. 이 자료가 1980년도에 보고됐다는 사실과 그 후의 소프트웨어가 성장, 발전해온 속도를 감안하면, 유지보수에 대한 비중이 점점 증가할 것임은 더 강조할 필요가 없다. 특히 노후시스템(legacy system)의 경우, 문서화가 제대로 되어있지 않고, 또 문서화가 되어있다 하더라도 소프트웨어의 빠른 진화를 반영하지 않는 경우가 대부분이어서 유지보수 관련 활동들은 소프트웨어 위기의 문제를 직면하고 있다.

이러한 소프트웨어 위기를 극복하기위한 방안중의 하나로 유지보수과정을 직접, 간접으로 지원하는 CASE (Computer Aided Software Engineering) 도구에 관한 연구와 개발이 최근 몇 년간 활발히 진행되어왔다. 그 중 프로그램 슬라이싱(slicing)은 1984년 Weiser[2]에 의해 처음 주창 된 이후 현재까지 미국, 유럽등지의 많은 연구소나 대학에서 연구되어온 기법으로 다양한 변형이 등장하였고 그 중 일부는 프로토타입 CASE도구로 이미 개발되었거나 현재 진행 중에 있다. 특히, C로 작성된 원시코드의 노후화가 진행되어온 1990년도 이후는 C언어를 대상으로 한 많은 연구가 수행되어져 왔다.

1984년 Weiser는 프로그래머들이 프로그램을 이해할 때 큰 프로그램을 그들이 원하는 어떤 것을 기준으로

본 연구는 '98 한국학술 진흥재단 신진교수 연구관지(과제번호 1998-003-200431)의 연구비지원에 의한 것임.
† 정 회 원 : 동덕여자대학교 컴퓨터학과 교수
pak@dongduk.ac.kr
논문접수 : 2000년 8월 30일
심사완료 : 2001년 4월 30일

머리속에서 작은 조각들로 분해하여 그들이 필요한 것만을 보고 이해한다는 것에 착안해 프로그램 슬라이싱이라는 개념을 처음 제시하였다[2]. 프로그램 슬라이싱이란 슬라이싱 기준(Slicing Criterion), <i, L>이 주어지면 문장위치 i에서 변수들 L에 영향을 줄 수 있는 문장들, 프로그램 슬라이스(Program Slice)을 찾아내는 일종의 분해(Decomposition)기법이다. 디버깅을 목적으로 하였던 초기에는 실행가능한 슬라이스만을 다루었으나 프로그램이해등을 실행되지않는 프로그램 슬라이스(non-executable slice)도 필요에 의해서 슬라이스의 범주에 넣고있다. 본 논문에서도 역시 생성되는 결과물을 실행가능한 슬라이스로 국한하지 않는다.

프로그램 슬라이스는

<i, L>

라는 슬라이스 기준에 대하여 프로그램 문장 i에서 변수 V∈L의 값에 직접, 간접으로 영향을 미치는 프로그램의 문장들로서 구성된다.

Weiser는 이러한 프로그램 슬라이스를 계산하여 생성하기 위하여 제어흐름 그래프(Control Flow Graph)를 이용하였다. 본 논문에서는 데이타와 제어의 흐름의 이용하여 프로그램 슬라이싱을 정의하고 예를 보이며 효율적으로 프로그램 슬라이스를 생성해내기 위해서 제어흐름그래프대신 종속그래프를 이용하여 계산 알고리즘을 제시한다.

1984년 이후 프로그램 슬라이싱은 세 가지 방향으로 발전되어져 왔다. 첫째는 슬라이싱의 기준(criterion)을 변화시킴으로써 다양한 슬라이스를 정의하고자 하는 슬라이싱의 '변형'에 관한 연구이다. Weiser의 고전적인 슬라이싱이래로 프로그램 다이싱[13], 동적 슬라이싱[14, 15], 분해 슬라이싱[3], 조건 슬라이싱[16] 등으로 다양하게 변형되어져 왔다.

둘째는 슬라이싱의 '방법'에 관한 연구로 슬라이스 생성을 위한 알고리즘을 다룬다. 이 연구는 정확한 슬라이스를 생성하는 것과 슬라이스의 생성을 효과적으로 하는데 연구의 초점을 맞춘다. 현재까지 제어 흐름 그래프를 사용하는 방법과 종속 그래프를 사용하는 알고리즘이 가장 많이 연구되어졌다.

셋째는 생성된 슬라이스와 유지보수 활동과의 연결을 구체화하기 위한 슬라이스의 '활용'에 관한 연구이다. 다양한 슬라이싱 기준에 의해 생성된 슬라이스들은 프로그램 디버깅, 프로그램 통합, 프로그램 이해 그리고 소프트웨어 유지보수 등에 활용되어질 수 있다.

본 연구는 위의 첫째와 둘째에 해당하는 슬라이싱의 변형과 슬라이싱의 방법에 관한 연구이다. 첫째 방향인 슬라이싱의 변형에 관한 연구들[9, 14]은 생성되어지는 슬라이스가 사이즈가 크고 충분히 집약적이지 않다는 문제점을 여전히 안고 있다. 이러한 문제점을 해결하기 위해 본 연구에서는 고전적인 슬라이싱 기준보다 많은 파라미터를 첨가함으로써, 함수적으로 집약적이며 크기가 작은 슬라이스를 만드는 데에 초점을 둔다. 둘째 방향인 슬라이싱 방법에 관한 연구를 위하여 프로그램 슬라이스를 계산하는 가장 효과적인 방법인 종속 그래프를 이용하였다. 이는 사이즈가 작고 함수적으로 집약된 슬라이스를 제시하였으나 그 계산방법을 효과적으로 제시하지 못한 변형슬라이싱[4]에서 향후과제로 남긴 종속그래프를 사용하는 문제를 본 논문에서는 변형된 종속그래프를 이용하여 해결하였다. 이를 위하여 선택슬라이싱을 위한 변형된 종속그래프를 새롭게 정의하고 이를 이용하여 선택슬라이스를 계산하는 알고리즘을 기술한다.

본 논문에서 제시하는 선택 슬라이스 기준은 다음의 세가지 파라미터로 구성된다.

<i, L_{exc}, L_{inc}>

이때, i는 프로그램의 문장을 나타내며, L_{exc}와 L_{inc}는 프로그램 내에서 사용되는 변수들의 집합들로 구성된다. 선택 슬라이스는 i의 위치에서 L_{inc}의 값에 영향을 미치는 부분을 포함하며 L_{exc}의 값에 영향을 미치는 부분은 제외한 프로그램 코드를 의미한다. 이때 L_{inc}는 임의의 변수가 아니고 문장 i에서 정의되거나 사용되어진다고 가정한다.

본 논문에서는 고전적 슬라이싱의 발전된 변형으로 새로운 슬라이싱 기법인 선택 슬라이싱을 정의하고 그 선택 슬라이스를 계산하는 방법을 제시한다. 2절에서는 고전적 슬라이싱의 정의를 이용, 이를 확장하여 선택 슬라이싱을 새롭게 정의한다. 3절에서는 현재까지 가장 효과적인 방법으로 사용되어지고 있는 종속 그래프를 선택 슬라이싱에 맞게 정의해보이고 이 종속그래프를 이용하여 선택 슬라이싱을 계산하기 위한 알고리즘을 제시한다. 4절에서는 관련연구로서 주로 슬라이싱 방법의 변형에 관한 다양한 연구들을 보여주고, 본 논문의 선택 슬라이싱과 가장 유사한 변환슬라이싱을 비교, 설명한다. 결론에 해당하는 5절에서는 선택 슬라이싱의 의미를 정리하고 향후 연구과제를 제시한다.

2. 선택 슬라이스

2.1 배경

2.1절에서는 슬라이싱에 관한 기본 용어에 대한 기본 정의와 고전적 프로그램 슬라이싱에 대한 정의를 보여

준다. 이 논문에서 새로이 정의된 부분과 차별화하기 위하여 인용부호 ‘ ’를 이용하였다.

‘정의1’ : *다이그래프(Digraph) G*는 <N, E>로 나타내며, *N*은 유한개의 노드의 집합이고, *E*는 {(n_i, n_j) | n_i, $n_j \in N$}인 에지(edges)의 집합이다. 에지 (n_i, n_j) $\in$ E가 주어지면 n_i는 *n_j의 선행자*(predecessor)라 하고 n_j는 *n_i의 후행자*(successor)라 하고 그 집합을 각각 *PRED(n_j)*과 *SUCC(n_i)*로 표기한다. 일련의 노드들로 구성된 $n_1n_2 \cdots n_k$(k≥1이고 (n_i, n_{i+1})(i=1, 2, …, k-1))를 *n_1—n_k 워크*라 정의한다.

‘정의2’ : *해먹그래프(Hammock Graph) G*는 일종의 다이그래프로 두개의 노드, 초기 노드(initial node) n_I와 최종 노드(final node) n_F를 포함한다.

‘정의3’ : *제어흐름그래프(Control Flow Graph) G*는 일종의 해먹 그래프로 하나의 프로그램 프로시져를 표현한다. 제어흐름그래프의 노드는 N(G) 로 표현되며 지정문, 입출력문, 조건서술문, 그리고 프로시져 호출문에 대응된다. 제어흐름그래프의 에지는 E(G)로 표현되며 문장들 사이의 제어 흐름을 표현한다. 본 논문에서는 하나의 진입점(entry)와 하나의 진출점(exit)을 갖는 프로시져를 대상으로 하며 이 진입점과 진출점은 각각 n_I와 n_F과 대응된다.

‘정의4’ : G를 제어흐름그래프라 하고 m과 n을 G안의 노드라 하자. 모든 n—n_F워크가 m을 포함하면 *m이 n을 전진 지배(forward dominate)한다*고 정의한다. 또한 m ≠ n이고 m이 n을 전진 지배하면 *m이 n을 정확(properly) 전진 지배한다*고 정의한다. 또한 모든 n—n_F 워크에서 n을 정확 전진 지배하는 첫번째 노드를 *n의 직접(immediate) 전진 지배자, ifd(n)*이라 한다.

‘정의5’ : G를 제어흐름그래프라하고 n을 G안의 어떤 노드라 하자. 어떤 문장 m이 n—ifd(n)워크에 있으면—n과 ifd(n)은 제외하고—, m은 *n에 의하여 조건적(conditioned)*라 정의한다. n에 의하여 조건적인 문장의 집합을 *INFL(n)*으로 나타낸다.

‘정의6’ : 문장 n의 실행에서 변수 x에 값을 지정할 때 *변수 x는 문장 n에서 정의된다*고 정의한다. *DEF(n)*은 문장 n에서 정의되는 변수들의 집합이다. 문장 n의 실행에서 변수 x의 값이 사용되어질 때 *변수 x는 문장 n에서 사용된다*고 정의한다. *USE(n)*은 문장 n에서 참조되는 변수들의 집합이다.

Weiser[2]가 사용한 프로그램을 조금 수정하여 살펴보자(그림 1).

프로그램중의 프로시져 P에 관한 제어흐름 그래프를

```
1.begin
2.   read(m);
3.   i := 1;
4.   sum := 0;
5.   prod := 1;
6.   n := m * 10;
7.   if (n > 50)
8.      then while (i <= n)
9.             begin
10.              sum := sum + i;
11.              prod := prod * i;
12.              i++;
13.            end
14.  write(sum);
15.  write(prod);
16.end
```

그림 1 예제 프로그램

G라 가정할 때 P에 대한 슬라이싱은 다음과 같이 정의된다.

‘정의7’ : *슬라이스 기준은 C=<i, L>*로 표현되며 이때 i ∈ N(G), L은 P에서 나타나는 변수들의 부분집합이다.

슬라이스 기준 C=<i, L>에 대한 *슬라이스 S*는 문장 i이전에 L안의 변수들의 값에 영향을 미치는 문장들로 구성된다. 보다 정형적인 정의는 아래와 같다.

‘정의8’ : 슬라이싱 기준을 C=<i, L>이라 하자. 문장 n이 실행될 때에 *C와 관련된 변수들의 집합, $R_c^0(n)$*을 다음과 같이 정의한다.

$$R_c^0(n) = \{v \in L \mid n = i\} \cup$$
$$\{USE(n) \mid DEF(n) \cap R_C^0(SUCC(n)) \neq \emptyset\} \cup$$
$$\{R_C^0(SUCC(n)) - D(n)\}$$

$R_c^o(n)$은 문장 n의 위치에서L에 속한 변수에 영향을 미치는 변수들의 집합이다. 이를 위하여 문장 i부터 시작하여 후진탐색(backward traversal search)을 하며 관련 변수들을 찾게 된다. 위의 첨자 0은 초기 후진탐색을 의미하며 $R_c^o(n)$는 초기 후진탐색의 결과, L에 속한 변수에 영향을 미치는 변수들을 의미한다. $R_c^x(n)$는 (x-2)번째 후진 탐색에서 얻어진 변수들에 영향을 미치는 변수를 찾는 (x-1)번째 후진 탐색의 결과 얻어진다.

‘정의9’ : 슬라이싱 기준을 C=<i, L>이라 하자. *C에 관련된 문장들의 집합, S_C^0*을 다음과 같이 정의한다.

$$S_C^0 = i \cup \{n \in G \mid DEF(n) \cap R_C^0(SUCC(n)) \neq \emptyset\}$$

S_C^0은 관련 변수에 직접 영향을 미치는 문장들을 포함

한다.

'정의10' : 슬라이싱 기준을 C=<i, L>이라 하자. S_C^0*의 실행을 제어하는 조건문의 집합,* B_C^0을 다음과 같이 정의한다.

$$B_C^0 = \{b \in G \mid INFL(b) \cap S_C^0 \neq \emptyset\}$$

L에 속하는 변수들에 직접, 간접으로 영향을 미치는 변수들과 문장들의 집합에 대하여 순환적으로(recursively) S_C가 정의되어진다. 0부터 시작하여 위첨자는 순환의 레벨을 나타낸다.

$$R_C^{i+1}(n) = R_C^i(n) \cup R_{\langle b, U(b)\rangle}{}^0(n) \text{ for all } b \in B_C^i$$

$$S_C^{i+1} = \{n \in G \mid DEF(n) \cap R_C^{i+1}(SUCC(n)) \neq \emptyset\} \cup B_C^i$$

$$B_C^{i+1} = \{b \in G \mid INFL(b) \cap S_C^{i+1} \neq \emptyset\}$$

'정의11' : 슬라이싱 기준 C = <i, L>에 대한 *슬라이스* S_C는

$$\forall n \in N : R_C^{f+1}(n) = R_C^f(n) = R_C(n)$$

인 경우의 S_C^{f+1}가 된다. 이때 f는 후진 탐색의 회수를 나타낸다.

고전적인 슬라이싱의 예

그림 1의 예제 프로그램에 대하여 슬라이싱 기준 C =<i=14, L={sum}>가 주어졌을 때 생성되는 프로그램 슬라이스 S_C는 그림 2에 반전되어 표시된다.

```
1.      begin
2.        read(m);
3.        i := 1;
4.        sum := 0;
5.        prod := 1;
6.        n := m * 10;
7.        if (n > 50)
8.          then while (i <= n)
9.                 begin
10.                sum := sum + i;
11.                prod := prod * i;
12.                i := i + 1;
13.              end
14.   write(sum);
15.   write(prod);
16.end
```

그림 2 기준 C=<i=14, L={sum}>에 대한 고전적인 프로그램 슬라이스

2.2 선택 슬라이스의 정의

선택 슬라이싱에 대한 기본 정의와 선택 슬라이스를 정의하기 위한 공식이 기존 슬라이싱에 관한 정의를 토대로 정의된다.

정의12 : *선택 슬라이스 기준은* $C=<i, L_{exc}, L_{inc}>$로 표현되며 이때 i는 프로그램 문장, L_{exc}, L_{inc}는 P에서 나타나는 변수들의 부분집합이다.

슬라이스 기준 C=<i, L_{exc}, L_{inc}>에 대한 *선택 슬라이스* $SelS_C$는 i의 위치에서 L_{inc}의 값에 영향을 미치는 부분을 포함하며 L_{exc}의 값에 영향을 미치는 부분은 제외한 프로그램 코드를 의미한다. 이때 L_{inc}는 임의의 변수가 아닌 L_{inc}는 i에서 정의되거나 사용되어지는 변수로 제한한다. 보다 정형적인 정의는 아래와 같다.

정의13 : 선택 슬라이스 기준을 C=<i, L_{exc}, L_{inc}>이라 하자. *문장 n이 실행될 때에 C와 관련된 변수들의 집합,* $SelR_C^0(n)$을 다음과 같이 정의한다.

$$SelR_C^0(n) = \{v \in L_{inc} \mid n = i\} \cup$$
$$\{USE(n) - L_{exc} \mid DEF(n) \cap SelR_C^0(SUCC(n)) \neq \emptyset\} \cup$$
$$\{SelR_C^0(SUCC(n)) - D(n)\}$$

$SelR_C^0(n)$은 문장 n의 위치에서 L_{inc}에 속한 변수에 영향을 미치는 변수들을 포함하며 L_{exc}에 속하는 변수들을 제외한다. 정의8과 같이 탐색은 문장 i부터 시작하여 후진 탐색을 하며 변수들을 찾아내는데 L_{exc}에 속하는 변수들은 제외시킨다.

정의14 : 선택 슬라이스 기준을 C=<i, L_{exc}, L_{inc}>이라 하자. *C와 관련된 문장들의 집합,* $SelR_C^0$을 다음과 같이 정의한다.

$$SelS_C^0 = i \cup \{n \in G \mid DEF(n) \cap SelR_C^0(SUCC(n)) \neq \emptyset\}$$

정의15 : 선택 슬라이스 기준을 C=<i, L_{exc}, L_{inc}>이라 하자. $SelR_C^0$*의 실행을 제어하는 조건문의 집합,* $SelR_C^0$을 다음과 같이 정의한다.

$$SelB_C^0 = \{b \in G \mid INFL(b) \cap SelS_C^0 \neq \emptyset\}$$

Weiser의 프로그램 슬라이스처럼, 선택 슬라이스도 다음과 같이 순환적으로 생성되어진다.

$$SelR_C^{i+1}(n) = SelR_C^i(n) \cup SelR_{\langle b, Lexc, USE(b)-Lexc\rangle}{}^0(n) \quad \text{for } all\ b \in SelB_C^i$$

$$SelR_C^{i+1} = \{n \in G \mid DEF(n) \cap SelR_C^{i+1}(SUCC(n)) \neq \emptyset\} \cup SelB_C^i$$

$$SelB_C^{i+1} = \{b \in G \mid INFL(b) \cap SelS_C^{i+1} \neq \emptyset\}$$

정의16 : 선택 슬라이스 기준 C=<i, L_{exc}, L_{inc}>에 대한 *선택 슬라이스* $SelS_C$는

$$\forall n \in N : SelR_C^{f+1}(n) = SelR_C^f(n) = SelR_C(n)$$

인 경우의 $SelS_C^{f+1}$가 된다. 이때 f는 후진 탐색의 회수를 나타낸다.

위의 공식은 고전적인 슬라이싱의 경우와 유사하다. 차이점은 문장마다 관련 변수를 선정함에 있어 L_{exe}에 속하는 변수를 제외시킨다는 것이다. 이 결과로 크기가 작은 슬라이스가 생성되게 된다.

선택 슬라이싱의 예

그림 1의 예제 프로그램에대하여 선택 슬라이싱 기준 C=<i=14, L_{exc}={n}, L_{inc}={sum}>가 주어졌을 때, 각 실행문에 대하여 다음과 같은 집합을 구할 수있다.

$SelR_C^0(14)=\{sum\}$ $SelR_C^0(12)=\{sum\}$
$SelR_C^0(11)=\{sum\}$ $SelR_C^0(10)=\{sum, i\}$
$SelR_C^0(8)=\{sum, i\}$

문장 8로부터는 문장 7과 문장 12로 두 가지 방향으로 진행된다.

$SelR_C^0(7)=\{sum, i\}$ $SelR_C^0(6)=\{sum, i\}$
$SelR_C^0(5)=\{sum, i\}$ $SelR_C^0(4)=\{i\}$
$SelR_C^0(3)=\varnothing$ $SelR_C^0(2)=\varnothing$
$SelR_C^0(12)=\{sum, i\}$ $SelR_C^0(11)=\{sum, i\}$
$SelR_C^0(10)=\{sum, i\}$

이하의 문장 8→문장 7에서는 변동처리가 없다. 순환회수 0에 대하여 다음과 같은 중간산물을 얻게된다.

$SelS_C^0=\{3,4,10,12,14\}$ $SelB_C^0=\{7,8\}$
$SelR_{<7,\{n\},\{n\}>}^0(7)=\varnothing$
$SelR_{<8,\{n\},\{i\}>}^0(8)=\{i\}$

문장 8로부터는 문장 7과 문장 12로 두가지 방향으로 진행된다.

$SelR_{<8,\{n\},\{i\}>}^0(7)=\{i\}$ $SelR_{<8,\{n\},\{i\}>}^0(6)=\{i\}$
$SelR_{<8,\{n\},\{i\}>}^0(5)=\{i\}$ $SelR_{<8,\{n\},\{i\}>}^0(4)=\{i\}$
$SelR_{<8,\{n\},\{i\}>}^0(3)=\varnothing$ $SelR_{<8,\{n\},\{i\}>}^0(2)=\varnothing$

$SelR_{<8,\{n\},\{i\}>}^0(12)=\{i\}$ $SelR_{<8,\{n\},\{i\}>}^0(11)=\{i\}$
$SelR_{<8,\{n\},\{i\}>}^0(10)=\{i\}$ $SelR_{<8,\{n\},\{i\}>}^0(8)=\{i\}$
$SelR_{<8,\{n\},\{i\}>}^0(7)=\{i\}$ $SelR_{<8,\{n\},\{i\}>}^0(6)=\{i\}$
$SelR_{<8,\{n\},\{i\}>}^0(5)=\{i\}$ $SelR_{<8,\{n\},\{i\}>}^0(4)=\{i\}$
$SelR_{<8,\{n\},\{i\}>}^0(3)=\varnothing$ $SelR_{<8,\{n\},\{i\}>}^0(2)=\varnothing$
$SelR_C^1(14)=\{sum\}$ $SelR_C^1(12)=\{sum\}$
$SelR_C^1(11)=\{sum, i\}$ $SelR_C^1(10)=\{sum, i\}$
$SelR_C^1(8)=\{sum, i\}$ $SelR_C^1(7)=\{sum, i\}$
$SelR_C^1(6)=\{sum, i\}$ $SelR_C^1(5)=\{sum, i\}$
$SelR_C^1(4)=\{i\}$ $SelR_C^1(3)=\varnothing$ $SelR_C^1(2)=\varnothing$

순환 회수 1에 대하여 다음과 같은 중간 산물을 얻게 된다. 이때 순환 회수를 증가하여도 같은 중간산물을 구하게 되므로 이것이 최종 슬라이스가 된다.

$SelS_C=SelS_C^1=\{3, 4, 7, 8, 10, 12, 14\}$

생성되는 선택 슬라이스는 (그림 3)에 반전되어 표시된다.

```
1.  begin
2.    read(m);
3.    i := 1;
4.    sum := 0;
5.    prod := 1;
6.    n := m * 10;
7.    if (n > 50)
8.      then while (i <= n)
9.    begin
10.     sum := sum + i;
11.     prod := prod * i;
12.     i := i + 1;
13.   end
14.  write(sum);
15.  write(prod);
16. end
```

그림 3 기준 C=<i=14, L_{exc}={n}, L_{inc}={sum}>에 대한 선택 슬라이스

3. 선택 슬라이싱

이 절에서는 위에서 정의한 선택 슬라이스를 계산하는 방법, 즉 *선택 슬라이싱*을 설명한다. 이 논문에서 대상으로 하는 언어는 C이지만 그 원리는 언어에 독립적으로 다른 언어에 쉽게 적용될 수 있다.

1984년 Weiser이후로 슬라이스 계산을 위한 다양한 방법들이 제시되었왔다. 초기에는 주로 제어흐름 그래프(Control Flow Graph: CFG)와 데이타 흐름 방정식을 이용하여 슬라이스를 계산하였으나 현재는 종속 그래프(Dependency Graph)를 사용하는 방법이 효율성이나 정확성 면에서 더 활발히 연구되고 있다. 그 중 [11]에서는 종속그래프를 이용한 완성도 있는 방법을 구체적이고 정형적으로 제시하고있다. CFG를 이용하는 것보다 -여러 개의 슬라이스를 생성하고자 할 경우- 빠른 속도로 슬라이스를 생성하는 것이 가능하고, 보다 정확한 슬라이스의 생성이 가능함을 상세히 보여준다[11].

선택 슬라이싱과 유사한 변환 슬라이싱[4]은 비효율적이고 부정확한 CFG를 사용하여 슬라이스를 생성하고 있다. 반면에 본 연구에서의 선택 슬라이싱은 종속 그래프를 사용하여 선택 슬라이스를 생성한다.

이 절은 다음의 순서로 진행된다. 3.1절은 하나의 프로시져로 구성된 프로그램을 표현하기 위해 사용되어지는 프로그램 종속 그래프(Program Dependency Graph: PDG)를 정의한다. 이는 기존의 PDG를 선택 슬라이싱에 맞게 변형한 형태의 그래프이다. 3.2절에서는 정의된 PDG에서 프로시져 내에서의 슬라이싱(intraprocedural slicing)을 위한 계산 알고리즘을 제시한다. 3.3절은 여러 개의 프로시져가 호출구조로 구성된 프로그램을 표현하기위한 시스템 종속 그래프(System Dependence Graph: SDG)로 PDG의 정의를 확장한다. 3.4절에서는 정의된 SDG에서 프로시져 간의 슬라이싱(interprocedural slicing)을 위한 계산 알고리즘을 제시하고 호출되는 프로시져의 호출 문맥을 정확하게 고려하는 방법을 보여준다.

3.1 선택 슬라이싱을 위한 프로그램 종속 그래프의 정의

프로그램 종속관련 표현에 관한 다양한 정의들이 제시되어져 왔는데 이는 활용하려는 응용분야에 따라 그 모양과 기능을 조금씩 달리한다. 컴파일러나 프로그램 개발 환경을 벡터라이징하거나 병렬화(parallelizing)하는데 사용되었던 프로그램 종속 그래프는 Ottenstein에 의해서 프로그램 슬라이스를 계산하는데 가장 이상적인 표현이 될 것이라 제시되었다[17].

본 논문에서 제시하는 프로그램 종속 그래프는 효시격인 [17]보다 제한된 언어를 처리하였다는 점에서 덜 일반적이다. 단순 변수나 지정문, 조건문, 반복문 등과 제한된 형태의 출력문을 대상으로 프로그램 종속 그래프를 정의하였다. [11]의 종속그래프에서 정의된 제어종속(control dependency)와 데이타종속(data dependency)의 표현에 추가적으로 데이타 흐름 분석에 관한 정보를 추가하여 선택 슬라이싱을 위한 종속 그래프를 만든다.

프로그램 종속 그래프를 위한 정형적이고 상세한 정의는 [11]를 따르며 여기서는 선택슬라이싱에 관련된 부분에 초점을 맞춘다. 선택 슬라이싱을 위한 프로그램 종속 그래프를 설명함에 있어, [11]와 일치하는 부분은 간략하게 정의하며 [11]와 다른 부분은 상세하게 정의되어 진다.

프로그램 P에 대한 프로그램 종속 그래프, G는 노드와 에지로 구성되는 일방향 그래프(directed graph)이다. 이때 노드들은 지정문, 제어서술문(control predicate)과 엔트리를 표현한다. 에지들은 프로그램 컴포넌트간의 *종속성*을 표현한다. 이때 종속성은 크게 *제어 종속성*(control dependence)과 *데이타 종속성*(data dependence)으로 나뉘며 데이타 종속성은 다시 *흐름 종속성*(flow dependence)와 *정의-순서 종속성*(def-order dependence)으로 나눈다.

v_1노드에서 v_2노드로의 *제어 종속 에지*는 $v_1 \rightarrow_c v_2$로 표기되며 그 의미는 다음과 같다.

> 실행 시에, v_1으로 표현되어지는 서술문이 평가(evaluation)되고, 그 평가된 값과 v_2로의 에지의 값(참 또는 거짓값)이 일치할 때 v_2가 실행되어진다.

이때, v_1은 엔트리나 제어서술문 노드이다. v_1이 엔트리이거나 반복을 나타내는 제어서술문인 경우는 참값을 나타내는 T(rue) 라벨이 붙으며 v_1이 조건을 나타내는 제어서술문인 경우는 then 브렌치인가 else 브렌치인가에 따라 T(rue), F(alse) 라벨이 붙는다. 보다 정형적인 정의는 [11]에 정의된다.

v_1노드에서 v_2노드로의 *데이타 종속 에지*의 의미는 다음과 같다.

> v_1노드와 v_2노드의 순서를 바꾸면 프로그램의 계산 결과가 변하게 된다. 데이타 종속 에지는 데이타 흐름 분석을 통하여 계산되어지며 흐름 종속(flow dependence) 에지와 정의-순서 종속(def-order dependence) 에지로 나누어 진다. 선택 슬라이싱을 위한 종속 그래프를 만드는데 있어 이들의 정의는 다음과 같다. [11]에서의 종속그래프보다 본 연구의 종속 그래프는 데이타 종속에 관한 정보를 추가적으로 포함한다.

정의17 : 다음의 모두를 필요충분조건으로 v_1노드에서 v_2노드에로의 *흐름 종속 에지*를 정의하며 $v_1 \rightarrow_{f(x)} v_2$으로 표기되어진다.

(1) v_1은 x를 정의하는 노드이다.

(2) v_2는 x를 사용하는 노드이다.

(3) v_1와 v_2사이에, x에 관한 정의가 없는 실행경로가 하나라도 있어야 한다.

흐름 종속성은 *루프 전달*(loop carried)과 *루프 독립*(loop independent)으로 더 세분되어질 수 있다.

정의18 : 위의 (1), (2), (3)에 더하여 다음을 필요충분조건으로 흐름 종속성이 루프 LP에 의해서 *루프 전달*된다고 정의하며 $v_1 \rightarrow_{lc(LP,\ x)} v_2$으로 표기되어진다.

(4) 위의 (3)의 조건을 만족시키고 루프 LP의 서술문으로의 백에지(backedge)를 포함하는 실행 경로가 있다.

(5) v_1과 v_2 모두 루프 LP안에 포함된다.

정의19 : 위의 (1), (2), (3)과 다음의 (4)를 필요충분조건으로 흐름 종속성을 *루프 독립*적이라 정의하며 $v_1 \rightarrow_{li(x)} v_2$으로 표기되어진다.

(4)′ 위의 (3)의 조건을 만족시키고 v_1과 v_2를 포함하

는 루프의 서술문으로의 백에지를 포함하지 않는 실행경로가 있다.

다음은 정의-순서 종속 에지에 관한 정의이다.

정의20 : 다음의 모두를 필요충분 조건으로 프로그램 종속 그래프는 v_1노드에서 v_2노드에로의 *정의-순서 종속 에지*를 정의하며 $v_1 \rightarrow_{do(x,v3)} v_2$로 표기되어진다.

(1) v_1과 v_2는 모두 같은 변수 x를 정의한다.

(2) v_1과 v_2는 이들을 포함하는 조건문에 대하여 같은 브랜치에 위치한다.

(3) 임의의 x에 대해서 $v_1 \rightarrow_{f(x)} v_3$와 $v_1 \rightarrow_{f(x)} v_3$인 v_3가 존재한다.

(4) 프로그램의 추상 문법 트리(abstract syntax tree)에서 v_1은 v_2의 왼쪽에 위치한다.

이러한 성질을 가진 프로그램 종속 그래프를 이용하여 선택 슬라이스를 계산해보자.

3.2 프로시져 내의 선택 슬라이스 계산

이 절에서는 프로그램 종속 그래프를 사용하여 하나의 프로시져로 이루어진 프로그램을 슬라이스하는 방법을 다룬다. 슬라이스를 계산하는 알고리즘은 다음의 세 가지 단계로 이루어진다.

단계1. 프로그램으로부터 종속 그래프를 만든다.

단계2. 종속 그래프를 슬라이싱한다.

단계3. 슬라이스된 종속 그래프로부터 프로그램 슬라이스를 구한다.

종속 그래프를 이용한 슬라이스 계산에 있어서의 장점은 위의 단계2와 단계3이 선형적(linear)이라는 데 있다. 단계1은 프로그램의 문장의 수에 대해서 $O(n^2)$이다. 즉, 일단 종속 그래프를 생성한 다음에 여러 번의 슬라이스 계산을 수행할 때 이 방법은 매우 효과적이다. 4.1절에서 종속그래프의 정의를 기술하였다. 그림 4는 그림 1의 예제 프로그램에 대한 종속그래프를 보여준다. 종속그래프의 생성방법은 [17]에 기술되어있다. 이 절에서는 단계2를 알고리즘으로 설명한다. 단계3은 기계적이므로(straightforward) 다루지 않는다.

프로시져 P에 대한 PDG를 G라 하자. G의 노드들의 집합을 N(G)라 하고 에지들의 집합을 E(G)라 한다. 하나의 프로시져 P에서 슬라이싱 기준 $C=<i, L_{exc}, L_{inc}>$에 대한 선택 슬라이스 $SelS_C(P)$를 계산하기위하여 우선 프로그램 종속 그래프 G를 슬라이싱한다. 슬라이싱 기준 $C=<i, L_{exc}, L_{inc}>$에 대하여 슬라이스된 프로그램 종속 그래프의 노드 SlicedN(G, C)와 에지 SlicedE(G, C)는 다음과 같다.

$SlicedN(G,C)=\{w | w \in N(G)\} w \rightarrow_{c,f(x)} v_i$, v_i는 i와 L_{inc}에 대한 노드, $x \notin L_{exc}\}$

$SlicedE(G,C)=\{(v \rightarrow_c w) | (v \rightarrow_c w) \in E(G)$ and $v,w \in SlicedN(G,C)\}$

$\cup \{(v \rightarrow_{f(x)} w) | (v \rightarrow_{f(x)} w) \in E(G)$ and $v,w \in SlicedN(G,C)$ and $x \notin L_{exc}\}$

$\cup \{(v \rightarrow_{do(x,u)} w) | (v \rightarrow_{do(x,u)} w) \in E(G)$ and $v,w,u \in SlicedN(G,C)$ and $x \notin L_{exc}\}$

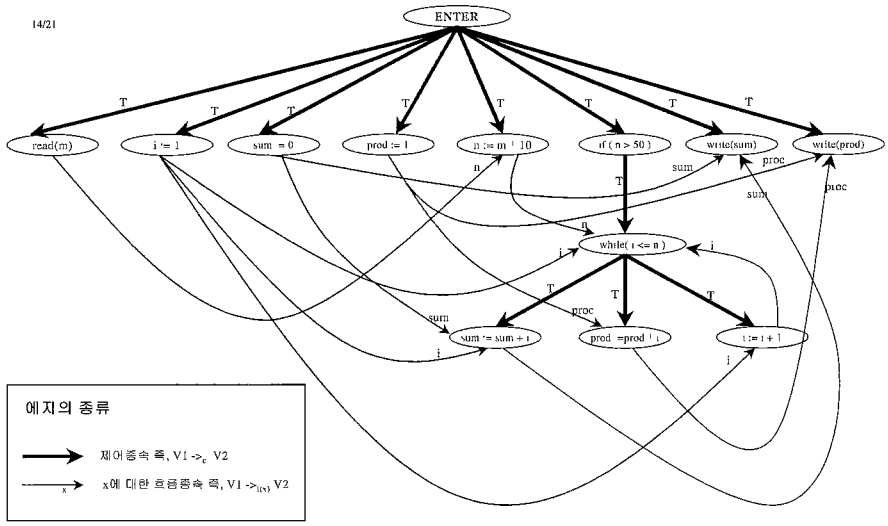

그림 4 그림 1의 예제 프로그램에 대한 프로그램 종속 그래프

선택 슬라이스를 생성하기 위한 대략적인 알고리즘이 그림5에 보여진다. $w \rightarrow_c^v$

```
procedure MarkNodesOfSlice(G, V, Lexc)
declare
        G: 프로그램 종속 그래프
        V: i와 Linc에 대한 노드들의 집합
        Lexc: 계산결과에 관심없는 변수들의 리스트
        WorkNodes: G에 속하는 노드들의 집합
        v, w: G에 속하는 노드
begin
   WorkNodes:=V
   while WorkNodes≠∅ do
      Select and remove vertex v from WorkNodes
      Mark v
      for each unmarked vertex w in E(G)
            such that edge w→f(x)v such that x∉Lexc
                  or edge w→cv
         do
            Insert w into WorkNodes
      od
   od
end
```

그림 5 슬라이스 기준 C=<i, L_{exc}, L_{inc}>에 대한 프로그램 종속 그래프의 슬라이싱

3.3 선택 슬라이싱을 위한 시스템 종속 그래프의 정의

프로시져 내의 슬라이싱에서 사용한 프로그램 종속 그래프와 슬라이싱 생성 알고리즘을 프로시져간의 슬라이싱에 적용하려면 종속 그래프와 알고리즘에 추가적인 작업을 필요로 한다. 이를 위해서 시스템 종속 그래프를 사용한다. 이는 [11]에서 필요성과 정의 및 슬라이싱 생성방법 등이 상세히 기술되었고 여기서는 선택 슬라이싱을 위한 부분만을 제시한다.

시스템 종속 그래프는 프로시져들과 프로시져 호출을 포함하는 프로그램을 표현하는 그래프적인 방법이다. 다음과 같이 언어를 제한하여 모델링한다.

(1) 하나의 main함수와 부수적인 프로시져들을 포함하는 완성된 시스템

(2) 파라미터는 call-by-value-result방법에 의하여 전달된다고 가정한다.

시스템 종속 그래프는 main함수에 대한 하나의 *프로그램 종속 그래프*와 호출되는 함수들에 대한 *프로시져 종속 그래프*로 구성된다. 추가적인 에지는 두 가지 종류로 나뉘는데, 이는 호출부분과 호출되는 프로시져간의 종속성을 표현해주는 *call* 에지와 호출로 인한 이행적(transitive) 종속성을 표현하는 에지로 나뉜다. 전자는 제어 종속 에지로 분류되며 후자는 다시 *파라미터-인*(parameter-in) 에지와 *파라미터-아웃*(parameter-out) 에지로 나뉘며 이는 데이타 종속 에지로 분류된다.

3가지의 새로운 에지 외에 다음과 같은 5가지의 새로운 노드가 추가된다. 이는 *호출*(call-site) 노드, *실질-인*(actual-in) 노드, *실질-아웃*(actual-out) 노드, *형식-인*(formal-in), *형식-아웃*(formal-out)이 이에 속한다.

[11]는 Weiser의 프로시져간의 슬라이싱 방법에 대한 부정확성을 지적하고 이를 해결하기 위한 방법으로 프로시져 호출의 결과에 의한 이행적 종속성을 표현하기 위한 새로운 에지를 추가하여 보다 정확한 슬라이스 생성 방법을 제시하였다. 이를 위해서 [11]에서는 연결 문법(linkage grammer)를 사용하여 이행 종속성 에지를 결정하였다. 본 연구의 선택 슬라이싱에서도 유사한 그래프 구조를 사용하여 그래프-도달(graph-reachability) 방법을 이용하여 슬라이스를 계산한다.

3.4 프로시져 간의 선택 슬라이스 계산

시스템 종속 그래프를 슬라이싱하는 알고리즘은 그림6에 제시된다. 슬라이스 기준 C=<i, L_{exc}, L_{inc}>에 대한 시스템 종속 그래프 G의 선택 슬라이스는 두가지 단계로 계산되어진다. 단계1과 단계2 모두 시스템 종속 그래프상에서 수행되어지며, 3.2절의 프로시져 내에서의 슬라이스 계

```
procedure MarkNodesOfSlice(G, V, Lexc)
declare
        G: 시스템 종속 그래프
        V: i과 Linc에 대한 노드들의 집합
        Lexc: 계산결과에 관심없는 변수들의 리스트
begin
   // 단계 1
      MarkReachingNodes(G, V, Lexc, {def-order, parameter-
         out})
   // 단계 2
      V :=all marked Nodes in G
      MarkReachingNodes(G, V, Lexc, {def-order, parameter-
         in, call})
end

procedure MarkReachingNodes(G, V, Kinds)
declare
   G: 시스템 종속 그래프
   V: 노드들의 집합
   Kinds: 에지종류들의 집합
   v, w: 노드들
   WorkNodes: 노드들의 집합
   WorkNodes := V
   while WorkNodes≠∅ do
      Select and remove vertex v from WorkNodes
      Mark v
      for each unmarked vertex w in E(G)
            such that edge w→f(x)v such that x∉Lexc
                  or edge w→cv
         do
            Insert w into WorkNodes
      od
   od
end
```

그림 6 슬라이스 기준 C=<i, L_{exc}, L_{inc}>에 대한 시스템 종속 그래프의 슬라이싱

산방법과 본질적으로 유사하다. 단계1은 호출되는 프로시져에 관한 부분을 제외하고 슬라이싱을 수행하며 단계2는 그 부분을 포함하며 호출 지점을 제외하고 슬라이스를 수행하는데 이 방법은 [11]에서 제시하였던 방법이다.

4. 관련연구와의 비교

이태리의 Bari대학의 변환 슬라이스[4]는 선택슬라이스와 유사하다. 변환 슬라이싱은 슬라이스 기준으로 세 개의 파라미터, n, V_{inp}, V_{out}을 사용하여 V_{inp}부터 시작하여 V_{out}에 영향을 미치는 프로그램 부분들을 추출해내며 V_{inp}에 직접적인 영향을 미치는 플래그를 포함한 조건문을 제외하여 변환 슬라이스를 생성한다. 여기에서 V_{inp}의 의미가 선택슬라이스의 L_{exc}와 와 제외의 개념을 이용하였다는 점에서는 유사하나 V_{inp}나 L_{exc}으로 인하여 제거되어지는 프로그램의 부분은 다르다는 차이점이 있다. V_{inp}의 존재는 이에 영향을 주는 플래그를 포함하는 조건문을 제외하는 것이 주 목적이며 L_{exc}의 경우에는 이에만 영향을 주는(L_{inc}에는 영향이 전혀 없는) 프로그램의 부분들을 제외시킴을 목적으로 한다.

변환 슬라이싱은 재사용 가능한 컴포넌트를 추출해내는데 활용시키는 데에 초점을 맞추며 영향력이 없는 플래그부분들을 슬라이스에서 제외시키므로, 실제로 변환 슬라이싱은 선택 슬라이싱보다 작은 슬라이스를 생성한다. 영향력이 없는 플래그를 제외시키기 위해서 나름의 공식을 이용하여 조건문을 선별하고 있으나 프로그램의 구조에 따라 제외되는 조건문이 문법적 맥락으로만 의미는 고려되지 않은채- 선택되어지는 단점이 있다. 본 논문의 선택 슬라이싱은 이에 비하여 단지 L_{exc}에만 영향을 주는 부분은 제외한다는 슬라이싱의 기본개념을 적용하고있으므로 변환슬라이싱보다 일반적이며 단순하기 때문에 적용이 쉽다는 장점이 있다. 변환슬라이싱[4]에서 사용한 프로그램으로 변환 슬라이싱, 선택 슬라이싱의 결과를 비교하면 다음과 같다.[그림 7, 그림 8, 그림 9]

```
1.  begin
2.     read(n);
3.     read(flag);
4.     i := 1;
5.     if flag = 1
6.        then begin
7.             sum := 0;
8.             while i <= n do
9.                begin
10.                  read(k);
11.                  sum := sum + k;
12.                  i := i + 1;
13.               end
14.            write(sum);
15.            end
16.     else begin
17.            prod := 1;
18.            while i <= n do
19.               begin
20.                  read(k);
21.                  prod := prod * k;
22.                  i := i + 1;
23.               end
24.            write(prod);
25.           end;
26.  end.
```

그림 7 예제 프로그램 2

```
1.  begin
2.     read(n);
3.     read(flag);
4.     i := 1;
5.     if flag = 1
6.        then begin
7.             sum := 0;
8.             while i <= n do
9.                begin
10.                  read(k);
11.                  sum := sum + k;
12.                  i := i + 1;
13.               end
14.            write(sum);
15.            end
16.     else begin
17.            prod := 1;
18.            while i <= n do
19.               begin
20.                  read(k);
21.                  prod := prod * k;
22.                  i := i + 1;
23.               end
24.            write(prod);
25.           end;
26.  end.
```

그림 8 기준 Ct=<i=14, V_{inp}={n}, V_{out}={sum}>에 대한 변환 슬라이스(반전한 부분)

```
1.  begin
2.     read(n);
3.     read(flag);
4.     i := 1;
5.     if flag = 1
6.        then begin
7.             sum := 0;
8.             while i <= n do
9.                begin
10.                  read(k);
11.                  sum := sum + k;
12.                  i := i + 1;
13.               end
14.            write(sum);
15.            end
16.     else begin
17.            prod := 1;
18.            while i <= n do
19.               begin
20.                  read(k);
21.                  prod := prod * k;
22.                  i := i + 1;
23.               end
24.            write(prod);
25.           end;
26.  end.
```

그림 9 기준 Cs=<i=14, L_{exc}={n}, L_{inc}={sum}>에 대한 선택 슬라이스(반전한 부분)

Lanubile[4]이 향후과제에서도 밝혔듯이 변환 슬라이싱의 취약점은 변환 슬라이싱 알고리즘은 제어 흐름 그래프를 사용하므로 알고리즘의 효율성이 없다는데 있다. 본 논문의 선택 슬라이싱의 장점은 선택슬라이스를 위한 종속그래프를 새롭게 설계하여 그로부터 알고리즘을 설계하였으므로 선형적인 시간내에 슬라이스 계산한 데 있다. 본 연구에서 제시한 종속 그래프를 이용한 알고리즘을 조금 수정하면 변환 슬라이싱에도 효율적인 슬라이스계산이 가능할 것이다.

5. 결론 및 향후과제

새로운 슬라이싱 기법으로 고전적 슬라이싱의 변형인 선택 슬라이스를 정의하고 그 계산 방법을 알고리즘으로 제시하였다. 고전적 슬라이싱 기준에 하나의 파라미터, L_{exc}를 추가함으로서 관련된 것을 포함시키는 것 뿐만아니라 제외까지도 선택한다는 개념을 이용함으로써 크기가 작고 함수적으로 응집력이 있는 슬라이스의 생성에 초점을 맞춘다. 선택 슬라이스의 기준인 <i, L_{exc}, L_{inc}>을 이용하여 문장 i의 위치에서 L_{inc}에 영향을 미치는 부분을 포함하며 L_{exc}에 영향을 미치는 부분을 제외하는 프로그래 코드를 추출하는 것이 선택 슬라이싱이다. 다시 말하면 L_{inc}에 있는 변수들로 시작하여, 그 변수들을 이용하여 L_{exc}에 이르는 계산과정을 추출해내는 과정이다. 현재 개발중인 선택 슬라이싱은 프로그램의 이해에 일차적인 목적을 두지만, 향후 보다 구체적인 활용에 관한 연구가 추가될 것이다. 변환 슬라이싱에서 시도하고 있듯이 재사용가능한 함수를 추출해내는 데에 선택 슬라이싱을 응용시킬 수 있을 것으로 본다.

선택 슬라이싱은 현재 노후시스템의 주종을 이루고 있는 구조적 언어인 C를 그 대상으로 한다. 객체지향 언어에 대한 슬라이싱 기법이 1996년 이후 꾸준히 연구되어지고 있다[12, 18]. 선택 슬라이싱의 객체지향언어에로의 적용도 연구되어져야 할 것이다.

참 고 문 헌

[1] B. P. Leintz and E. F. Swanson, *Software Maintenance Management*, Addison-Wesley, 1980.

[2] Mark Weiser, Program Slicing, *IEEE Transactions on Software Engineering*, 10-4, July 1984

[3] K.B. Gallagher and J.R, Lyle, Using Program Slices in Maintenance, *IEEE Transactions on Software Engineering*, 17-8, August, 1991.

[4] F. Lanubile and G. Visaggio, Extracting Reusable Functions by Program Slicing, *IEEE Transactions on Software Engineering*, vol23. no4. 1997. 4.

[5] P.A. Hausler, M.G. Pleszkoch, R. Linger and A.R. Hevner, Using Function Abstraction to Understand Program Behavior, *IEEE Software*, January, 1990.

[6] F. Cutillo, F. Lanubile and G. Vissaggio, Extracting Application Domain Functions From Old Code: A Real Experience, Proceedings, *Second Workshop on Program Comprehension*, IEEE, Capri, July, 1993

[7] J Jiang, X Zhou, and D Robson, Program Slicing For C - The Problems In Implementation, *11th International Conference on Software Engineering*, IEEE Computer Society Press, 1991

[8] W. Kozaczynski and J. Ning, SRE: A Knowledge-based Environment for Large-scale Software Re-engineering Activities, *Proceedings of Eleventh International Conference on Software Engineering*, IEEE, Pittsburgh, 1989

[9] A Cimitile, A. De Lucia, and M. Munro, Identifying Reusable Functions Using Specification Driiven Program Slicing: A Case Study, *Proceedings of IEEE Internation Conference on Software Mainteance*, Nice, France, Oct. 1995, IEEE Comp. Soc. Press, pp.124-133

[10] D. W. Binkley, and K. B. Gallagher, Program Slicing, *Advances in Computers*, 1996

[11] S. Horwitz, T. Reps, and D. Binkley, Interprocedural Slicing Using Dependence Graphs, *ACM Transaction on Programming Languages and Systems*, vol 12. no. 1, 1990. 1

[12] L. Larsen, and M. J. Harrold, Slicing Object-oriented Software, *Proceedings of the 1996 International Conference on Software Engineering*, Berlin, March, 1996

[13] J. R. Lyle, and M. D. Weiser, Automatic Program Bug Location by Program Slicing, *Proceedings of the 2nd International Conference on Computers and Applications*, Peking, China, June 1987,

[14] H. Agrawal, and J. R. Horgan, Dynamic Program Slicing, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June, 1990

[15] B. Korel, and J. Laski, Dynamic Program Slicing, *Information Processing Letters*, vol.29, no.3, October 1988.

[16] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca, "Software Salvaging based on Conditions," *Proceedings of International Conference on Software Maintenance*, Victoria, Canada, September 1994

[17] J. Ferrante, K. Ottenstein, and J. Warren, The Program Dependence Graph and its Use in Optimization, *ACM Transaction on Programming*

Languages and Systems, vol.9, no.3, 1987
[18] Slicing Multi-threaded Java Programs: A Case Study, Technical Report KSU CIS 99-7, Kansas State University, 1999
[19] A. Lucia, A. Fasolino, and M. Munro, Understanding Function Behaviors through Program Slicing, *Proceedings of 4th IEEE Workshop on Program Comprehension,* Berlin, Germany, March 1996
[20] D. Jacson and J. Eugene, Abstraction Mechanisms for Pictorial Slicing, *Proc. Workshop on Program Comprehension*, Washington, DC, Nov. 1994
[21] F. Tip, A Survey of Program Slicing Techniques, *Journal of Programming Languages*, 3(3): 121-189, Sep 1995

박 수 희
1986년 서울대학교 독어독문학과(학사). 1989년 서울대학교 계산통계학과(학사). 1991년 University of California, San Diego 전산학과(석사). 1994년 University of California, San Diego 전산학과(박사). 1995년 9월 ~ 현재 동덕여자대학교 컴퓨터학과 부교수. 관심분야는 객체지향방법론, 유지보수, 테스팅, 소프트웨어역공학