PIFG를 이용한 프로그램 슬라이싱 : Jump문을 중심으로

강 원 임† · 박 중 양†† · 박 재 흥†††

요 약

프로그램 슬라이싱은 프로그램의 특정 위치에서 변수들의 값에 영향을 주는 문장을 추출하는 방법이다. 프로그램 슬라이싱의 유용성은 디버깅, 최적화, 프로그램 유지보수, 테스팅, 재사용 부품 추출 그리고 프로그램 이해를 포함하는 다른 응용 분야에 널리 알려져 있다. 본 논문은 Jump문 C언어에서의 -goto, break, continue-을 포함하는 프로그램 슬라이싱에 관한 연구이다. 기존의 슬라이싱 알고리즘들은 Jump문을 포함한 프로그램에 적용했을 때 구문식으로 정확한 슬라이스 생성에 실패한다. 본 논문에서는 기존 방법들의 문제점을 지적하고, 정확하고 수행가능한 슬라이스를 생성하는 효율적인 방법을 제안한다.

Program Slicing Using the PIFG : with emphasis on Jump Statement

Won-Im Kang† · Joong-Yang Park†† · Jae-Heung Park†††

ABSTRACT

Program slicing is a technique to extract the statements which influence the value of a variable at a particular point of the program. It has been proposed that this technique is useful in debugging, optimization, program maintenance, testing, extracting reusable components and other applications including the understanding of the program behavior. This paper aims to address the problem of slicing programs with Jump goto, break, continue in C statements. It was found that previous slicing algorithms do not always generate semantically correct slices when applied to such programs. We, therefore, point out why the previous methods do not work in this more general setting, and describe our efficient solutions that compute more correctly executable slices for such programs.

1. 서 론

슬라이싱은 프로그램의 특정 문장에서 변수의 값에 영향을 주는 문장들을 추출하는 방법이다. 프로그램 슬라이싱은 Mark Weiser[2]에 의해 처음으로 소개되어졌으며, 정적 프로그램 슬라이싱[2,7]과 동적 프로그램 슬라이싱[13,15]이 있다. 동적 프로그램 슬라이싱은 프로그램 수행시 특정 입력값이 주어졌을 때 생성되는 프로그램 경로에 대하여 슬라이싱 기준에 영향을 주는 문장을 추출하는 방법이고, 정적 프로그램 슬라이싱은 프로그램 수행 전에 제어 흐름과 자료 흐름 정보를 기반으로 슬라이싱 기준에 영향을 주는 문장을 추출하는 방법이다. 프로그램 슬라이싱의 응용 분야에는 프로그램 이해[8,19], 유지보수[9], 디버깅[3,9,16], 테스팅[20],

† 준 회 원 : 경상대학교 대학원 전산학과
†† 정 회 원 : 경상대학교 통계학과(정보통신연구센터) 교수
††† 정 회 원 : 경상대학교 컴퓨터과학과(정보통신연구센터) 교수
논문접수 : 1998년 1월 16일, 심사완료 : 1998년 7월 16일

재사용 부품 추출[18], 프로그램 통합[6,8], 프로그램 최적화[4]등이 있다.

[2]는 제어 흐름과 자료 흐름 정보를 이용하여 임의의 문장번호 n과 변수 집합 V의 쌍인 <n, V>가 슬라이싱 기준으로 주어졌을 때, 문장 n이 수행하기 직전까지 V에 있는 변수에 영향을 주는 문장을 추출하는 슬라이싱 방법을 제안하였다. 또한 생성된 슬라이스는 그 자체가 수행 가능한 프로그램이며, 동일한 입력값에 대해 원래 프로그램의 동작과 동일하다고 정의하였다[2]. [1,5,7]은 프로그램 슬라이스는 슬라이싱 기준에 나타난 변수에 직접·간접적으로 영향을 주는 프로그램 상의 모든 문장을 추출하는 것으로 정의하였으며, 프로그램 종속성 그래프(이하 PDG라 함)로 표현된 프로그램에 대하여 PDG의 특정 노드 s에 추이적 흐름 종속성(transitive flow dependence) 관계와 추이적 제어 종속성(transitive control dependence) 관계가 존재하는 노드들을 추출하여 슬라이스를 생성하는 방법을 제안하였다.

기존의 슬라이싱 알고리즘들은 비구조화 프로그램, 즉 C 언어에서의 goto, continue, break와 같은 Jump문이 존재하는 프로그램에 대한 정확한 슬라이스 생성에 부적합하다[10,11,12,13]. [2]의 방법을 적용할 경우, Jump문에서는 정의되거나 참조된 변수의 정보를 추출할 수 없으므로 필요한 Jump문 식별에 실패한다. PDG를 이용한 슬라이싱도 마찬가지로 Jump문에 대응되는 노드로부터 나가는 흐름 종속성 에지(flow dependence edge)나 제어 종속성 에지(control dependence edge)가 존재하지 않으므로 필요한 Jump문 식별에 실패한다[10].

본 논문에서는 Jump문이 존재하는 비구조화 프로그램을 중심으로 프로그램 정보 흐름 그래프(Program Information Flow Graph:이하 PIFG라 함)를 이용한 정적 슬라이싱 방법을 제안한다. PIFG는 프로그램의 제어 흐름 그래프 상에 제어 종속성 에지와 각 노드에 대응되는 프로그램의 문장에서 정의되거나 참조되어진 변수들을 각각 집합으로 나타낸 그래프이다. PIFG를 이용해 생성된 슬라이스는 원래 프로그램의 부분 집합으로 기존의 방법으로 생성된 슬라이스들 보다 더 정확하고 작다. 슬라이싱 대상 프로그램의 구성 요소는 입력·출력문, 스칼라 변수와 상수를 지닌 수식문과 배정문, Jump(goto, continue, break)문, While 문, If 문으로 C언어와 유사하게 구성되어있으며, 논의를 간단하게 하기 위해 함수 호출 관계가 없는 단일한 프로그램으로 제한한다.

```
1    if(p1) {
2        s1;
3        if(p2) {
4            s2;
5            goto k; }
6    }
7    else s3;
8    s4;
9    s5
10 k : s6;
```

(그림 1) goto문이 존재하는 프로그램 예제
(Fig. 1) Example of program with goto statemen

```
1  if(p1) {
3     if(p2) {
4         s2;}
6  }
7  else s3;
9  s5
```

(그림 2) 부정확한 슬라이스
(Fig. 2) incorrect slice of fig1

```
1    if(p1) {
3       if(p2) {
4           s2;
5           goto k; }
8    }
7    else s3;
9    s5
10  k : s6;
```

(그림 3) goto 문을 식별한 슬라이스
(Fig. 3) Slice of fig1 identify goto statement

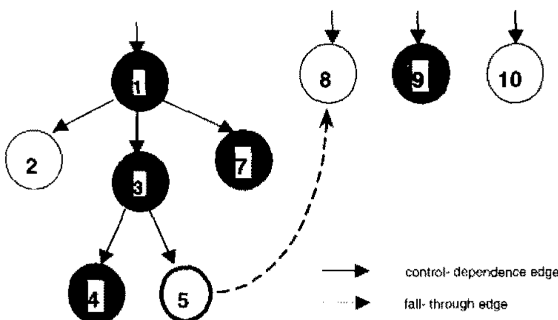
본 논문의 구성은 다음과 같다. 2장은 Jump문이 존재하는 프로그램에 대한 기존의 슬라이싱 관련 연구들을 분석하고 문제점을 지적한다. 3장은 본 논문에서 제안하는 슬라이싱에 기반이 되는 PIFG를 정의한다. 4장은 본 논문에서 제안하는 알고리즘에 기반이 되는 기존의 슬라이싱 방법을 기술하고, PIFG를 이용한 프로그램 슬라이싱 방법을 제안하며, 5장은 결론 및 향후 연구 과제이다.

2. 연구 배경

다음 (그림 1)과 같은 프로그램 코드를 고려해보자. 슬라이싱 기준이 c=<n, V> (단, n>10)로 주어졌을 때 슬라이싱의 수행 결과가 (그림 2)와 같이 {1,3,4,7,9}인

경우, 슬라이스의 동작은 원래 프로그램의 동작과 일치하지 않는다. (그림 1)에서는 4번 문장에서 9번 문장으로의 제어흐름이 발생하지 않는데 반해 (그림 2)에서는 4번에서 9번으로의 제어 흐름이 발생함으로써 부정확한 슬라이스를 생성하게 된다. 이는 기존의 슬라이싱 방법들이 Jump문의 존재를 고려하지 않았기 때문이다. (그림 3)처럼 (그림 2)에서 식별에 실패한 goto문과 goto문의 대응레이블문인 10번 문장을 식별해도, 10번 문장이 원래 슬라이싱 기준 $c=<n, V>$에 나타난 V변수의 값에 영향을 주지 않는 문장이라면 10번 문장은 식별될 필요가 없다.

Jump문이 존재하는 프로그램에 대한 기존의 슬라이싱 방법들은 개념적으로 대부분 동일하다. 그러나 기존의 방법들로 생성된 슬라이스는 원래 프로그램의 행위와 일치하지 않거나 더 큰 슬라이스를 생성한다. [11]은 Jump문이 존재하는 프로그램에 대하여 두 가지 슬라이싱 방법을 제안하였다. 첫 번째는 Jump문에 대응되는 PDG의 노드와 Jump문을 제거할 경우 실행 제어가 도달되는 노드간에 폴-쓰루(fall-through)에지를 추가하여 PDG의 변형된 그래프인 APDG(Augumented Program Dependence Graph)로 표현하여 Jump문을 식별하는 방법이다. 그러나 실제 슬라이싱 기준에 나타난 변수에 영향을 주지 않는 문장을 포함하여 큰 슬라이스를 생성하며, 또한 생성된 슬라이스는 원래 프로그램의 행위와 일치하지 않는 경우도 있다.



(그림 4) (그림 1)에 대한 APDG
(Fig. 4) APDG of fig1

예를 들어 (그림 1)을 [11]에서 제안한 APDG로 표현하면 (그림 4)와 같다. 제어 종속성 에지 관계만 나타낸 전체 APDG의 일부분이다. Goto문에서는 어떤 변수가 정의되거나 참조되지 않으므로 필요한 goto문 식별을 위해 APDG 상에 흐름 종속성 에지는 나타내지 않았다. (그림 4)의 그래프에서 보는 바와 같이 5번 노드와 폴-쓰루 에지 관계를 지니는 8번 노드가 슬라이스에 포함되지 않음으로써 5번 노드에 대응되는 goto문은 슬라이스에 포함되지 않는다. 즉, 필요한 goto문 식별에 실패한다.

두 번째는 기존의 PDG를 이용하였으며, 첫 번째 방법보다는 더 작은 슬라이스를 생성하였으나, 슬라이스의 수행 동작이 원래 프로그램과 일치함을 보이기 위해 슬라이싱 기준에 나타난 변수에 영향을 주지 않는 원래 프로그램의 문장을 goto문으로 치환하여 슬라이

```
1:        sum = 0;
2:        positives = 0;
3: L3:    if (eof()) goto L14;
4:        read(x);
5:        if (x>0) goto L8;
6:        sum = sum + x;
7:        goto L3;
8: L8:    positives = positives + 1;
9:        if (x%2 != 0) goto L12;
10:       sum = sum + x;
11:       goto L3;
12: L12:  sum = sum + x;
13:       goto L3;
14: L14:  write(sum);
15:       write(positives);
```

(그림 5) goto 문이 존재하는 프로그램
(Fig. 5)program with goto st.

```
2:       positives = 0;
3: L3:   if (eof()) goto L14;
4:       read(x);
5:       if (x>0) goto L8;
7        goto L13;
8: L8:   positives = positives + 1;
```

(그림 6) 부정확한 슬라이스
(Fig. 6) incorrect slice

```
2:       positives = 0;
3: L3:   if (eof()) goto L14;
4:       read(x);
5:         if (x>0) goto L8;
7          goto L13;
8: L8:   positives = positives + 1;
9:         if (x%2!=0) goto L12;
11:        goto L3;
   L12:
13:        goto L3;
   L14:
15:        write(positives);
```

(그림 7)정확한 슬라이스
(Fig. 7) correct slice

스에 포함시킨다. 따라서 생성된 슬라이스는 많은 goto문을 지니며, 원래 프로그램의 부분집합이 아니다.

[13]은 렉시칼 후속자 트리(Lexical successor tree)와 제어 종속성 그래프 그리고 포스트도미네이터 트리(Postdominator tree)를 이용하여 슬라이스에 필요한 Jump문을 식별하는 방법을 제안하였으나, [11]의 첫 번째 방법과 유사하게 슬라이싱 기준에 나타난 변수에 영향을 주지 않는 불필요한 문장을 포함한다. (그림 5)[13]의 goto문이 포함된 프로그램에 대하여 슬라이싱 기준이 c=<15, positives>로 주어졌을 때, 기존의 Jump문이 없는 슬라이싱 알고리즘으로 생성된 슬라이스는 (그림 6)과 같다. 이때 7번 goto문을 식별하지 못하여 생성된 슬라이스와 원래 프로그램의 동작은 일치하지 않으며, 동일한 입력에 대하여 15번 문장에서의 positives 값도 달라진다. [13]의 알고리즘에 의해 생성된 슬라이스는 (그림 7)과 같다. 이 경우도 정확한 슬라이스는 생성하였으나, 실제 슬라이싱 기준에 나타난 positives 변수에 영향을 주지 않는 9번 문장을 포함한다. [13]의 알고리즘은 슬라이스에 포함된 Jump문 노드와 제어 종속성 관계를 지닌 제어문도 슬라이스에 포함한다. 따라서 11번 goto문이 포함됨으로써 11번 goto문의 제어문인 9번 문장이 포함되었다.

(그림 8)에 대하여 슬라이싱 기준 c=<9, y>로 주어졌을 때 [13]의 방법에 의해 생성된 슬라이스는 (그림 9)와 같다. 그러나 실제 9번 문장에 나타난 y 변수에 영향을 주는 문장은 3, 9번이다. Jump문이 존재하는 프로그램에 대한 기존의 알고리즘들은 두 단계로 슬라이스를 생성한다. 먼저 필요한 Jump문을 식별하기 전에 기존의 Jump문이 없는 프로그램에 대한 슬라이싱 알고리즘으로 슬라이싱 기준에 영향을 주는 문장을 추출한 후, 두 번째로 슬라이스에 필요한 Jump문을 식별한다.

(그림 8)의 경우 첫 번째 단계에서 2번 goto문을 고려하지 않으면, 1번 문장은 슬라이스에 포함된 3번 문장의 문맥상 제어문이므로 1번 문장이 슬라이스에 포함된다. 그러나 실제 제어 흐름 그래프상에서 1번과 3번 문장은 제어 종속성 관계를 지니지 않는다. [13]의 알고리즘은 첫 번째 단계에서 Jump문을 포함한 제어 종속성 관계를 고려하지 않았기 때문에 큰 슬라이스를 생성한다.

[21]도 goto문을 해결하려했으나 goto문의 레이블이

```
1:          if (C1) {
2:              goto L6;
3: L3:          y = ...;
4:              goto L8;
  }
5:          z = ...;
6: L6:      x = ...;
7:          goto L3;
8: L8:      write(x);
9:          write(y);
10:         write(z);
```

(그림 8) goto문이 존재하는 프로그램
(Fig. 8) program with goto st.

```
1:        if (C1) {
2:              goto L6;
3: L3:          y = ...;
4:              goto ...;
                }
   L6:

7:        goto L3;
   L8:
9:              write(y);
```

(그림 9) 부정확한 슬라이스
(Fig. 9) incorrect Slice

```
3: L3:    y = ...;
9:              write(y);
```

(그림 10) 정확한 슬라이스
(Fig. 10) correct slice

붙은 문장이 슬라이스에 포함된 경우만 goto문을 포함함으로써 슬라이스에 필요한 goto문 식별에 실패한다. [12]은 필요한 goto문이 어떤 제어문과 그 제어문의 최초의 포스트도미네이터 사이에 존재하고 그 사이에 어떤 노드도 슬라이스에 포함되지 않은 경우, 필요한 goto문을 식별하지 못하여 정확한 슬라이스 생성에 실패한다. 즉, (그림 5)에서 11번이나 13번 goto문의 제어문인 9번 문장이 슬라이스에 포함되지 않으므로 필요한 goto문을 정확하게 식별하지 못한다. [10]도 임의의 제어문을 지닌 프로그램에 대한 슬라이싱 방법을 제안하였으나 또한 슬라이스에 필요한 Jump문을 식별하지 못하여 정확한 슬라이스를 생성에 실패한다.

3. 프로그램 정보 흐름 그래프(Program Information Flow Graph)

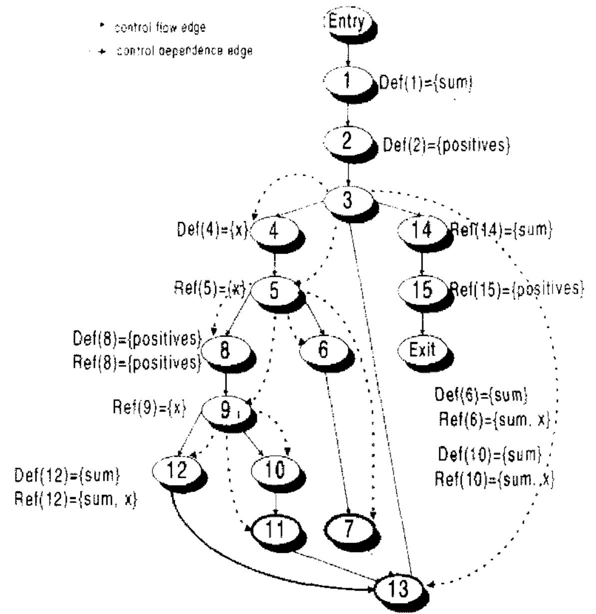
기존의 제어 흐름 그래프 상에 자료 정보와 제어 종속성 정보를 추가한 프로그램 정보 흐름 그래프를

표기한다. 프로그램의 각 문장을 노드로 표현하고 각 노드에 Def, Ref 정보를 표현한다. 또한 문장의 제어 흐름과 제어 종속성을 각각 다른 종류의 에지로 표현한다. PIFG 정의에 기반이 되는 그래프들과 용어에 대한 정의는 다음과 같다.

정의3.1 : 흐름 그래프(Flowgraph) G는 $G=<N,E,n_0>$로 표현한다. N은 유한개의 노드의 집합이고 E는 $N \times N$ $\{(n_i, n_j)|n_i, n_j \in N\}$로 노드들의 순서쌍인 에지의 집합이며 n_0는 초기 노드이다. 에지 $(n_i, n_j) \in E$에 대하여 n_i는 n_j의 선행자(predecessor)이며, n_j는 n_i의 후행자(successor)이다. Pred(n)은 n의 선행자들의 집합이고, Succ(n)은 n의 후행자들의 집합이다. 그래프의 경로 W는 $n_0 n_1 n_2 \ldots n_k$ ($k \geq 1$, (n_i, n_{i+1}) $i=1,2,\cdot\cdot,k-1$)인 노드들의 순열로써 프로그램 실행 제어는 n_i후에 n_{i+1}에 도달한다.

정의3.2 : 해먹 그래프(Hammock graph) G는 $G=<N, E, n_{Initial}, n_{Final}>$로 표현되며, 흐름 그래프 $<N,E,n_{Initial}>$와 $<N,E^{-1},n_{Final}>$의 속성을 동시에 지닌다. $E^{-1}=\{(n_j, n_i)|(n_i, n_j) \in E\}$를 의미한다. 초기($n_{Initial}$) 노드와 최종($n_{Final}$) 노드는 프로그램의 진입점(Entry point)과 진출점(Exit point)를 나타낸다. $n_{Initial}$에서 $n(\in N)$으로 가는 모든 경로 상에 노드 $m(\in N)$이 존재한다면 m을 n의 도미네이터(dominator)라 하고 DOM(n)으로 나타낸다. DOM(n) 중에서 제일 첫 번째 나타나는 노드를 즉시 도미네이터(Immediate dominator)라고 하며 IDOM(n)으로 나타낸다. n에서 n_{Final}로 가는 모든 경로 상에 노드 m이 존재한다면 m을 n의 포스트도미네이터(postdominator)라 하고 PDOM(n)으로 나타낸다. PDOM(n) 중에서 제일 첫 번째 나타나는 노드를 즉시 포스트도미네이터(Immediate postdominator)라고 하며 IPDOM(n)으로 나타낸다. IMS(n)은 노드 n 바로 직후에 나타나는 후행자들의 집합으로 즉시 후행자(Immediate successor)이다. ND(n)은 n에서 IPDOM(n)까지의 경로 상에 나타나는, n과 IPDOM(n)을 제외한 모든 노드들의 집합이다. 만약 $|IMS(n)| \leq 1$라면 $ND(n)=\emptyset$이 된다. IMP(n)은 문장 n 바로 직전에 나타나는 선행자들의 집합으로 즉시 선행자(Immediate predecessor)이다.

정의3.3 : 제어 흐름 그래프(Control flow graph) G는 $G=<N, E, Entry, Exit>$로 표현되는 해먹 그래프이다. 각 노드는 프로그램의 한 문장에 대응되며, 각 에지 (n_i, n_j)들은 프로그램의 문장들간의 제어 흐름을 표현하는 제어 흐름 에지(control flow edge)로써 $n_i \rightarrow_{cf} n_j$로 표기한다. Entry와 Exit는 프로그램의 진입점과 진출점을 나타내는 노드들이다.




```
1:        sum=0;
2:        positives = 0;
3: L3:    if (eof()) goto L14;
4:        read(x);
5:        if (x>0) goto L8;
6:        sum = sum + x;
7:        goto L13;
8: L8:    positives = positives + 1;
9:        If (x%2 != 0) goto L12;
10:       sum = sum + x;
11:       goto L13;
12: L12:  sum = sum + x;
13: L13:  goto L3;
14: L14:  write(sum);
15:       write(positives);
```

(그림 11) goto문을 지닌 프로그램과 대응 PIFG
(Fig. 11) program with goto st. and its PIFG

PIFG는 제어 흐름 그래프의 속성을 모두 지니면서 제어 종속성 정보가 제어 흐름 정보와는 다른 종류의 에지로 표현되며, 각 노드에는 대응되는 프로그램의 문장에서 정의되고 참조되어진 변수들의 정보를 나타낸 그래프이다. PIFG에 추가된 정보들에 대한 정의는 다음과 같다.

정의3.4 : 한 노드 n에 대한 Def(n)과 Ref(n)은 대응되는 프로그램 문장에서 각각 정의되고 참조되는 변수들의 집합을 나타낸다.

정의3.5 : 노드 n에서 노드 m으로의 제어 종속성

에지 관계를 $n \rightarrow_{cd} m$으로 표현하며 다음의 조건을 만족한다.

(1) n에서 m으로 가는 경로 W가 존재하며, W상에서 n을 제외한 다른 노드와 제어 종속성 관계를 지나지 않는 임의의 노드 n'에 대하여, $m \in PDOM(n')$이다.

(2) $m \notin PDOM(n)$ 즉, m은 n의 포스트도미네이터가 아니다.

본 논문에서 사용하는 제어 종속성 관계는 제어문 식별을 위해서이다. 따라서 PDG에서 나타내는 Entry 노드와의 제어 종속성 관계는 나타내지 않았다.

정의3.6 : PIFG는 $\Theta = \langle N,E,C,\Sigma,D,U \rangle$로 표현한다. 각 노드는 배정문, 입출력문, 조건문, Jump문, 반복문으로 이루어진 프로그램의 한 문장에 대응된다. PIFG는 노드에 Def, Ref 정보가 추가된 변형된 제어 흐름 그래프이다. N은 제어 흐름 그래프의 노드를 포함하며, E는 제어 흐름 그래프의 에지를 포함한다. C는 제어 종속성 에지를 나타내며, Σ는 프로그램에 나타난 변수들의 집합이다. D: $N_\Theta \rightarrow \Delta(\Sigma)$, U: $N_\Theta \rightarrow \Delta(\Sigma)$로써, 프로그램의 각 노드를 대응되는 문장에서 정의되고 참조되어진 변수로 매핑해주는 함수이다.

(그림 11)은 goto문이 존재하는 프로그램과 대응 PIFG의 예이다.

4. 프로그램 슬라이싱

자료 정보, 제어 흐름 정보 그리고 제어 종속성 정보를 이용하여 Jump문이 존재하는 프로그램을 제어 흐름 그래프 상에서 쉽게 변형할 수 있는 PIFG로 표현하여 기존의 알고리즘들보다 더 간단한 방법으로 정확한 슬라이스를 생성함을 보인다.

4.1 모듈내 슬라이싱 알고리즘

[Weiser]가 제안한 슬라이스의 정의는 다음과 같다[2].

정의4.1 : 슬라이싱 기준 c=<i, V>는 프로그램의 문장 번호와 프로그램에 나타나는 변수들의 부분 집합의 쌍으로 주어진다. 슬라이싱 기준이 주어졌을 때 i문장이 수행하기 직전까지 V에 있는 변수들에게 영향을 줄 수 있는 문장들을 추출함으로써 생성된 슬라이스는 동일한 입력값에 대해 원래 프로그램의 동작과 동일하며, 독립적으로 수행 가능한 코드이다.

[2]의 정의에 따르면, 슬라이싱 기준 c가 주어졌을 때 i문장이 수행하기 직전까지 V에 있는 변수들에게 영향을 줄 수 있는 n(∈Pred(i)) 문장에 나타난 변수들의 집합을 $RIN^0_c(n)$으로 나타낸다. 즉, $RIN^0_c(n)$에 있는 변수는 i노드 수행 직전까지 V에 있는 변수들에게 직접적으로 영향을 줄 수 있는 변수들의 집합이다. i노드 수행 직전까지 $RIN^0_c(n)$에 있는 변수들의 값이 변경되면, i노드 수행 후 V에 있는 변수들의 값은 원래 프로그램의 값과 달라지게 된다.

슬라이싱 기준 c=<i, V>가 주어졌을 때 $RIN^0_c(n)$을 구하는 공식은 다음과 같다.

$$RIN^0_c(n) = \{v \in V | n = i\} U$$

$$\{Ref(n) | RIN^0_c(IMS(n)) \cap Def(n) \neq \emptyset\} U$$

$$\{RIN^0_c(IMS(n)) - Def(n)\}$$

$RIN^0_c(n)$ 정보를 이용하여 슬라이싱 기준에 나타난 V변수에 직접적으로 영향을 주는 문장을 추출하는 공식은 다음과 같다.

$$S^0_c = \{n | Def(n) \cap RIN^0_c(IMS(n)) \neq \emptyset\}$$

S_c^0에 어떤 제어문에 종속적인 문장이 포함되어졌다면, 그 문장에 대한 제어문도 추출해야하며, 제어문에서 참조된 변수에 영향을 주는 문장도 슬라이스에 포함해야한다. 슬라이스에 포함된 문장에 대한 제어문을 추출하는 기존의 공식은 다음과 같다.

$$B^0_c = \{b | ND(b) \cap S^0_c \neq \emptyset\}$$

그러나 본 논문에서 제어문을 식별하기 위한 방법은 [정의3.5]의 제어 종속성 정보를 사용한다.

$$B^0_c = \{b | b \rightarrow_{cd} A, \exists A \in S^0_c\}$$

첨자 0이 붙은 RIN_c^0은 슬라이싱 기준에 나타난 V의 각 원소들에게 직접적으로 영향을 주는 관련 있는 변수들의 집합이고, S_c^0는 직접적으로 영향을 주는 문장들의 집합이다. 슬라이스는 슬라이싱 기준에 나타난 변수들에게 간접적으로 영향을 주는 문장도 포함해야 하며, 슬라이싱 기준 c에 직접·간접적인 영향을 주는 변수들과 문장을 추출하기 위한 반복적인 방정식은 다음과 같다.

$$RIN^{i+1}_c(n) = RIN^i_c(n) \bigcup_{b \in B^i_c} RIN_{B_c(b)}{}^0(n)$$

$$S^{i+1}_c = \{n | Def(n) \cap RIN^{i+1}_c(IMS(n)) \neq \emptyset \text{ or } n \in B^i_c\}$$

$$B^{i+1}_c = \{b | b \rightarrow_{cd} A, \exists A \in S^{i+1}_c\}$$

$B_c(b)$은 제어문 b의 슬라이싱 기준으로써 <b, Ref(b)>

이다. 슬라이싱 반복 수행은 다음의 조건을 만족하면 종료하게 된다.

$$\forall n \in N,\ RIN_c^{i+1}(n) = RIN_c^{i}(n)$$
$$\text{or}\ \ S_c^{i+1} = S_c^{i}$$

4.2 PIFG를 이용한 모듈내 슬라이싱

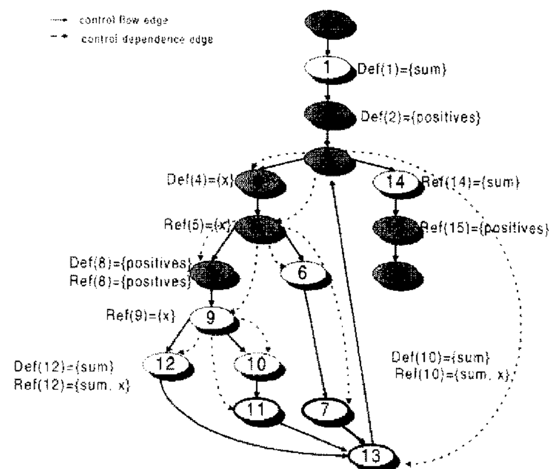
PIFG를 이용하는 프로그램 슬라이싱은 [2]의 방법을 기반으로 하여 Jump문이 존재하는 프로그램에 대한 슬라이싱까지 확장하였다. PIFG를 이용하여 생성된 프로그램 슬라이스 정의는 다음과 같다.

정의4.2 : 프로그램 P에 대하여 슬라이싱 기준 c=<n, V>가 주어졌을 때, S_P(PIFG, c)는 PIFG를 이용하여 생성된 프로그램 P의 슬라이스이다. 프로그램 P의 PIFG에서 슬라이싱 기준에 영향을 주지 않는 노드들을 제거함으로써 S_P의 PIFG를 생성하게 된다. 생성된 슬라이스는 동일한 입력에 대해 P의 행위와 동일한 수행 가능한 프로그램이며, $S_P \subseteq P$이다. 임의 Jump문이 슬라이스에 포함되고 내응 레이블이 붙은 문장 n이 슬라이스에 포함되지 않은 경우 n의 포스트도미네이터 중 슬라이스에 포함된 첫 번째 포스트도미네이터에 레이블을 추가한다.

Jump문이 포함된 프로그램에 대한 기존의 슬라이싱 방법들은 정확하지 않거나 큰 슬라이스를 생성한다. PIFG를 이용한 Jump문이 존재하는 프로그램에 대한 슬라이싱은 기존의 방법들보다 더 정확하고 작은 슬라이스를 생성한다. PIFG를 이용한 Jump문이 존재하는 프로그램에 대한 슬라이싱은 기존의 방법과 유사하게 두 단계로 수행된다. 첫 번째 단계에서 원시 프로그램을 PIFG로 표현한 후, PIFG의 Jump문을 제외한 각 노드에 대해 (그림 12)의 알고리즘을 적용하여 슬라이싱 기준에 영향을 끼치는 노드들을 식별한다. 기존의 알고리즘들은 첫 번째 단계에서 Jump문을 포함한 제어 흐름을 고려하지 않았기 때문에 실제 슬라이스보다 큰 슬라이스를 생성했다. 그 이유는 Jump문이 존재하는 프로그램을 제어 흐름 그래프로 표현했을 경우, Jump문의 특성으로 인해 어떤 문장이 구문적으로는 제어문 내부에 존재하더라도 실제 그 문장이 제어문의 수행 결과에 영향을 받지 않을 수도 있다. 따라서 Jump문이 존재하는 프로그램에 대한 슬라이싱은 첫 번째 단계에서 Jump문을 포함한 제어 종속성 정보를 고려해야한다.

```
Construct PIFG

Procedure MakeSlice1(P, c)
P : Original Program's PIFG
S : Set of nodes in P
c = <n, V> : slicing criterion, n∈S
begin
        CRI={c}
        Call MarkedPIFG(CRI)
end

Procedure MarkedPIFG(CRI)
begin
Mark n
while CRI is not empty
     Select and remove an element c=<n, V> from the CRI
     if n→cd t, t is marked then
          Mark n
          Insert <n, Ref(n)> into CRI
     endif
     for ∀m s.t m→d n, ∃v∈V
     do
          if v∈Def(m)&Ref(m)=∅ then
          Mark m
          V=V-v
          if V≠∅ then
                              Insert <m, V> into CRI
          else if v∉Def(m)  then
          Insert <m, V> into CRI
          else if v∈Def(m)&Ref(m)≠∅ then
          Mark m
          V=(V-v)∪x, ∀x∈Ref(m)
          Insert <m, V> into CRI
          endif
  od
endwhile
end
```

(그림 12) 슬라이싱 알고리즘 : pass-1
(Fig. 12) slicing algorithm : pass-1



(그림 13) (그림 11)에 대해 c=⟨15, positives⟩로 주어졌을 때 (그림 12) 알고리즘 수행 후 MPIFG
(Fig. 13) MPIFG obtained using (fig12) algorithm about (fig11), c=⟨15, positives⟩

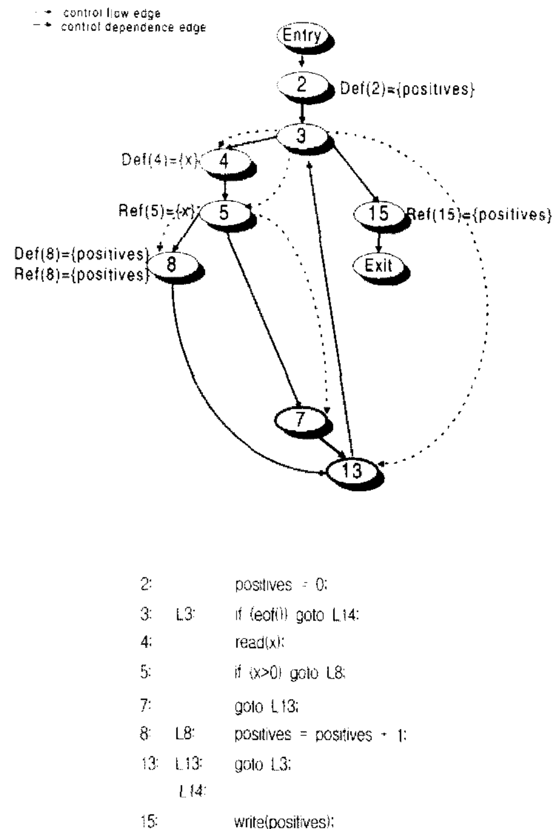
(그림 12)의 알고리즘은 원래 프로그램의 PIFG에 대하여 슬라이싱 기준 c가 주어졌을 때, 슬라이싱 기준에 영향을 주는 노드를 표시한 MPIFG(Marked PIFG)를 생성한다. (그림 11)의 프로그램에 대하여 슬라이싱 기준 c=<15, positives>이 주어졌을 때 생성된 MPIFG는 (그림 13)과 같다.

```
Procedure MakeSlice2(M, c)
begin
M : Original Program's MPIFG produced by MakeSlice1.
S : Set of nodes in P;
Jumpcandidate=∅;
RPIFG=MPIFG;
While ∀m∈Pred(n)&RPIFG s.t. m is not visited
            ReducedPIFG(n);
            Jumpcollect(Jumpcandidate, RPIFG);
endwhile;
For each goto statements. Goto L. in slice, if the statements
labled L is not in slice then associate the label L
with its immediate postdominator in Slice;
return(RPIFG);
end

Procedure ReducedPIFG(n)
begin
for ∀m∈IMP(n) s.t m is not visited
      if m is "Entry" then continue;
      if m is marked then
            ReducedPIFG(m);
      else if m is Jump
            then
               insert m into Jumpcandidate;
            else finding k s.t k is the immediate predecessor
                 among all Pred(n) which are marked by MakeSlice1
                 connect(k, n);
                 remove all nodes k→*cf n excluding k, n;
                     if k is "Entry" then continue;
                     ReducedPIFG(k);
od
end

Procedure Jumpcollect(Jumpcandidate, RPIFG)
begin
while Jumpcandidate≠∅
   select and remove n from Jumpcandidate;
   if ∃m→*cf n s.t m is the immediate predecessor among all
      Pred(n) which are marked by MakeSlice1
   then if m→*cf n has no Jump except n
        then Mark n
             connect(m, n);
             remove all nodes m→*cf n excluding m and n;
             ReducedPIFG(m);
        else insert Jump into Jumpcandidate;
        endif;
   else remove all nodes ∀k→*cf n;
   endif;
od
end
```

(그림 14) 슬라이싱 알고리즘 : pass-2
(Fig. 14) Slicing algorithm : pass-2

두 번째 단계에서는 (그림 12)의 알고리즘에 의해 생성된 MPIFG에 대하여, 표시되지 않은 노드들과 슬라이스에 불필요한 Jump문에 대응되는 노드들, 그리고 그 노드들에 연결된 에지를 제거함으로써 RPIFG (Reduced PIFG)를 생성한다. RPIFG는 독립적으로 수행 가능한 슬라이스 S_P(PIFG, c)의 PIFG이며, 동일한 입력에 대해 원래 프로그램과 행위가 동일하다.

(그림 14)는 두 번째 단계에 대한 알고리즘이다. 첫 번째 단계로부터 생성된 MPIFG를 두 번째 단계 시작 전에 초기 RPIFG로 두고, 슬라이싱 기준에 나타난 문번호 n부터 시작하여 n의 모든 선행자들 중 방문되지 않은 노드 m에 대하여 m이 Jump문이면 Jumpcandidate 집합에 노드 번호를 추가한다. Jump문이 아니면서 슬라이스에 포함되지 않은 노드일 경우, MPIFG상의 $l \rightarrow^{*}_{cf} m$ 관계를 이루는 노드들 중 최초로 슬라이스에 포함된 노드 l을 만나게 되면 connect(l, n)연산에 의해 $l \rightarrow_{cf} m$ 관계를 생성한다. 동 시에 l과 n을 제외한 $l \rightarrow^{*}_{cf} n$ 사이에 존재하는 모든 노드들을 MPIFG상에서 제거한다. 제거되는 노드들의 들어오는 에지와 나가는



(그림 15) 정확한 슬라이스와 대응 RPIFG
(Fig. 15) correct slice and its RPIFG

```
1: sum = 0;
2: positives = 0;
3: while (!eof()) {
4:    read(x);
5:    if (x <= 0) {
6:          sum = sum + f1(x);
7:          continue; }
8: positives = positives + 1;
9: if (x%2 != 0) {
10:         sum = sum + f2(x);
11:    continue; }
12: sum = sum + f3(x);     }
13: write(sum);
14: write(positives);
```

(그림 16) continue문이 존재하는 프로그램
(Fig. 16) progrma with continue st.

```
2: positives = 0;
3: while (!eof()) {
4:    read(x);
5:    if (x <= 0) {
7:             continue; }
8: positives = positives + 1;
14: write(positives);
```

(그림 17) 정확한 슬라이스
(Fig. 17) correct slice

에지들도 제거한다.

Jumpcandidate 집합에 존재하는 각 Jump문에 대응되는 노드 n에 대하여, Pass-2에서 아직 방문되지 않은 노드들 중 $m \rightarrow_{cf}^{*} n$ 관계를 이루는 노드 m에 대하여 슬라이스에 포함된 노드 중에서 n의 즉시 선행자라면 connect(m, n)을 수행하고, m과 n을 제외한 $m \rightarrow_{cf}^{*} n$ 사이에 존재하는 모든 노드들과 에지들을 MPIFG 상에서 제거한다. 조건을 만족하는 m이 존재하지 않는다면, $l \rightarrow_{cf}^{*} n$ 관계를 이루는 l과 n을 포함한 모든 노드와 n으로부터 나가는 에지를 제거한다.

(그림 13)에 대하여 (그림 14)의 알고리즘을 적용하여 생성된 RPIFG와 대응 슬라이스 S_P(PIFG, c)는 다음 (그림 15)와 같다. 본 논문에서 제안하는 Jump문이 포함된 프로그램에 대한 슬라이싱 알고리즘은 기존의 방법들보다 더 정확하고 작은 슬라이스를 생성하게 된다.

Continue문과 break문은 goto문의 특별한 경우로써, continue문은 자신의 제어문으로 제어를 옮겨주고, break문은 자신과 제어 종속성 관계를 지니는 제어문 n의 IPDOM(n)으로 제어를 옮겨준다. 본 논문에서 제안하는 방법은 어떤 종류의 Jump문에도 잘 적용된다. Continue문이 존재하는 (그림 16)의 프로그램에서 슬라이싱 기준이 c=<14, positives>로 주어졌을 때 생성된 슬라이스는 (그림 17)이며, [12]에 의한 방법으로 생성된 슬라이스보다 작은 슬라이스이다. swich-case문 내에 존재하는 문장이 슬라이스에 포함되어진 경우 생성된 슬라이스는 모든 break문이 포함되어진다.

본 논문에서 제안하는 방법은 Jump-goto, continue, break-문이 존재하는 비구조화 프로그램에 대하여 일관적인 방법으로 슬라이스를 생성하며, 기존의 슬라이싱 방법들보다 더 정확하고 작은 슬라이스를 생성한다.

5. 결론 및 향후 연구 과제

슬라이싱은 프로그램의 특정 계산에 관련 있는 문장 추출에 기반을 둔 프로그램 분해 방법이다. 본 논문은 Jump-goto, break, continue-문이 존재하는 프로그램에 대한 기존의 슬라이싱 방법들이 정확하지 않기나 큰 슬라이스를 생성함을 보였고, 기존의 방법들보다 간단하면서 더 정확하고 더 작은 실행가능한 슬라이스 생성을 위한 방법을 제안하였다. 이 방법은 Jump문이 존재하는 프로그램에 대한 정확한 슬라이스 생성을 위해 제어 흐름 그래프상에 제어 종속성 정보와 각 노드에 자료 정보를 표현한 프로그램 정보 흐름 그래프를 기반으로 하였으며, PIFG로 표현된 프로그램에 대한 슬라이스는 기존의 방법들보다 더 정확하고 크기가 작은 슬라이스를 생성함을 보였다.

현재 포인터 변수와 배열이 존재하는 프로그램에 대한 정확한 슬라이싱 문제와 모듈간 슬라이싱 문제를 해결 중에 있으며, 자동화된 C언어에 대한 슬라이서 도구 개발이 향후 연구 과제이다. 생성된 슬라이스들은 프로그램 이해, 유지보수, 디버깅, 테스팅, 재사용 부품 추출 등에 유용하게 이용되어 소프트웨어 개발에 드는 비용을 크게 줄일 수 있고, 나아가 품질 향상도 기대될 것으로 전망한다.

참 고 문 헌

[1] OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in a software development environment. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments(Pittsburgh, Pa., April 23-25,1984). ACM SIGPLAN Not. 19, 5(May 1984), pp.177

184.

[2] Mark Weiser, Program Slicing, IEEE Trans. Software Eng., Vol.SE-10, No.4, July 1985.

[3] LYLE,J., AND WEISER,M. Experiments on slicing based debugging tools. In proceedings of the First Conference on Empirical Studies of Programming(June 1986).

[4] FERRANTE,J.,OTTENSTEIN,K.,and WARREN,J. The program dependence graph and its use in optimization.ACM Trans.Program.Lang.Syst.9, 3 (July 1987), pp.319-349.

[5] Reps, T., AND YANG, W. The semantics of program slicing. TR-777, Computer Sciences Dept., univ. of Wisconsin, Madison, june 1988.

[6] T.Reps and W.Yang, The semantics of program slicing and program integration, in Proceedings of the Colloquium on current Issues in Programming Languages, (Barcelona, Spain, March pp.13-17, 1989), Lecture Notes in Computer Science, Springer-Verlag, New York, NY(March 1989).

[7] SUSAN HORWITZ, THOMAS REPS, and DAVIDBINKLEY, Interprocedural Slicing using Dependence Graph, ACM Trans. Programming Languages and Systems, Vol.12, No.1, January 1990.

[8] S.Horwitz, Identifying the semantic and textual differences between two versions of a program, Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation(published as SIGPLAN Notices) 25(6) pp.234-245 ACM,(June 20-22, 1990).

[9] Keith Brian Gallagher, and James R. Lyle, Using Program Slicing in Software Maintenance, IEEE Trans. Software Eng., Vol.SE-17, No.8, august 1991.

[10] THOMAS BALL and SUSAN HORWITZ. Slicing Programs with Arbitrary Control Flow. proc. of the 1st International Workshop on Automated and Algorithmic Debugging: LNCS Vol.749, 1993.

[11] Jong-Deok CHOI and Jeanne Ferrante, Static Slicing in the Presence of Goto statements, ACM Transactions of Programming Languages and Systems, Vol.16, No.4, July 1994, pp.1097-1113

[12] Jingyue Jiang, Xiling Zhou, David Robson, Program Slicing For C-The Problems in Implementation Proc. IEEE International Conf. Software Maintenance, pp.182-190, 1991.

[13] Hiralal Agrawal, On Slicing Programs with Jump statements, ACM SIGPLAN 94-6, pp.302-312, 1994.

[14] B. Korel and J. Laski, Dynamic Program Slicing, Information processing letters, Vol.29, No.3, Oct. 1988.

[15] B. Korel, Comcputation of Dynamic Program Slices for unstructured programs, IEEE Transactions on Software Engineering, Vol.23, No.1, Januart 1997.

[16] Hiralal Agrawal, Towards Automatic Debugging of Computer Programs, Ph.D, 1991.

[17] Thomas Jaudon Ball, The Use of control Flow and Control Dependence in Software Tools, Ph.D, dissertation, 1993.

[18] Andrea De Lucia, Identifying Reusable Functions in Code Using Specification Driven Techniques, Ph.D, dissertation, 1995.

[19] Horwits, S., Prins, J., and Reps, T., Identifying non-interfering versions of Programs, ACM Transactions on Programming Language and Systems, 11(3), pp.345-387, July 1989.

[20] Hiralal Agrawal, Incremental Regression Testing, In Proceedings of the IEEE Conference on Software Maintenance, pp.348-357, Sept. 1993.

[21] K. B. Gallagher, Using Program Slicing in Software Maintenance, Ph.D. dissertation, Univ. Michigan, Ann arbor, MI, 1989

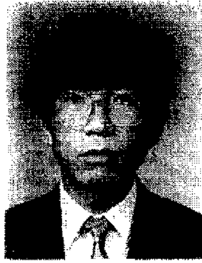
강 원 임

1991년 경상대학교 전산통계학과 졸업(학사)

1995년 경상대학교 대학원 전산학과 (공학석사)

1997년 경상대학교 대학원 전산학과 (박사 수료)

관심분야 : 소프트웨어 공학, 퍼지 정보 검색, 프로그램 분석, 테스팅



박 중 양

1982년 연세대학교 응용통계학과 졸업(학사)

1984년 한국과학기술원 산업공학과 응용통계전공(석사)

1994년 한국과학기술원 산업공학과 응용통계전공(박사)

1984년~현재 경상대학교 통계학과 교수

관심분야 : 소프트웨어 신뢰성, 신경망, 선형통계모형, 실험계획법

박 재 흥

1978년 충북대 수학교육과 졸업(학사)

1980년 중앙대 대학원 전산과(석사)

1989년 중앙대 대학원 전산과(박사)

1983년~현재 경상대 컴퓨터과학과 교수

관심분야 : 소프트웨어 공학, 테스팅, 소프트웨어 신뢰성