# 함수 호출이 존재하는 프로그램 슬라이싱에 관한 연구

홍중기* · 강원임 · 박재흥
경상대학교 전자계산학과

# A Study for Program Slicing in the presence of Function Calls

Joong-Ki Hong · Won-Im Kang · Jae-Heung Park
Dept of Computer Science, Gyeongsang National University

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 C언어에서의 함수 호출이 존재하는 모듈간 프로그램 슬라이싱에 관한 연구이다. 기존의 모듈간 슬라이싱 알고리즘들은 함수의 호출 문맥 설명에 실패하거나 문장 기반 그래프를 사용함으로써 정확한 슬라이스 생성에 실패한다. 본 논문에서는 기존 방법들의 문제점을 지적하고, 시스템 정보 흐름 그래프(System Information Flow Graph)를 이용하여 정확하고 수행 가능한 모듈간 슬라이스 생성 방법을 제안한다.

## 1. 서론

프로그램 분해 방법 중의 하나인 프로그램 슬라이싱은 프로그램의 특정 문장에서 관심있는 변수들의 값에 영향을 주는 문장들을 추출하는 방법이다. 프로그램 슬라이싱은 [Mark Weiser][1]에 의해 처음으로 소개되어졌으며, 정적 프로그램 슬라이싱과 동적 프로그램 슬라이싱이 있다. 프로그램 슬라이싱의 응용 분야에는 프로그램 이해[5], 유지보수[6], 디버깅[2,6], 프로그램 통합[5], 프로그램 최적화[3] 등이 있다.

본 논문에서는 함수 호출이 존재하는 프로그램에 대해 시스템 정보 흐름 그래프(System Information Flow Graph:이하 SIFG라 함)를 이용한 정적 프로그램 슬라이싱 방법을 제안한다 슬라이싱 대상 프로그램은 C언어로 작성된 프로그램이며, 입·출력문, 스칼라 변수와 상수를 지닌 수식문, 배정문, While문, If문, 함수 호출문으로 구성되어있다. 전역 변수의 사용은 제한되며, 포인터 변수에 대한 연산들도 제한된다. 단, 함수 호출 시 포인터 변수가 인자로 나타나는 경우와 *x=2, *x=*y와 같은 연산은 허용한다. 또한 재귀적 함수 호출과 앨리어싱 발생도 제한한다.

본 논문의 구성은 다음과 같다. 2장은 함수 호출이 존재하는 프로그램에 대한 기존의 슬라이싱 관련 연구들을 분석하고 문제점을 지적한다. 3장은 본 논문에서 제안하는 슬라이싱에 기반이 되는 그래프를 정의한다. 4장은 기존의 슬라이싱 방법을 기술하고 SIFG를 이용한 모듈간 프로그램 슬라이싱 방법을 보이며, 5장은 결론 및 향후 연구과제이다.

## 2. 연구 배경

[Weiser]의 모듈간 슬라이싱 알고리즘은 호출된 함수에 대한 슬라이싱 수행 시 호출 문맥을 정확하게 설명하지 못하여 정확한 슬라이스 생성에 실패하게 된다. 슬라이싱 기준이 문장 번호 n과 관심 있는 변수 집합 V의 쌍인 c=<n, V>로 주어지고, 함수 P는 초기 슬라이싱 기준에 나타난 문장 n을 포함한다고 하자. 이때 P의 호출 함수들에 대한 슬라이싱 기준들의 집합인 $UP_0(c)$과 P가 호출하는 함수들에 대한 슬라이싱 기준 $DOWN_0(c)$이 필요하게 되며, 또한 $UP_0(c)$과 $DOWN_0(c)$에 대하여 새로운 슬라이싱 기준들의 집합이 발생하게 된다. 즉, 호출하고 호출되는 함수들에 대한 슬라이싱 기준들의 집합인 $(UP \cup DOWN)^*$이 생성된다. $(UP \cup DOWN)^*(<n, V>)$ 관계는 UP(DOWN(<n, V>)) 관계를 포함한다. 이는 DOWN(<n, V>)에 있는 함수를 호출하는 모든 호출 위치에 대한 슬라이싱 기준을 생성함을 의미하며, 또다시 n문장을 지니고 있는 함수 P에 대한 새로운 슬라이싱 기준도 생성시킨다. 여기에서 호출된 함수들에 대한 정확한 함수 호출 문맥 설명에 실패한다.

[Horwitz]는 함수 호출 관계가 존재하는 프로그램을 표현할 수 있는 시스템 종속성 그래프(System Dependence Graph 이하 SDG라 함)를 이용한 모듈간 슬라이싱 방법을 제안하였다[4]. 그러나 SDG는 문장 기반 그래프이다. 즉 하나의 문장을 하나의 노드로 표현함으로써 큰 슬라이스를 생성하게 된다. [Jingyue]의 경우, 슬라이싱 대상 함수에 대하여 호출하고 호출되어지는 함수의 슬라이싱 기준 확장에 대한 정확한 규칙 설명에 실패한다[7].

## 3. 시스템 정보 흐름 그래프 (System Information Flow Graph)

### 3.1 프로시듀어 정보 흐름 그래프(Procedure Information Flow Graph)

프로그램은 제어 흐름 그래프(Control Flow Graph:이하 CFG라 함)로 쉽게 변환되어질 수 있다. 프로시듀어 정보 흐름 그래프(Procedure Information Flow Graph. 이하 PIFG라 함)는 CFG를 기반으로 하며, 정확한 슬라이스 생성을 위하여 필요한 정보가 추가된 그래프이다. 본 장에서는 기존의 CFG 상에 자료 정보와 제어 종속성 정보, 그리고 문맥 종속성 정보 등을 추가한 PIFG를 정의한다. PIFG 정의에 기반이 되는 용어 정의는 다음과 같다.

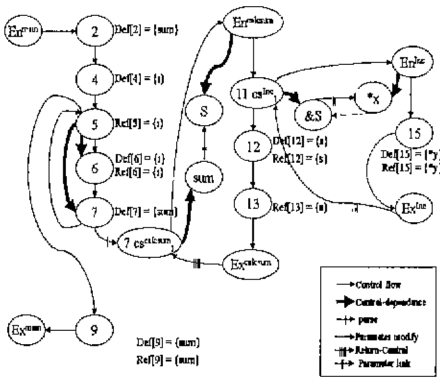
**정의3.1** : CFG는 G=<N, E, Entry, Exit>로 표현되며, 각 노드는 프로그램의 한 문장에 대응된다. 문장들간의 제어 흐름을 표현하는 제어 흐름 에지는 $n_i \rightarrow_{cf} n_j$로 표기한다($n_i, n_j \in N$). Entry와 Exit는 프로그램의 진입점과 진출점을 나타내는 노드들이다.

**정의3.2** : CFG의 에지 $n_i \rightarrow_{cf} n_j \in E$에 대하여, $n_i$는 $n_j$의 선행자이며, $n_j$는 $n_i$의 후행자이다 n의 선행자들의 집합을 Pred(n)으로,

```
[1]void main() {
[2] int sum=0;
[3] int i;
[4] i=0;
[5] while( i< 10 ) {
[6] i = i + 1;
[7] sum=calcsum(sum); }
[9] sum=sum; }

[10] int calcsum(int s) {
[11] inc(&s);
[12] s= s + 9;
[13] return s; }

[14] void inc(int *x) {
[15] *x=*x +1; }
```




**<그림1> 함수 호출이 존재하는 프로그램과 대응 SIFG**

n의 후행자들의 집합을 Succ(n)으로 표현한다. IMS(n)은 노드 n 바로 직후에 나타나는 후행자들의 집합으로 즉시 후행자이다 IMP(n)은 문장 n 바로 직전에 나타나는 선행자들의 집합으로 즉시 선행자이다.

**정의3.3** : CFG의 n에서 Exit로 가는 모든 경로 상에 노드 m이 존재한다면 m을 n의 포스트도미네이터라고 한다

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

**정의3.5** : 노드 n에서 노드 m으로의 제어 종속성 에지 관계를 $n \rightarrow_{cd} m$으로 표기하며, 다음의 조건을 만족한다.

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

(2) m∉PDOM(n) 즉, m은 n의 포스트도미네이터가 아니다.

C언어에서의 goto, continue, break, return문과 같은 Jump문도 정확한 슬라이스 생성을 위하여 식별되어져야 한다. Jump문 식별을 위해 필요한 용어들에 대한 정의는 다음과 같다.

**정의3.6** : Jump문에 대응되는 노드 J와 J가 존재하지 않을 경우 문맥상 실행 제어가 도달되는 노드 m에 대하여 문맥 종속성 에지를 연결하고 $J \rightarrow_{cxd} m$으로 표기한다.

**정의3.7** : PIFG는 $\Theta$=<N,E,Σ,D,U>로 표현한다 N은 CFG의 노드를 나타내며, E는 제어 흐름 에지와 제어 종속성 에지 그리고 문맥 종속성 에지를 나타낸다 Σ는 프로그램에 나타난 변수들의 집합이다. D: $N_{\Theta} \rightarrow \Delta(\Sigma)$, U: $N_{\Theta} \rightarrow \Delta(\Sigma)$로써, 각 노드를 대응되는 프로그램 문장에서 정의되고 참조되는 변수로 대응해 주는 함수이다

### 3.2 시스템 정보 흐름 그래프(System Information Flow Graph)

SIFG는 여러 개의 함수로 구성된 하나의 프로그램에 대하여 각각의 함수를 PIFG로 표현한 후, 이들의 호출 구조를 표현한 그래프이다 SIFG는 함수 호출이 존재하는 프로그램 슬라이싱에 이용되어진다. SIFG는 기본적으로 문장 기반 그래프이지만, 정확한 슬라이스 생성을 위하여 반환값이 있는 함수 호출이 프로그램은 메인 프로그램[1] M과 여러 함수($F^k$)(k≥0)로 구성된다. 존재하는 문장은 해당 문장 노드의 함수 호출 노드로 분해하여 표현한다.

#### 3.2.1 노드

함수간의 호출 구조 표현을 위해 필요한 SIFG의 노드들은 다음과 같다 함수 $F^k$를 호출하는 호출 노드를 $cs^{Fk}_j$로 표현한다. SIFG 상에서 문장 n에 대응되는 노드에서 발생한 함수 호출 노드는 n-$cs^{Fk}_j$로 레이블링 한다. j는 함수 $F^k$의 정적 호출 횟수를 세기 위해 사용되는 음이 아닌 정수이다. SIFG에서 반환값이 없는 함수에 대한 호출 문장은 하나의 함수 호출 노드로 표현한다. 그리고 반환값이 있는 함수에 대한 호출 문장은 해당 문장번호로 레이블링된 문장 노드와 그 문장에서 발생한 각각의 함수 호출에 대한 함수 호출 노드로 표현한다 PIFG상에서 반환값이 있는 함수 호출이 발생하는 문장 n에 대하여, n으로 들어오는 에지들과 n으로부터 나가는 에지는 n으로 레이블링된 노드와 연결한다

함수 호출 노드의 실인자에 대한 actual 노드와 형식인자에 대한 formal 노드를 생성한다. actual 노드를 $actual^{Fk}_{i,j}$로 표현하며 i는 실인자의 위치 값이다 실인자들은 호출 노드들에 대해 제어 종속적이다.

$$\forall j\ \forall k\ \forall v \in \left(actual^{F^k}_{i,j}\right) \rightarrow (cs^{F^k}_{j} \rightarrow_{cd} v)$$

함수 $F^k$의 "entry"노드를 $En^{Fk}$로 표현하고, "exit"노드를 $Ex^{Fk}$로 표현한다. $actual^{Fk}_{i,j}$ 에 대응하는 formal 노드를 $formal^{Fk}_i$로 표현한다. 위 노드들은 함수 $F^k$의 "entry" 노드인 $En^{Fk}$에 제어 종속적이다

$$\forall k\ \forall v \in \left(formal^{F^k}_{i}\right) \rightarrow (En^{F^k} \rightarrow_{cd} v)$$

호출 노드와 함수 $F^k$의 "entry" 노드인 $En^{Fk}$ 간에 제어 흐름 에지를 연결한다.

$$\forall j\ (cs^{F^k}_{j} \rightarrow_{cf} en^{F^k}),\ \text{indegree}(En^{Fk}) = j_k$$

#### 3.2.2 에지

반환값을 지니는 함수 $F^k$에 대한 호출이 발생한 문장 n은, n으로 레이블링된 노드와 $cs^{Fk}_j$ 레이블링된 노드로 분해되어진다 그리고 n에서 $cs^{Fk}_j$로 파스(parse)에지를 연결한다

$$.\ (\ st.\ n\ has\ function\ calls\ for\ F^k\ that\ return\ a\ value) \rightarrow (n \rightarrow_{p} cs^{F^k}_{j})$$

함수 $F^k$의 return문에 대응되는 노드는 $Ex^{Fk}$ 노드로 제어 흐름 에지를 지닌다. return문은 프로그램 상에서 무조건 Jump문에 대응된다. 따라서 return문에 대응되는 노드와 return문이 문맥상 제거될 경우 실행 제어가 도달되는 함수 내의 노드들끼리 문맥 종속성 에지를 연결한다. 이는 필요한 return문을 식별하여 원래 프로그램과 수행 동작이 동일한 슬라이스를 생성하기 위해서이다.

함수 $F^k$의 "exit" 노드인 $Ex^{Fk}$와 호출 노드간에 return-control 에지를 연결한다.

$$\forall j\ (Ex^{F^k} \rightarrow_{rc} cs^{F^k}_{j}),$$

actual 노드들과 대응되는 formal 노드들 간에 parameter_link 에지를 연결한다.

$$\forall j\ \forall i\ (actual^{F^k}_{i,j} \rightarrow_{p_{link}} formal^{F^k}_{i})$$

함수 P가 슬라이싱 되어지고, P의 i번 문장에서 함수 Q에 대한 호출이 발생한다고 하자. 이때 call-by-reference 방식으로 전달된 $actual^{Fk}_{i,j}$ 노드와 이에 대응되는 $formal^{Fk}_i$ 노드에 나타나는 변수가 Q내부에서나 혹은 Q $call^*$ R의 호출 관계에 놓인 함수 내부에서 변경되어지는 노드 n이 존재한다면 $formal^{Fk}_i$ 노드에서 대응 $actual^{Fk}_{i,j}$ 노드로 parameter_modify 에지를 연결한다.

$$\forall j\ \forall i\ [\ formal^{F^k}_{i} \rightarrow_{pm} actual^{F^k}_{i,j}\ ] \rightarrow$$

$$[\exists n,\ formal^{F^k}_{i} \in Def(n) \wedge formal^{F^k}_{i}\ is\ a\ call-by-reference\ variable]$$

1) 본 논문에서 메인 프로그램은 함수 용어로 통일한다

함수 호출이 존재하는 프로그램과 대응 SIFG는 <그림 1>과 같다

## 4. 프로그램 슬라이싱

함수 호출이 존재하는 프로그램을 SIFG로 표현하여 기존의 알고리즘들보다 더 정확한 슬라이스를 생성함을 보인다.

### 4.1 PIFG를 이용한 모듈내 슬라이싱

슬라이싱 기준 c=<i, V>는 프로그램의 문장 번호와 프로그램에 나타나는 변수들 부분 집합의 쌍이다. [Weiser]의 정의에 따르면, 슬라이싱 기준 c가 주어졌을 때 i문장이 수행하기 직전까지 V에 있는 변수들에게 영향을 준 n(∈Pred(i)) 문장에 나타난 변수들의 집합을 $RIN_c^0(n)$으로 나타낸다 슬라이싱 기준 c=<i, V>가 주어졌을 때, $RIN_c^0(n)$을 구하는 공식은 다음과 같다

$$RIN_c^0(n) = \{v \in V \mid n = i\} \cup$$
$$\{Ref(n) \mid RIN_c^0(IMS(n)) \cap Def(n) \neq \emptyset\} \cup \{RIN_c^0(IMS(n)) - Def(n)\}$$

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

$$S_c^0 = \{n \mid Def(n) \cap RIN_c^0(IMS(n)) \neq \emptyset\}$$

본 논문에서의 슬라이싱 알고리즘은 기본적으로 [Weiser]의 자료 흐름 방정식을 이용한다. PIFG를 이용하여 생성된 프로그램 슬라이스 정의는 다음과 같다.

**정의4.1** : 프로그램 P에 대하여 슬라이싱 기준 c=<n, V>가 주어졌을 때, $S(PIFG_P, c)$는 프로그램 P의 PIFG(이하 $PIFG_P$로 표현)를 이용하여 생성된 슬라이스이다. $PIFG_P$에서 슬라이싱 기준에 영향을 주지 않는 노드들을 제거함으로써 $S$의 PIFG(이하 $PIFG_S$로 표현)를 생성하게 된다. 생성된 슬라이스는 동일한 입력에 대해 P의 수행 동작과 동일한 수행 가능한 프로그램이며, $S \subseteq P$이다.

### 4.2 SIFG를 이용한 모듈간 슬라이싱

Q는 슬라이싱을 수행하고 있는 함수이고, Q의 n번 문장에서 함수 P를 호출할 경우, 함수 P는 반환값이 없고 함수 내부에서 슬라이싱 기준에 나타나는 변수들의 값을 변경하지 않는다면 그 함수에 대한 슬라이싱을 수행할 필요는 없다. 함수 P에 대한 슬라이싱 수행 이전에 변수 x의 값이 함수 P의 영향을 받지 않는다는 정보를 추출할 수 있다면 슬라이싱 계산 시간을 단축시킬 수 있다. SIFG의 formal 노드로부터 actual 노드로의 parameter-modify 에지는 call-by-reference 방식으로 전달된 formal 노드의 값이 함수 내부에서 변경된 경우 존재하는 에지이다. 따라서 슬라이싱 기준에 나타난 call-by-reference 방식으로 전달된 실인자 중 들어오는 parameter-modify 에지가 존재한다면 그 함수는 슬라이싱 되어져야한다. 즉, 함수 P가 슬라이싱 기준에 나타나는 변수 중 call-by-reference 방식으로 전달된 변수들의 값을 변경한다면, 새로운 슬라이싱 기준을 생성하여 P는 슬라이싱 되어져야한다.

또한 함수 P가 반환값을 지니는 경우, P에 대한 호출이 나타나는 문장이 슬라이스에 포함된 경우만 슬라이싱을 수행한다 즉 함수 P의 반환값은 Def(n)의 값에 영향을 주기 때문이다. 그러나 문장 n이 슬라이스에 포함되지 않더라도 call-by-reference 방식으로 전달된 인자 중 슬라이싱 기준에 포함된 변수가 존재한다면, 그 노드에 대한 parameter-modify 에지로써 슬라이싱 수행 여부를 결정짓는다.

호출 함수의 경우, 슬라이싱 초기 기준이 Q내에 존재하고 P $call^*$ Q의 관계가 존재할 경우 P에 대한 슬라이싱을 수행한다. 함수 호출이 존재하는 모듈간 프로그램 슬라이싱 방법은 다음과 같다

Ⅰ. 호출하는 함수에 대한 슬라이싱

c=<n, V>, q=IMP(s), n은 함수 $F^k$에 존재하고, $F^x$ $call^*$ $F^k$의 관계에 있다고 하자. 또한 $F^x$의 s 문장에서 $F^k$에 대한 함수 호출이 발생했다고 가정하자

1. $\forall j, \exists formal^{Fk}_j \in RIN_{CFK}^0(En^{Fk}) \wedge \exists x \{x \mid x \in S_c^0 \wedge x \in F^k\} \rightarrow$
$[S_c^0 \cup s\,cs^{Fk}_j] \wedge$
$c_{Fk} = \langle q, RIN_c^0(s) \cup actual^{Fk}_{i,j} \cup RIN_{CFK}^0(En^{Fk})\rangle$
2. otherwise, $c_{Fk} = \langle q, RIN_c^0(s) \cup RIN_{CFK}^0(En^{Fk})\rangle$

Ⅱ. 호출되는 함수에 대한 슬라이싱

호출되는 함수의 경우, 반환값을 가지는 경우와 가지지 않는 경우를 각각 고려해야한다.

c=<i, V>, m=IMS(n), q=IMP(n)이라고 가정하자.

1 노드 n은 반환값이 있는 함수 $F^k$가 호출되는 문장에 대응될 경우,

a) $[Def(n) \in RIN_c^0(m)] \wedge \forall j, [\exists$(call-by-reference된 $actual^{Fk}_{i,j}) \in RIN_c^0(m)] \rightarrow [S_c^0 \cup n] \wedge RIN_c^0(n)$ is suspended
$C_{Fk} = \langle Ex^{Fk}$, return문에 존재하는 변수 $\cup\ formal^{Fk}_j \rangle$

b) $[Def(n) \in RIN_c^0(m)] \rightarrow [S_c^0 \cup n] \wedge RIN_c^0(n)$ is suspended
$C_{Fk} = \langle Ex^{Fk}$, return문에 존재하는 변수$\rangle$

c) $[Def(n) \notin RIN_c^0(m)] \rightarrow$ Ⅱ.2.a) 수행

2. 노드 n은 반환값이 없는 함수 $F^k$가 존재하는 문장에 대응될 경우,

a) $\forall j, \exists\{K \mid k \in$(call-by-reference된 $actual^{Fk}_{i,j}) \wedge RIN_c^0(m)\} \wedge$
$\exists\{k \mid formal^{Fk}_j \rightarrow_{pm} k\} \rightarrow [S_c^0 \cup n] \wedge RIN_c^0(n)$ is suspended
$C_{Fk} = \langle Ex^{Fk}, formal^{Fk}_j \rangle$

3 otherwise, $RIN_c^0(q)$ 계산수행

호출되는 함수쪽으로 슬라이싱 수행이 진행될 때, $RIN_c^0(n)$에 대한 계산은 잠시 중단된다 호출되는 함수에 대한 슬라이싱 계산이 종료하게 되면 중단되었던 $RIN_c^0(n)$ 계산이 재개되며 n으로 제어흐름 에지가 들어오는 q=IMP(n)쪽으로의 계산이 계속 진행된다 이때 중단된 $RIN_c^0(n)$에 대한 계산은 Ⅰ.1 또는 Ⅰ.2와 같다.

## 5 결론

본 논문에서는 정확한 모듈간 슬라이스 생성을 위해 제어 흐름 그래프 상에 제어 종속성 정보와 문맥 종속성 정보, 정의되거나 참조되어진 변수들에 대한 자료 정보 그리고 함수들의 호출 구조를 표현하기 위해 필요한 정보가 추가된 SIFG를 기반으로 하였으며, SFG로 표현된 프로그램에 대한 슬라이스는 기존의 방법들보다 더 정확한 슬라이스임을 보였다

현재 포인터 변수와 배열이 존재하는 프로그램에 대한 정확한 슬라이싱 문제를 연구 중에 있으며, 자동화된 C언어에 대한 슬라이서 도구 개발이 향후 연구 과제이다.

## 참 고 문 헌

[1] Mark Weiser, "Program Slicing", IEEE Trans Software Eng., vol. SE-10, no 4, July 1985

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

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

[4] SUSAN HORWITZ, THOMAS REPS, and DAVIDBINKLEY, "Interprocedural Slicing using Dependence Graph", ACM Trans Programming Languages and Systems, vol 12, no 1, January 1990

[5] 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)

[6] Keith Brian Gallagher, and James R Lyle, "Using Program Slicing in Software Maintenance", IEEE Trans Software Eng, vol. SE-17, no 8, august 1991

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