요약 해석을 이용한 프로그램 슬라이싱
(Program Slicing using Abstract Interpretation)

정 인 상 † 창 병 모 ††
(In sang Jung) (Byeong-Mo Chang)

요 약 정적 슬라이싱과 동적 슬라이싱의 차이는 동적 슬라이싱은 프로그램에 주어진 입력을 가정하는 반면에 정적 슬라이싱은 입력에 대한 가정을 하지 않는다는 점이다. 동적 슬라이싱은실행-시간 정보를 이용할 수 있으므로 정적 슬라이싱보다 적은 슬라이스를 만들 수 있으나 특정 입력 상태에만 적용될 수 있다는 제한을 갖는다. 이 논문은 초기 상태들의 집합에 대해서 프로그램을 슬라이싱하는 요약 프로그램 슬라이싱이라는 새로운 기법을 제시한다. 이 방법은 프로그램으로부터 슬라이스를 구하는데 요약 해석을 이용한다. 요약 해석은 프로그램 실행 없이 프로그램의 실행-시간 동작에 대한 안전한 정보를 제공한다. 따라서 결과적으로 얻은 요약 슬라이스는 주어진 입력 상태들의 집합에 대해서 정적으로 계산되었다는 점에서 동적 슬라이스와는 다르다. 또한 요약 프로그램 슬라이싱은 배열과 같은 자료구조를 정적 슬라이싱보다 정확하게 다룰 수 있으며 슬라이스 크기도 줄일 수 있다.

Abstract The difference between static and dynamic slicing is that dynamic slicing assumes a fixed input for a program, whereas static slicing does not make assumptions regarding the input. Due to the availability of run-time information, dynamic slicing makes slices considerably smaller than slices produced by static slicing, but limits its applicability to just one particular initial state characterized by the given input. In this paper, we present a new slicing technique named abstract program slicing that allows a decomposition of a program for the set of initial states. We apply abstract interpretation to the derivation of slices from existing programs. Abstract interpretation allows us to yield safe information about the run-time behavior of the program without having to run it for all input data. Thus, the resulting slice differs from the dynamic slice in that its computation is done statically for the given set of initial states. Also, abstract program slicing can handle data structures such as arrays more precisely than static slicing and the size of slice can be reduced significantly.

1. 서 론

프로그램 슬라이싱(program slicing)은 역공학[8], 테스팅[5,7], 디버깅[1,9,13], 재사용[10], 복잡도 분석[2] 등을 포함하는 여러 소프트웨어 공학 분야에서 이용되어 왔다. 프로그램 슬라이싱은 프로그램 내의 위치 p 및 변수들 v로 이루어진 슬라이싱 기준(slicing criterion) (p, V)에 대해서 수행된다. 프로그램 슬라이싱의 주요 목표는 슬라이싱 기준에 있는 변수들의 값에 직접 혹은 간접적으로 영향을 주는 문장들을 추출하는 것이다[14].

다양한 프로그램 슬라이싱 개념이 제안되어 왔지만 대부분 정적 슬라이싱(static slicing)과 동적 슬라이싱(dynamic slicing)을 기초로 하고 있다. 정적 슬라이싱과 동적 슬라이싱의 차이는 동적 슬라이싱은 프로그램에 대한 지정된 입력을 가정하는 반면에 정적 슬라이싱은 입력에 대한 가정을 하지 않는다는 점이다[12].

동적 슬라이싱 기준은 입력을 명시하며 실행 궤적에서 한 문장이 여러 번 나타나는 것을 구별한다[1,9]. 동적 슬라이싱은 실행 시간 정보를 이용하여 정적 슬라이싱보다 작은 프로그램 슬라이스를 만들 수 있다. 예를 들어 배열을 처리하는 방식을 생각해보자. 동적 슬라이싱은 프로그램 실행 중에 배열의 어떤 원소가 사용되거

· 이 논문은 1999년도 한국학술진흥재단의 연구비에 의하여 지원되었음 (KRF-99-E00280)
† 종신회원 : 한성대학교 정보전산학부 교수
insang@hansung.ac.kr
†† 종신회원 : 숙명여자대학교 정보과학부 교수
chang@cs.sookmyung.ac.kr
논문접수 : 2000년 12월 29일
심사완료 : 2001년 6월 7일

나 갱신되었는지 결정할 수 있지만 정적 슬라이싱은 배열 원소의 사용 혹은 갱신을 전체 배열의 사용 혹은 갱신으로 간주한다[6,14]. 결과적으로 동적 슬라이싱은 정적 슬라이싱보다 작은 프로그램 슬라이스를 만들 수 있다. 그러나 동적 슬라이싱은 주어진 입력에 따른 특정 궤적에만 제한되며 결과적으로 동적 슬라이스는 오직 하나의 입력에 대해서만 원래 프로그램과 같은 동작 유지를 보증한다.

본 논문은 *요약 프로그램 슬라이싱*(abstract program slicing)이라는 새로운 슬라이싱 기법을 제시한다. 이 기법은 하나의 지정된 입력이 아니고 여러 입력들의 집합에 대해서 프로그램의 슬라이스를 구한다. 이를 위해서 *요약 해석*(abstract interpretation)이라는 프로그램 분석 기법을 적용한다. 요약 해석은 프로그램을 실제 실행시키지 않고 프로그램의 실행 시간 동작에 대한 안전한 정보를 제공한다[4]. 따라서 *요약 슬라이스*(abstract slice)라고 불리는 프로그램 슬라이스는 여러 입력들의 집합에 대해서 정적으로 계산되었다는 점에서 동적 슬라이스와는 다르다. 또한 요약 슬라이싱은 배열과 같은 자료구조를 정적 슬라이싱보다 정확하게 다룰 수 있으며 슬라이스의 크기도 상당히 줄일 수 있다.

본 논문은 요약 해석에 대한 배경 설명으로 시작한다. 3절에서는 요약 프로그램 슬라이싱을 소개하고 프로그램을 의존 그래프로 표현하여 요약 프로그램 슬라이스를 구하는 방법을 설명한다. 4절에서는 관련 연구에 대해서 토의한다. 결론에서는 향후 연구 방향을 제시한다.

2. 요약 해석

프로그램 분석의 목표는 프로그램을 실제 실행하지 않고 프로그램의 실행 시간 동작에 대한 가능한 많은 정보를 얻는 것이다. 프로그램 분석을 위한 이러한 기법의 하나가 요약 해석이다[3,4]. 요약 해석은 실제 값 대신에 요약 값을 사용하여 프로그램의 동작 정보를 계산할 수 있으며 이를 통해서 안전한(safe) 근사 정보를 제공한다. 여기서 안전의 의미는 프로그램의 모든 가능한 실제 실행결과를 포함한다는 것을 의미한다.

요약 해석에서는 해당 프로그래밍 언어에 대해서 분석하고자 하는 성질만을 고려한 모음 시맨틱스(collecting semantics) 즉 실제 시맨틱스(concrete semantics)를 정의한다. 직관적으로 설명하면 이 실제 시맨틱스는 관심이 되고 있는 성질만을 기술하는데 가장 정확한 시맨틱스이며 앞으로 근사 혹은 요약 시맨틱스의 안전성을 증명하는데 기준 역할을 한다. 요약 해석을 위해서는 앞서 설명한 실제 시맨틱스를 기반으로 수행시 동작에 대한 안전한 근사 정보를 주는 요약 시맨틱스(abstract semantics)를 정의하고 이를 계산한다. 실제 시맨틱스와 요약 시맨틱스 간의 대응관계는 한 쌍의 함수들 (α,γ)로 정의되는 Galois 연결(connection)로 기술되는데 이 연결은 실제 값에 대응하는 최선의 근사치인 요약 값이 존재한다는 것을 표현한다.

실제 도메인(concrete domain)이 래티스(lattice)$(D, \sqsubseteq)$로 표현된다면 이를 요약하는 요약 도메인을 래티스(lattice)$(D^{\#}, \sqsubseteq^{\#})$로 정의할 수 있으며 $d_1^{\#} \sqsubseteq d_2^{\#}$는 $d_1^{\#}$이 $d_2^{\#}$보다 정확한 값임을 의미한다. 요약 과정을 나타내는 요약 함수(abstraction function) $\alpha : D \rightarrow D^{\#}$의 목적은 실제 도메인 D 내의 실제 값을 요약 도메인 $D^{\#}$ 내의 근사값에 순서를 존중하면서 대응시키는 것이다. $d^{\#}$가 d의 유효한 근사치라는 사실은 $\alpha(d) \sqsubseteq^{\#} d^{\#}$로 표시될 수 있다. 요약된 값의 의미를 나타내는 실제화 함수(concretization function) $\gamma : D^{\#} \rightarrow D$는 $D^{\#}$내의 요약 값을 D 내의 값에 순서를 존중하면서 대응시킨다. $d^{\#}$가 d에 대한 유효한 근사치라는 사실은 $d \sqsubseteq \gamma(d^{\#})$ 처럼 표시될 수 있다.

이러한 두 조건이 동등할 때 $<\alpha,\gamma>$는 두 래티스를 오가면서 순서를 잃지 않는 Galois 연결이 된다. 직관적으로 Galois 연결은 최선의 요약(best approximateion) 정보가 α에 의해서 정의된다는 것을 의미한다. 정형화해 보면 D와 $D^{\#}$ 사이의 Galois 연결은 다음 조건을 만족하는 두 함수 $\alpha : D \rightarrow D^{\#}$와 $\gamma : D^{\#} \rightarrow D$의 쌍 $<\alpha,\gamma>$이다

$$\forall d \in D : \forall d^{\#} \in D^{\#} : \alpha(d) \sqsubseteq^{\#} d^{\#} \leftrightarrow d \sqsubseteq \gamma(d^{\#})$$

예를 들어 정수 집합 Z의 멱집합 $\wp(Z)$를 부호를 중심으로 요약하는 요약 도메인 L^{sign}를 살펴보자. 이 요약 도메인은 래티스로 구성되며 각 원소들 간의 관계는 그림 1과 같다.

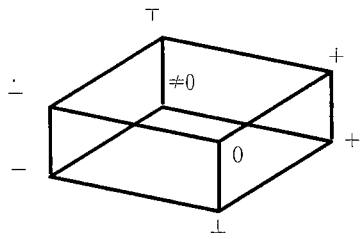

그림 1 요약 도메인 L^{sign}

또한 실체화 함수 $\gamma : L^{sign} \rightarrow \wp(Z)$는 요약 값을 다음과 같이 대응시킨다.

$\gamma(\top)=Z$ $\gamma(\div)=\{i \mid i\leq 0\}$ $\gamma(+)=\{i \mid i\geq 0\}$
$\gamma(\neq)=Z$ $\gamma(-)=\{i \mid i<0\}$ $\gamma(+)=\{i \mid i>0\}$
$\gamma(0)=\{\}$ $\gamma(\bot)=\{\}$

프로그램의 근사 시맨틱스인 요약 시맨틱스를 정의하기 위해서는 실제 도메인 상에서 정의된 실제 연산에 대한 근사치를 제공하는 요약 연산을 요약 도메인 상에서 정의해야 한다.

요약 계산은 실제 계산을 시뮬레이션한 것으로 요약 계산 결과의 실제화된 의미가 실제 계산 결과 값에 대한 안전한 근사값이다. 예를 들어 "부호 규칙"에 따른 덧셈 혹은 곱셈 요약 연산을 이용하여 요약 해석에서는 "이 루프에 진입할 때마다 변수 x의 값은 양수 값이 된다"와 같은 프로그램의 어떤 성질을 알아낼 수 있다.

[4]에서 처럼 프로그램 P의 실제 시맨틱스는 *State*가 프로그램의 모든 가능한 상태들의 집합이라고 하면 P에 대해 정의된 시맨틱 함수 $F_P: \wp(State) \rightarrow \wp(State)$의 최소고정점 $lfp(F_P)$로 정의된다고 가정한다. 실제 시맨틱스는 무한할 수 있으므로 실제 시맨틱스에 대한 안전한 근사값(즉 요약 시맨틱스)을 요약해석으로 계산한다. 프로그램 P의 요약 시맨틱스는 요약 시맨틱 함수 $F_P^{\#}$의 최소고정점 $lfp(F_P^{\#})$으로 정의되며 $F_P^{\#}$가 F_P에 대한 안전한 근사 연산이면 실제 시맨틱스에 대한 안전한 근사값을 제공한다.

이는 실제 도메인($\wp(State), \subseteq$)과 요약 도메인($AbsState, \sqsubseteq$) 사이의 Galois 연결 $(\alpha_\sigma, \alpha_\sigma)$에 의해 설명될 수 있는데 *State*와 *AbsState*는 각각 가능한 실제 상태들의 집합과 요약 상태들의 집합을 의미한다. 일단 요약 도메인 *AbsState*가 선택되면 시맨틱 함수로부터 요약 시맨틱 함수 $F_P^{\#}$를 유도할 수 있다. 요약 시맨틱 함수는 기초 연산들의 요약 버전들을 이용하는 방식으로 특정 프로그램과 무관하게 정의될 수 있다. 요약 시맨틱 함수 $F_P^{\#}$가 시맨틱 함수 F_P에 대한 안전한 근사 연산 즉

$$\alpha_\sigma \circ F_P \circ \gamma_\sigma \sqsubseteq F_P^{\#}$$

이면 $F_P^{\#}$의 최소고정점으로 정의되는 프로그램 P의 요약 시맨틱스는 실제 시맨틱스에 대한 안전한 근사값이다[3]. 즉, 다음 정리가 성립한다.

정리 1[3]. 프로그램 P에 대한 요약 시맨틱 함수 $F_P^{\#}$가 시맨틱 함수 F_P에 대한 안전한 근사 연산이면 다음이 성립한다.

$$lfp(F_P) \subseteq \gamma_\sigma(lfp(F_P^{\#}))$$

3. 요약 해석 기반 프로그램 슬라이싱

이 절에서는 요약 해석을 기초로 한 슬라이싱 모델을 기술한다. 요약 해석을 이용하여 명시된 입력들의 집합에 대해서 각 프로그램 포인트마다 요약 시맨틱스를 계산하게 된다. 이 정보를 이용하여 명시된 입력들에 대해서 어떤 프로그램 부분이 실행되지 않는지 구별한다. 원래 프로그램으로부터 실행되지 않는 부분을 제거한 후에 이 프로그램을 요약 프로그램 의존성 그래프 형태로 표현한다. 이 그래프에 대한 그래프 도달 문제에 대한 답을 구함으로써 프로그램 슬라이스를 계산하게 된다.

3.1 요약 프로그램 슬라이스

전통적인 정적 슬라이싱 기준은 어떤 프로그램 포인트에서 변수들의 집합이다. 결과적으로 전통적인 정적 슬라이스는 *임의의* 가능한 실행에 대해서도 기준에 나타난 변수들과 프로그램 포인트에 대해서 원래 프로그램의 동작을 보존한다. 그렇지만 우리가 제안한 슬라이싱 모델은 이 동작 보존 개념을 프로그램 실행들의 지정된 집합에 한정하므로 슬라이싱 기준의 정의에서 입력에 대한 명세를 고려할 필요가 있다.

정의 1. ϕ를 입력 값에 대한 가정을 나타내는 입력 변수들 V_{in}에 대한 프레디킷트라고 하자. 프로그램 P에 대한 (실제) 슬라이싱 기준은 3-튜플 $C=(\phi, p, V)$로 p는 P 내의 프로그램 포인트이고 V는 P 내의 변수들의 부분집합이다.

일반적으로 $C=(\phi, p, V)$를 슬라이싱 기준으로 *정확한* 프로그램 슬라이스를 계산하는 것은 각 프로그램 포인트에 대해서 귀납 가정의 결정 혹은 불변 조건의 고정점 계산과 같은 시맨틱 분석을 포함한다. 이러한 작업들은 보통 사람과의 상호작용을 필요로 하며 시간이 걸린다. 슬라이싱 기법이 실용적이기 위해서는 프로그램에 대한 추가 정보를 손수 첨부하는 것과 같은 수작업 없이 프로그램 슬라이싱 전과정이 자동화되어야 한다. 이를 위해서는 프로그램에 대한 근사 정보를 이용하여 정확에 가까운 프로그램 슬라이스를 자동적으로 계산하는 방법이 필요하다.

정의 2. 도메인 D와 $D^{\#}$는 Galois 연결 (α, γ)에 의해 연결되어 있다고 하자. 실제 슬라이싱 기준이 $C=(\phi, p, V)$ 이라면 프로그램 P의 요약 슬라이싱 기준은 3-튜플 $C^{\#}=(\phi^{\#}, p, V)$에 의해 정의되며 $\phi^{\#}=\alpha(\phi)$이다.

즉 $\alpha(\phi)$는 초기 조건 $\phi \in D$를 요약한 안전한 근사 표현이다. 예를 들어 ϕ를 변수 x와 y에 대한 다음과 같은 초기 조건이라고 하자 :

$$\phi = (x>y+3) \wedge (y\geq 0)$$

만약 그림 1의 부호 래티스를 프레디킷트 $P \in Integer \rightarrow Boolean$에 대한 안전한 요약 도메인으로 생각하면 α

(ϕ)를 다음의 α_1과 α_2로 정의할 수 있다.

$$\alpha_1 = \lambda p \in Integer \rightarrow Boolean \cdot \begin{cases} \bot & if\ p = \lambda x.false \\ 0 & if\ p \Rightarrow \lambda x.x = 0 \\ - & if\ p \Rightarrow \lambda x.x < 0 \\ + & if\ p \Rightarrow \lambda x.x > 0 \\ -\!\!\!\!\!\!\cdot & if\ p \Rightarrow \lambda x.x \leq 0 \\ \neq & if\ p \Rightarrow \lambda x.x \neq 0 \\ \top & otherwise \end{cases}$$

$$\alpha_2 = \lambda p \in Integer^2 \rightarrow Boolean. \langle \alpha_1(\exists y.p(x,y)), \alpha_1(\exists x.p(x,y)) \rangle$$

따라서 $\alpha_2(\lambda\langle x,y\rangle.((x>y+3)\wedge(y\geq 0))=\langle\alpha_1(\lambda x.(x\geq 3)), \alpha_1(\lambda y.(y\geq 0))\rangle=\langle +, +\rangle$이며 결과적으로 $\phi^{\#}$는 다음과 같다.

$$\phi^{\#} = \alpha(\phi) = (x = +) \wedge (y = +)$$

요약 슬라이싱 기준 $C^{\#}$과 실제 슬라이싱 기준 C 사이의 연결관계는 Galois 연결 (α, γ)에 의해서 설정된다. 따라서 $\phi \sqsubseteq \gamma \circ \alpha(\phi)$이 성립함을 알 수 있다. 이는 $C^{\#}$가 실제 슬라이싱 기준 C에 대한 안전한 근사 기준임을 의미한다. 결과적으로 $C^{\#}$에 대한 프로그램 P의 (요약) 슬라이스는 변수들 V에 대해서 C에 대한 P의 실제 슬라이스의 모든 가능한 실행 결과를 포함해야 한다.

이제는 요약 프로그램 슬라이스의 개념을 정의한다. 먼저 필요한 몇 개의 표기법을 살펴보자. $p_0, p_1, \cdots, p_k$는 프로그램 P내의 프로그램 포인트를 나타낸다. $\sigma_i \in State$는 p_i를 실행하기 직전의 상태를 나타내며 $\sigma_i|V$는 변수들 V에 제한된 σ_i를 나타낸다. 또한 슬라이스의 성질을 설명하는데 필요한 *상태 궤적*(state trajectory)과 프로젝션 함수를 소개한다[14] .

정의 3 초기 상태 σ에 대한 프로그램의 상태 궤적은 순서쌍들의 유한 시퀀스 $\tau = \langle (p_0, \sigma_0), (p_1, \sigma_1), \cdots, (p_k, \sigma_k)\rangle$이며 여기서 $p_1, \sigma_1, \cdots, p_k$는 프로그램 실행 경로이다.

정의 4 $\tau = \langle \tau_0, \tau_1, \cdots, \tau_k\rangle$이고 $\tau_i (1 \leq i \leq k)$는 시퀀스 τ의 I-번재 순서쌍 (p_i, σ_i) 라고 가정한다. $Proj_{(p,V)}(\tau_i)$는 다음과 같이 정의된다.

$$Proj_{(p,V)}(\tau_i) = \begin{cases} \lambda & \text{if } p_i \neq p \\ \langle (p_i, \sigma_i|V)\rangle & \text{if } p_i = p \end{cases}$$

정의 5 $Proj$는 $Proj'$를 전체 궤적에 확장한 것으로 정의된다:

$$Proj_{(p,V)}(\tau) = Proj'_{(p,V)}(\tau_0) \cdots Proj'_{(p,V)}(\tau_k)$$

이제 상태 궤적과 슬라이싱 기준 C의 초기 조건 ϕ사이의 관계를 살펴보자. 초기 요약 조건 $\alpha(\phi)$는 요약 상태 $\sigma^{\#}$에 의해서 특징지어진다고 가정한다. 각 요약 상태 $\sigma^{\#}$는 실제화 함수 $\gamma_\sigma : AbsState \rightarrow 2^{(State)}$를 통해서 $State$ 내의 실제 상태들의 집합에 대응된다. 따라서 $\sigma \in \gamma_\sigma(\sigma^{\#})$인 각 상태 σ는 상태 궤적 τ를 식별한다. 요약 슬라이스는 프로그램 문장들의 부분집합으로 각 궤적에 대해서 원래 동작을 보존해야 한다.

정의 6 프로그램 P'는 프로그램 P로부터 0개 혹은 그 이상의 문장을 제거한 P일부로 $P' \leq P$로 표시된다고 하자. 슬라이싱 기준 $C = (\phi, p, V)$에 대한 프로그램 P의 요약 슬라이스는 $P' \leq P$인 실행 가능한 프로그램 P'으로 P가 $\sigma \in \gamma \circ \alpha(\phi)$ 인 초기 상태 σ에서 상태 궤적 τ에 대해서 끝날 때마다 P'도 σ에서 $Proj_{(p,V)}(\tau) = Proj_{(p,V)}(\tau')$ 인 상태 궤적 τ'에 대해서 끝나야 한다.

프로그램 슬라이싱의 핵심 요소의 하나가 *안전성*(safety)이다. 즉 요약 슬라이스는 슬라이싱 기준 C에 대해서 원래 프로그램의 동작을 보존함을 보장해야 한다. ''안전성'' 성질은 정리 2에서처럼 슬라이싱 기준 C에 대해서 프로그램 P와 슬라이스 P' 사이의 모음 시맨틱스 관계로 설명할 수 있다. 정리 2는 정의 6과 정리 1의 요약 해석의 안전성으로부터 쉽게 증명될 수 있다.

정리 2 슬라이싱 기준 $C = (\phi, p_i, V)$에 대해서 P'를 프로그램 P의 요약 슬라이스라고 하자. 초기 조건 ϕ에 대해서 $(p_i, \sigma_i) \in lfp(F_P)$이면 $\sigma_i|V = \sigma'|V$인 $(p_i, \sigma') \in lfp(F_{P'})$가 존재한다.

프로그램 P로부터 요약 슬라이스를 추출하기 위해서 P의 프로그램 포인트들 사이의 제어 및 자료 종속성(control and data dependency) 관계를 파악해야 한다[6]. 이를 위해서 요약 해석을 통해서 얻어진 프로그램의 동적 성질을 이용하여 정적 슬라이스에 사용된 것보다 정확한 종속성을 구한다. 요약 해석을 통한 동적 성질을 이용하면 주어진 초기 조건에 맞는 초기 상태들 하에서 어떤 프로그램 블록이 실행되지 않는지 파악할 수 있다. 종속성을 파악할 때 주어진 초기 상태들 하에서 실행되지 않는 문장들은 무시해도 된다. 실행되지 않는 문장들은 프로그램 P의 요약 시맨틱스를 계산함으로써 파악할 수 있다.

예를 들어 그림 2의 예제 프로그램을 살펴보자. 다음과 같이 정의된 간격(interval) 요약 도메인을 $(L_{int}, \sqsubseteq_{int})$라고 하자 :

$$L^{int} = \{\bot\} \cup \{[l,u] \mid l \in Integer \cup \{-\infty\} \wedge u \in Integer \cup \{+\infty\} \wedge l \leq u\}$$

그림 2의 프로그램에서 변수 ''i''와 ''n''에 대해서는 요약 도메인 L^{int}를 사용하고 다른 변수들에 대해서는 L^{sign}를 사용한다. 프로그램 변수 전체를 위한 요약 도메인은 [4]에서처럼 위의 요약 도메인들을 요소별 요약

```
int n,i,a[100],sum,asum,psum,nsum, pos,neg,zero;

1    read(n);
2    read(a);
3    sum = asum = psum = nsum = pos = neg = zero = 0;
4    i = 1;
5    while (i <= n) do
6        if (a[i] > 0) then
7                psum = psum + a[i];
8                pos = pos + 1;
9        else
10               if (a[i] < 0) then
11                       nsum = nsum + a[i];
12                       neg = neg+1;
13               else
14                       if (a[i] == 0) then
15                               zero = zero + 1;
16                       if (psum > -nsum) then
17                               sum = psum+nsum;
18                       else
19                               sum = 0;
20                       fi
21               fi
22       fi
23       fi
24       i = i +1;
25   end
26   if (neg ==0) then
27               sum = asum = psum;
28   else
29               sum = psum + nsum;
30               asum = psum - nsum;
31   fi
32   print(sum);
33   print(asum);
```

그림 2 슬라이싱을 위한 예제 프로그램

을 통해서 합성함으로써 정의할 수 있다. 각 프로그램 포인트에 대해서 요약 상태를 최소고정점 등식 시스템 $\overline{X}=F_P^{\#}(\overline{X})$를 통해서 연관시킬 수 있다. 프로그램 변수들 〈n, i, a, sum, asum, psum, nsum, pos, neg, zero〉에 대한 등식 시스템은 다음과 같다.

$$\left\{\begin{array}{l} X_1=\langle \top,\top,\cdots,\top\rangle \\ X_2=[read(n)]^{\#}(X_1) \\ X_3=[read(a)]^{\#}(X_2) \\ X_4=\langle X_3[n],X_3[i],X_3[a],0,0,0,0,0,0,0\rangle \\ X_5=[i=1]^{\#}(X_4)\sqcup X_{25} \\ X_6=[i\le n]^{\#}(X_5)\ \ but\ X_6[i]= \\ \qquad [i\le n]^{\#}(X_5)[i]\sqcap[-\infty,\max(X_5[n])] \\ \qquad\qquad \cdots \end{array}\right)$$

여기서 예를 들어 $[i\le n]^{\#}$는 다음을 만족하는 요약 테스트를 나타낸다 :

$$[i\le n]^{\#}(X)\supseteq\alpha_\sigma(\{\sigma\in\gamma_\sigma(X)\mid\sigma(i)\le\sigma(n)\})$$

요약 해석은 시맨틱 함수가 단조 함수이고 요약 도메인이 유한하거나 상승 체인 유한(ascending chain finite)하면 유한 시간에 계산 가능하다. 그렇지 않으면 무한 계산을 포함할 수 있다. 이러한 경우에 필요하면 유한 계산을 위해서 확장(widening)과 같은 속도 증가 기법을 사용할 수 있다. $\triangledown$로 표시되는 확장 연산자(widening operator)의 중요한 성질의 하나는 $\forall x,y\in D^{\#}$, $x\triangledown y\supseteq x\sqcup y$이라는 것이다. 즉 확장 연산자는 $x\sqcup y$에 대한 안전한 근사치를 제공한다. 따라서 모든 상승 체인에 대해서 확장 연산자를 이용하여 결국 끝나는 체인을 정의할 수 있다. 예를 들어 확장 연산자를 X_5의 등식에 적용하면 다음과 같이 될 것이다.

$$X_5=X_5\triangledown([i=1]^{\#}(X_4)\sqcup X_{25})$$

다른 등식은 변하지 않는다. 이렇게 함으로써 원래 등식 시스템에 대한 보다 근사 해를 유한 시간에 구할 수 있다[3,4].

초기 조건 ϕ가 $\forall 1\le i\le n.a[i]>0$이면 이의 요약 $\phi^{\#}$는 $\forall 1\le i\le n.a[i]=+$ 이다. 이 조건 하에서 위에서 변경된 고정점 등식에 대한 해의 일부가 그림 3이다.

$$\begin{array}{l} X_1=\langle \top,\top,\cdots,\top\rangle \\ X_2=\langle[100,100],\top,\cdots,\top\rangle \\ X_3=\langle[100,100],\top,a,\top\cdots,\top\rangle\ where\ a=\langle+,\cdots,+\rangle \\ X_4=\langle[100,100],\top,a,0,0,0,0,0,0,0\rangle \\ X_5=\langle[100,100],[1,+\infty],a,0,0,0,0,0,0,0\rangle \\ X_6=\langle[100,100],[1,100],a,0,0,+,0,+,0,0\rangle \\ \qquad\qquad \cdots \end{array}$$

그림 3 등식 시스템의 해

일단 초기 조건 하에서 프로그램의 요약 시맨틱스를 계산하면 실행되지 않을 문장을 판별할 수 있으며 이들은 프로그램 슬라이싱 과정에서 고려하지 않을 것이다. 프로그램을 슬라이싱하기 위한 변환 규칙을 정의하기 위하여 몇 개의 정의를 도입한다. 불리언 제약 함수 $R[b]\sigma$는 불리언 조건이 참인 실제 상태를 판별하기 위해 사용되며 다음과 같이 정의된다.

$$R[b]\sigma=\left\{\begin{array}{ll}\sigma & if\ b\ is\ true\ in\ \sigma \\ \emptyset & otherwise\end{array}\right.$$

다음으로 요약 불리언 제약 함수 $R^{\#}[b]\sigma^{\#}$는 다음과 같이 정의된다.

$$R^{\#}[b]\sigma^{\#}=\alpha_\sigma\{R[b]\sigma\mid\sigma\in\gamma_\sigma(\sigma^{\#})\}$$

여기서 $\Sigma=\emptyset$이면 $\alpha_\sigma(\Sigma)=\perp_\sigma$이다. 요약 상태 $\sigma^{\#}$에 가능한 값이 없으면 결과 상태는 $\perp_\sigma$임을 유의해야 한다.

$\sigma^{\#}$를 프로그램 내의 문장 S를 실행하기 직전의 요약 상태라고 하면 다음 변환 규칙을 이용하여 프로그램 P로부터 보다 간단한 프로그램 P^T를 구할 수 있다.

- 규칙 1 S가 *if b then* S_1 *else* S_2 형태의 조건문이고 $R^{\#}[\neg b]\sigma^{\#} = \perp_{\sigma}$이면 $P[skip/S_2]$이 된다. 여기서 $P[S_i/S]$는 S를 S_i로 대치한 것을 의미하고 *skip*는 빈 문장을 나타낸다.
- 규칙 2 S가 *if b then* S_1 *else* S_2 형태의 조건문이고 $R^{\#}[b]\sigma^{\#} = \perp_{\sigma}$이면 $P[skip/S_1]$이 된다.
- 규칙 3 S가 *if b then* S_1 *else* S_2 형태의 조건문이고 $R^{\#}[\neg b]\sigma^{\#} \neq \perp_{\sigma}$이고 $R^{\#}[b]\sigma^{\#} \neq \perp_{\sigma}$이면 S는 변하지 않는다. 즉 $P[S/S]$이 된다.
- 규칙 4 S가 *while b do* S_1 *end* 형태의 while-루프이고 $R^{\#}[b]\sigma^{\#} = \perp_{\sigma}$이면 $P[skip/S_1]$이 된다.
- 규칙 5 S가 *while b do* S_1 *end* 형태의 while-루프이고 $R^{\#}[b]\sigma^{\#} \neq \perp_{\sigma}$이면 S는 변하지 않는다. 즉 $P[S/S]$이 된다.

위의 규칙들의 안전성은 정리 2로 설명 될 수 있다. $R^{\#}[x]^{\#} = \perp_{\sigma}$는 불리언 수식 x를 True로 만드는 상태가 주어진 초기 조건에 대해 존재하지 않는다는 사실을 의미한다 . 따라서, 정리 2로부터 x가 True가 될 때 실행되는 문장(들)을 제거해도 원래 프로그램의 시맨틱스에 영향을 주지 않는다는 것을 알 수 있다. 규칙 1은 불리언 수식 b가 요약 상태 $\sigma^{\#}$에서 항상 참이면 *else* 분기는 안전하게 제거 될수 있다는 것을 의미한다. 규칙 2도 비슷하게 설명할 수 있다. 규칙 3은 불리언 수식이 참 혹은 거짓이 될 수 있으면 이 조건문은 변하지 않는다는 것을 의미한다. 규칙 4와 규칙 5는 반복문에 관한 것으로 규칙 4는 불리언 식 b가 참이면 루프의 본체를 제거할 수 있고 그렇지 않으면 규칙 5에 의해서 변하지 않는다.

이 규칙들을 그림 2의 프로그램에 초기 조건 $\forall 1 \le i \le n.\, a[i] > 0$과 $n = 100$ 하에서 적용해 보자. 그림 3는 그림 2의 프로그램에서 각 문장 실행 직전의 요약 상태를 나타내고 있다. 5번째 줄에 있는 프레디킷트의 실행 직전에 요약 상태 $\sigma_5^{\#}$를 나타내는 X_5를 고려해보자. $\sigma_5^{\#}(i) = [1, +\infty]$이고 $\sigma_5^{\#}(n) = [100, 100]$이므로 두 결과가 가능하다. 이는 규칙5에 따라 루프 본체를 제거할 수 없다는 것을 의미한다.

그러나 $\sigma_6^{\#}$는 $[\langle [100,100], [1,100], a, 0, 0, +, 0, +, 0, 0\rangle]$으로 이는 6번째 줄의 프레디킷트는 항상 참임을 의미한다. 따라서 *else* 분기는 규칙1에 따라 제거할 수 있다. 비슷하게 26번째 줄의 *else* 분기는 변수 ''neg''가 요약 상태 $\sigma_{26}^{\#}$에서 0이므로 절대 실행되지 않는다. 따라서 규칙1를 적용하여 29번째 줄과 30번째 줄을 제거할 수 있다.

3.2 요약 의존성 그래프

프로그램 P의 요약 의존성 그래프 $G_P^{\#}$는 변환 규칙을 적용하여 P로부터 구한 프로그램 P^T에 대해서 정의된 프로그램 의존성 그래프이다. $G_P^{\#}$의 노드는 P^T에 나타나는 배정문 프레디킷트와 같은 프로그램 구성요소들이다. 전통적인 프로그램 의존성 그래프[6]처럼 $G_P^{\#}$의 에지(edge)는 제어 의존성과 데이타 의존성을 나타낸다.

u부터 v로의 제어 의존성 에지를 $u \rightarrow_{c^{\#}} v$로 나타내고 자료 의존성 에지를 $u \rightarrow_{d^{\#}} v$로 나타낸다. 다음 중의 하나를 만족하면 $G_P^{\#}$ 내에 u부터 v로의 제어 의존성 에지가 존재한다.

1. u가 출입 vertex이고 v는 루프 혹은 조건문에 중첩되어 있는 P^T의 컴포넌트를 나타낸다. 이러한 에지는 true 레이블을 붙인다.

2. u가 제어 프레디킷트를 나타내고 v는 루프 혹은 조건문에 바로 중첩되어 있는 프레디킷트가 u인 P^T의 컴포넌트를 나타낸다. u가 while-루프의 프레디킷트이면 $u \rightarrow_{c^{\#}} v$ 에지에 true 레이블을 붙인다. u가 조건문의 프레디킷트이면 v가 then 분기 혹은 else 분기에 있느냐에 따라 $u \rightarrow_{c^{\#}} v$ 에지에 true 레이블 혹은 false 레이블을 붙인다.

$G_P^{\#}$에서 데이타 종속성의 정의는 전통적인 프로그램 의존성 그래프 [6]에서 사용된 것과 차이가 없다. $G_P^{\#}$에서 두 노드 사이의 $u \rightarrow_{d^{\#}} v$와 같은 데이타 의존 에지는 u로 부터 v로의 데이타 의존성을 나타낸다. 그렇지만 역이 반드시 성립하는 것은 아니다. 이를 명확히 하기 위해 다음 데이타 흐름 개념을 살펴보자.

$D(p_i)$는 p_i에서 l-값(l-value)이 사용된 변수들의 집합을 나타내고 $U(p_i)$는 p_i에서 r-값(r-value)이 사용된 변수들의 집합을 나타낸다. 이들은 어느 정도 동적인 성질을 지녔다는 점에서 정적인 정의와는 약간 다르다. 예를 들어 다음 프로그램 일부를 살펴보자.

```
        ...
p_l1:  a[i] = a[j] * k;
p_l2:  i = i + 2;
p_l3:  z = a[i];
        ...
```

$\sigma_{l_1}^{\#}(i)$는 [2..3] 이고 $\sigma_{l_1}^{\#}(j)$는 [3..5]라고 가정하면 다음 변수들은 사용되고 정의된다.

$D(p_{l_1}) = \{a[2], a[3]\}$ $\quad U(p_{l_1}) = \{a[3], a[4], a[5], j, k\}$

비슷하게 p_{l_2}에서 배정문의 효과를 고려하여 $D(p_{l_2})$과

$U(p_{l_1})$를 계산할 수 있다.

$$D(p_{l_3})=\{z\} \quad U(p_{l_4})=\{a[4],a[5],i\}$$

반면 전통적인 정적 슬라이싱은 전체 배열을 하나의 변수로 취급하는데 이는 정적 슬라이싱이 배열의 특정 원소에 대한 정보를 고려할 수 없기 때문이다. 결과적으로 슬라이스는 불필요하게 커진다. 정적 슬라이싱에서 사용된 의존성은 모든 가능한 입력 상태에 대해서 정의되는 반면에 요약 슬라이싱은 명시된 입력 상태에 대해서만 의존성을 정의한다는 점을 주의해야 한다. 또한 요약 해석을 이용하면 프로그램의 각 지점에서 사용되거나 갱신된 배열의 원소를 결정할 수 있다. 예를 들어 전통적인 정적 분석은 위의 프로그램에 대해서 다음과 같이 의존성을 파악한다.

$$D(p_{l_1})=\{a\},\ U(p_{l_1})=\{a,\ j,\ k\},\ D(p_{l_3})=\{z\},\ U(p_{l_3})=\{a,i\}$$

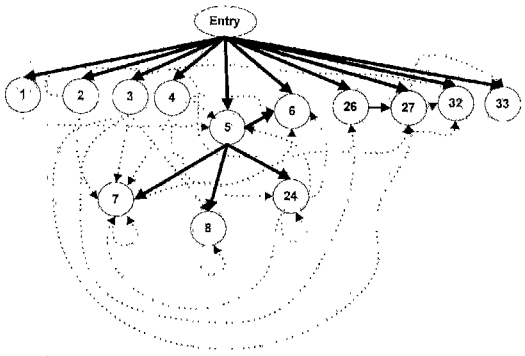

그림 4 요약 프로그램 의존성 그래프 예 : 실선은 제어 의존성을 점선은 데이타 의존성을 나타낸다.

p_{l_1}로부터 p_{l_3}으로의 데이타 의존성은 정적 프로그램 의존성 그래프에는 표현되나 요약 프로그램 의존성 그래프에는 나타나지 않는다. 그러나 이 프로그램 요소가 루프 내에 있고 루프 인덱스 변수에 확장 연산이 적용되면 이 정확성은 줄어든다.

이 논의로부터 슬라이싱 기준 $C=(\phi,p,V)$ 에 대한 프로그램 P의 요약 슬라이스는 $v\rightarrow^{*}_{c',d'}p$ 관계를 만족하는 $G_P^{\sharp}$의 노드 v에 대응하는 문장들의 집합으로 정의할 수 있다. 그림 4은 그림 2의 프로그램에 변환 규칙을 적용하여 얻어진 프로그램의 요약 프로그램 의존성 그래프를 보여주고 있다.

그림 5는 $\phi=\forall i:\ 1\le i\le n, a[i]>0$이고 n이 100인 슬라이싱 기준 $C=(\phi,32,\{sum\})$ 하에서 요약 슬라이스를 보여주고 있다. 이렇게 구해진 요약 슬라이스는 슬라이싱 기준 $(\forall i:\ 1\le i\le n,\ a[i]=1,32,\{sum\})$ 하에서 구해진 동적 슬라이스와 일치한다. 또한 6번째 줄과 26번째 줄의 프레디킷트가 항상 참이라는 것을 이용하여 슬라이스의 크기를 더 줄일 수 있다.

```
1       read(n);
2       read(a);
3       sum = psum = neq = 0;
4       i = 1;
5       while (i <= n) do
6             if (a[i] >0) then
7                   psum = psum + a[i];
23            fi
24                  i = i +1;
25      end
26      if (neq = 0) then
27                        sum   = psum;
31      fi
32      print(sum);
```

그림 5 요약 슬라이스 예

4. 관련 연구

정적 슬라이싱 [14], 동적 슬라이싱 [1,9], 의사-정적 슬라이싱(quasi-static slicing)[15], 조건 슬라이싱(conditioned slicing) [11] 등 몇 가지 프로그램 슬라이싱 기법과 응용에 대한 자세한 설명은 [12]에 있다. 이 절에서는 요약 슬라이싱이 다른 슬라이싱 방법들을 비교 설명할 것이다.

정적 슬라이싱은 모든 가능한 실행에 대해서 원래 프로그램의 동작을 보존하는 프로그램 슬라이스를 위한 모델이라고 할 수 있다. 요약 슬라이싱 방법도 슬라이싱 기준 $C=(\phi,p_i,V)$를 $\alpha(\phi)=\forall v_{i(i=1,\ldots,n)}\in V_{in}: v_i=\top$로 함으로써 정적 슬라이스를 생성할 수 있다. 여기서 $v_i=\top$는 입력 변수 v_i가 아무 값이나 가질 수 있다는 것을 의미한다. 이렇게 얻은 요약 슬라이스는 대응되는 정적 슬라이스의 모든 프로그램 수행을 포함하는 것을 보장할 수 있다. 한편 일반적인 요약 슬라이스는 배열의 원소 사이의 데이타 종속성 등을 보다 자세히 함으로써 결과 슬라이스의 크기를 정적 슬라이스보다 줄일 수 있다.

동적 슬라이스는 특정 프로그램 궤적 상에서 관심있는 변수에 영향를 미치는 문장들만을 포함하므로 슬라이스의 크기는 줄일 수 있으나 그 응용은 프로그램 디버깅과 같은 분야에만 한정된다. 요약 슬라이스는 특정

궤적의 시작 상태를 요약함으로써 동적 슬라이스에 대한 안전한 근사 슬라이스를 제공한다. 이 경우에 요약 슬라이스는 동적 슬라이스보다 크기는 커질 수 있으나 하나의 초기 상태가 아니라 초기 상태들의 집합을 고려할 수 있게 된다.

의사-정적 슬라이싱은 정적 슬라이싱과 동적 슬라이싱 사이에 있는 방법으로 어떤 입력은 고정되어 있고 다른 입력 값은 변할 때 프로그램의 어떤 점에서 변수 값에 영향을 미치는 문장들을 추출한다. 의사-정적 슬라이싱은 모든 입력 값이 변할 때 정적 슬라이싱과 일치한다. 동적 슬라이싱의 경우처럼 의사-정적 슬라이싱은 요약 슬라이싱을 통해서 근사치를 구할 수 있다. 예를 들어 변수 프로그램 내의 p 지점에서 하나의 입력 변수 v_1만 5로 고정된 상태에서 변수 out에 대한 의사-정적 슬라이싱을 구한다고 하자. 이에 대한 근사치를 요약 슬라이싱 기준 $C^{\#}=(\alpha(v_1=5), p, \{out\})$으로 함으로써 요약 슬라이싱을 통해서 구할 수 있다.

또 하나의 흥미로운 슬라이싱은 조건 슬라이싱으로 요약 슬라이싱과 비슷하다. 큰 차이는 슬라이스를 계산하는 방법에 있다. 조건 슬라이싱은 입력 조건에 대한 명세를 필요로 하고 실행되지 않는 블록을 찾기 위해 심볼 실행(symbolic execution)을 이용한다. 그러나 루프의 심볼 실행에는 문제가 있어서 조건 슬라이싱에서는 루프의 반복 횟수가 수작업을 통해서 계산되여 제공됨을 가정하고 있다. 이에 반해 요약 슬라이싱은 실행되지 않는 블록을 찾는데 있어서 수작업 혹은 사용자와의 상호작용 없이 자동적으로 할 수 있다. 또한 조건 슬라이싱은 단지 정적 슬라이싱의 데이타 의존성 개념을 사용하는 반면에 요약 슬라이싱은 프로그램 각 지점에서 요약 시맨틱스를 고려하여 데이타 의존성을 개선한다.

5. 결 론

이 논문에서는 요약 해석을 이용하여 초기 상태들의 집합에 대해서 프로그램을 슬라이싱하는 새로운 슬라이싱 기법을 제시하였다. 결과적으로 얻은 요약 슬라이스는 주어진 입력 상태들의 집합에 대해서 정적으로 계산되었다는 점에서 동적 슬라이스와는 다르며 배열과 같은 자료구조를 정적 슬라이싱보다 정확하게 다룰 수 있으며 슬라이스 크기도 줄일 수 있다는 장점이 있다.

요약 슬라이스는 새로운 개념이고 전통적인 슬라이싱 기법에 약점을 개선하는데 기여하고 있으나 몇 가지 향후 연구가 필요하다. 이론적인 면에서 요약 슬라이스의 몇 가지 변종을 연구하고 그들의 특징 및 그들 사이의 관계를 정리할 필요가 있다. 예를 들어 요약 슬라이스는 명세에 나타난 정보를 완전히 이용함으로써 보다 정확한 슬라이스를 구할 수 있다. 본 연구에서는 초기 상태를 나타내는 전조건(precondition)만을 이용하였으나 추출하고자 하는 구성요소의 특징을 나타내는 후조건(postcondition)도 고려할 수 있을 것이다. 이 경우에는 후조건을 슬라이싱 기준으로 대응시키는 일이 필요하며 이는 역방향 시맨틱 분석을 적용하여 역방향으로 후조건을 전달해야 할 것이다. 또 하나의 중요한 연구는 요약 슬라이싱을 재구조성, 프로그램 이해, 재사용등의 분야에 적용하는 것이다.

참 고 문 헌

[1] H. Agrawal and J. Horgan, "Dynamic program slicing," In *Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pp.246-256, 1990.

[2] J. Bieman and B. K. Kang, "Measuring design-level cohesion," *IEEE Trans. on Software Engineering*, vol. 24, no. 2, 1998, pp.111-124.

[3] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. of 4th ACM Symp. on Principle of Programming Languages*, pp.238-252, ACM Press, 1977.

[4] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proc. of 4th ACM Symp. on Principle of Programming Languages*, pp.269-282, ACM Press, 1979.

[5] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proc. of Conf. on Software Maintenance*, Orlando, Florida, U. S. A., IEEE CS Press, 1992, pp.299-308.

[6] S. Horwitz, T.Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 1, pp.35-46, Jan. 1990.

[7] M. Kamkar, "Interprocedural dynamic slicing with applications to debugging and testing," *PhD Thesis*, Linkoping University, 1993.

[8] H. S. Kim, Y. R. Kwon, and I. S. Chung, "Restructuring programs through program slicing," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 3, pp.349-368, 1994.

[9] B. Korel and J. Laski, "Dynamic program slicing,"

Information Processing Letters, vol. 29, no. 3, pp. 155-163, 1988.

[10] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow-graph based program slicing," *IEEE Trans. on Software Engineering*, vol. 23, no. 4, pp.246-259, April 1997.

[11] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," in *Proc of 4th Workshop on Program Comprehension*, Berlin, Germany, IEEE CS Press, 1996, pp.9-18.

[12] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, 1995, pp.121-189.

[13] M. Weiser, "Programmers use slices when debugging," *Comm. of the ACM*, vol. 25, 1982, pp. 446-452.

[14] M. Weiser, "Program slicing," *IEEE Trans. on Software Engineering*, vol. 10, no. 4, pp.352-357, July 1984.

[15] G. Venkatesh, "The Semantic Approach to Program Slicing," In *Proc. of the ACM SIGPLAN91 Conf. on Programming Languages Design and Implementation*, pp.26-28, June 1991.

정 인 상

1987년 서울대학교 컴퓨터공학과 학사. 1989년 한국과학기술원 전산학과 석사. 1993년 한국과학기술원 전산학과 박사. 1993년 9월 ~ 1994년 2월 한국전자통신연구원 박사후연수연구원. 1995년 7월 ~ 1995년 5월 8일 영국 Durham 대학 Centre for Software Maintenance 방문연구원. 1994년 3월 ~ 1999년 2월 한림대학교 교수. 1997년 8월 ~ 1998년 7월 미국 Purdue대학 SERC 방문교수. 1999년 3월 ~ 현재 한성대학교 정보전산학부 교수. 관심분야는 소프트웨어 테스팅, CBSE, Formal Specification Techniques

창 병 모

정보과학회논문지 : 소프트웨어 및 응용
제 28 권 제 6 호 참조