명세 기반 프로그램 슬라이싱 기법과 응용
(Specification-based Program Slicing and Its Applications)

정 인 상 † 윤 광 식 †† 이 완 권 ††† 권 용 래 ††††
(In Sang Chung) (Gwang Sik Yoon) (Wan Kwon Lee) (Yong Rae Kwon)

요 약 기존의 프로그램 슬라이스에 관한 정의들은 주로 프로그램 변수들간의 문법적인 관계만을 고려한다. 이에 반해, 이 논문에서는 프로그램 변수들간의 의미론적인 관계를 고려함으로써 기존의 슬라이스보다 더 정확한 프로그램 슬라이스를 구할 수 있는 명세 기반 슬라이싱 기법을 제시한다. 명세 기반 슬라이스는 선행 조건, 후행 조건의 쌍으로 주어진 프로그램의 명세에 대해 원래 프로그램의 행위와 올바름을 보존하는 프로그램 문장들의 부분집합으로 구성된다. 명세 기반 슬라이싱 기법은 주어진 명세에 나타나 있는 프로그램의 기능과 관련한 프로그램 문장만을 다루기 때문에, 소프트웨어 공학의 여러 가지 문제들을 더욱 효과적으로 해결할 수 있도록 지원한다. 여러 적용 가능한 문제들 중에서도, 이 논문에서는 소프트웨어 재사용과 소프트웨어 재구성 과정이 명세 기반 슬라이싱 기법을 이용함으로써 어떻게 향상될 수 있는지를 보인다.

키워드 : 소프트웨어 공학, 프로그램 슬라이싱, 소프트웨어 재사용, 소프트웨어 재구성

Abstract More precise program slices could be obtained by considering the semantic relations between variables of interest, compared to the existing slicing techniques considering only the syntactic relations. In this paper, we present specification-based slicing that allows a better decomposition of the program by taking a specification as its slicing criterion. A specification-based slice consists of a subset of program statements which preserve the behavior and the correctness of the original program with respect to a specification given by a pre-postcondition pair. Because specification-based slicing enables one to focus attention on only those program statements which realize the functional abstraction specified by the given specification, it can be widely used in many software engineering areas. Of its possible applications, we show how specification-based slicing can improve the process for extracting reusable parts from existing programs and restructuring complex programs for better maintainability.

Key words : Software Engineering, Program Slicing, Software Reuse, Software Restructuring

1. 서 론

프로그램 슬라이싱 기법(program slicing technique)은 Weiser 에 의해 제안된 이후 소프트웨어 공학의 여러가지 문제들에 대한 해결책으로 이용되어 왔다. 프로그램 슬라이싱은 슬라이싱 기준(slicing criterion) (p, V)에 대하여 행해진다. 이때, p는 프로그램의 위치이며, V는 프로그램 변수들의 부분집합이다. 프로그램 슬라이싱의 가장 중요한 목적은 슬라이싱 기준에 포함되어 있는 변수들의 값에 직접적 또는 간접적으로 영향을 미치는 프로그램 문장들을 추출하는 것이다[1].

슬라이싱 기법은 소프트웨어 공학의 많은 문제들의 해결에 도움을 줄 수 있다. 예를 들어, 명세를 만족하는 기존의 소프트웨어로부터 재사용 가능한 컴포넌트를 추출하는 문제를 살펴보자. 프로그램 슬라이싱은 재사용 가능한 컴포넌트를 추출하는 과정 중에 많은 문장들을 없앨 수 있는 가능성을 제공하기 때문에 재사용(software reuse)에 유용한 기법이다[2,3,4,5,6]. 기존의 시스템으로부터 재사용 가능한 컴포넌트를 추출하기 위해서

· 이 논문은 1999년 한국학술진흥재단의 연구비에 의하여 지원되었음 (KRF-99-E00280)
† 종신회원 : 한성대학교 컴퓨터공학부 교수
insang@hansung.ac.kr
†† 비 회 원 : 매크로임팩트(주) 시스템소프트웨어연구소
yoon@macroimpact.com
††† 비 회 원 : 전주대학교 정보기술컴퓨터공학부 교수
wklee@jeonju.ac.kr
†††† 종신회원 : 한국과학기술원 전산학과 교수
kwon@salmosa.kaist.ac.kr
논문접수 : 2000년 6월 9일
심사완료 : 2002년 6월 18일

는 *완전성*(completeness)과 *정확성*(accuracy)을 고려하여야 한다[5]. 완전성은 추출된 컴포넌트가 요구되는 기능을 구현하는 데 필요한 모든 문장을 포함하고 있어야 한다는 조건이며, 정확성은 추출된 컴포넌트에 요구되는 기능을 수행하는 데 필요하지 않은 문장들을 포함하고 있지 않아야 한다는 조건이다. 프로그램 슬라이싱을 이용한 간단한 소프트웨어 재사용 기법은 슬라이싱 기준의 V에 명세에 이용된 모든 변수를 포함시키고, 이 기준에 대해 슬라이스를 구하는 것이다. 하지만, 이렇게 구해진 프로그램 슬라이스는 필요이상으로 많은 문장을 포함하고 있는 컴포넌트로써 재사용시에 요구되는 *정확성*이 충분치 못한 경우가 많다. 그 이유는 기존의 프로그램 슬라이싱 기법들이 명세에 나타나 있는 변수들간의 의미론적 관계를 무시하거나, 또는 부분적으로만 이용하기 때문이다.

소프트웨어 재사용외에도, 프로그램 슬라이싱 기법은 복잡한 소프트웨어를 재구성(software restructuring)하기 위해 이용될 수 있다. 어떤 프로그램이 실제 적용되는 현장에서는 많은 유지보수를 통해 프로그램이 몇십년동안 이용되는 경우도 있다. 하지만 프로그램이 유지보수를 거치는 동안 프로그램의 크기가 증가하며, 구조가 불건전해진다. 따라서 프로그램을 이해하기도 어려워진다. 이런 문제를 해결하기 위해 유지보수과정중에 소프트웨어 재구성이 필요하다. 시스템의 구조는 각 모듈의 응집도(cohesion)를 최대화하고 모듈간의 결합도(coupling)를 최소화함으로써 향상될 수 있다. 재구성은 주로, 프로그램 슬라이싱 기법을 이용하여 프로그램의 모듈중 응집도가 낮은 모듈들을 분별하고, 이 모듈들을 응집도가 높은 모듈로 변환하는 과정을 통해 이루어진다[4]. 하지만 기존의 슬라이싱 기법은 프로그램의 의미보다는 프로그램의 문법에 기반한 기법임으로, 변경된 모듈들이 사용자의 관점에서도 의미있는 방식으로 변경된다는 것을 보장하지 못한다.

이러한 한계를 극복하기 위해, 이 논문에서는 슬라이싱 기준에 명세(specification) C를 포함시킨 명세 기반 슬라이스(specification-based slice)를 정의한다. 명세 C는 선행-후행 조건 쌍 (P,Q)로 표현된다. 선행 조건 P는 입력 변수들에 관한 술어(predicate)로써 입력 값들에 대한 가정을 표현한다. 후행 조건 Q는 출력 변수들에 관한 술어로써 슬라이싱이 목표로 하는 컴포넌트의 행위에 대한 요구사항들을 표현한다. 명세 기반 슬라이싱의 가장 주된 목적은 선행-후행 조건 쌍 (P,Q)가 주어졌을 때, 원래 프로그램으로부터 S의 수행 전에 P가 만족했다면, S의 수행 후에 Q가 만족되는 그런 프로그램 S를 추출하는 것이다.

이 논문에서는 최약 선행 조건(weakest precondition)과 최강 후행 조건(strongest postcondition)의 개념을 이용하여 명세 기반 슬라이스를 도출한다.

이 논문의 구성은 다음과 같다. 먼저 2절에서는 명세 기반 슬라이싱을 정의하는 데 중요한 역할을 하는 최약 선행 조건과 최강 후행 조건에 대해 알아보며, 3절에서는 명세 기반 슬라이싱의 다양한 정의 및 이러한 명세 기반 슬라이스를 유도할 수 있는 규칙들을 서술한다. 4절에서는 3절의 내용을 확장하여 프로시져 호출을 고려한 명세 기반 슬라이싱 기법을 제시한다. 5절에서는 소프트웨어 재사용과 소프트웨어 재구성에 명세 기반 슬라이싱 기법이 어떻게 이용될 수 있는지 그 예를 보인다. 6절에서는 이 논문에서 제안하고 있는 슬라이싱 기법과 기존의 프로그램 슬라이싱 기법과의 관계를 살펴보며, 7절에서는 결론 및 향후 연구방향을 제시한다.

2. 최약 선행 조건과 최강 후행 조건

프로그램의 상태는 var_i를 프로그램 변수, val_i 를 해당 변수의 값이라고 할 때, $\{(var_0, val_0), (var_1, val_1), \ldots\}$ 라고 정의될 수 있다. P와 Q를 프로그램 σ의 상태에 대한 술어(predicate)라고 하자. σ와 Q에 관한 최약 선행 조건($wp(\sigma, Q)$)이 P라는 것(" $wp(\sigma, Q) = P$")은 프로그램이 P를 만족하는 상태에서 실행되기 시작한다면, 프로그램이 종료된다는 것과 프로그램 종료시의 상태가 Q를 만족한다는 것이 보장된다는 것을 뜻한다.

예를 들어, 프로그램 σ가 다음과 같이 하나의 문장[1] "if x ≥ y then z:=x else z:=y fi"로 구성되어 있다고 하자. 이때 후행 조건 Q가 "z=y"라면 $wp(S, Q) = y \geq x$ 이다. 이 사실로부터, 조건문의 실행이 y ≥ x를 만족하는 프로그램 상태에서 시작되었다면, 주어진 조건 문이 반드시 종료하며, 종료시의 프로그램 상태는 "z=y"가 되리라는 것을 확신할 수 있게 된다.

문장 S와 선행 조건 P가 주어졌을 때, 최강 후행 조건 $sp(S, P)$는 프로그램의 실행이 P를 만족하는 상태에서 시작하고 종료한 경우, 최종 프로그램 상태가 만족하는 술어들 중 가장 강한 술어이다. 후행 조건 P가 가장 강하다는 것은 프로그램 수행후의 상태에 대해 참(true)으로 평가되는 모든 술어 Q에 대해, $P \Rightarrow Q$라는 것을 뜻한다. 최강 후행 조건은 최약 선행 조건과는 달리 전향적(forward)이며 개방적(liberal)이다. 전향적이라는 의

1) 이 논문에서는 특별히 혼동할 염려가 없는 한 "프로그램"과 "문장"을 같은 뜻으로 이용한다.

미는 프로그램의 수행이 된 후의 상태로부터 수행이 시작하기 전의 상태를 추론하는 최약 선행 조건과는 추론의 방향이 반대라는 의미이다. 또, 개방적이라는 조건은 최약 선행 조건에서는 보장되는 프로그램의 종료가 보장되지 않고 가정된다는 의미이다[7].

이 논문에서는 다음과 같은 문장들로 이루어진 프로그램에 대해 최약 선행 조건과 최강 후행 조건을 적용한다.

S :: = skip (무행위)
| abort (비정상적 프로그램 종료)
| <variable> := <expression> (대입문)
| $S_1;S_2$ (순차적 문장 결합)
| if B then S_1 else S_2 fi (조건문)
| do B S od (반복문)

이때 B는 부울대수적 표현이며, S_1, S_2, S는 프로그램 문장이다. 또, 논의의 편의를 위해, <variable>과 그에 상응하는 <expression>은 서로 호환가능한(compatible) 타입이라고 가정하며 다른 표현문들도 모두 적법한 형태를 갖는다고 가정한다. 또, "do B S od"의 의미는 $\mu X.$ if B then $S;X$ else *skip* fi이다. $\mu X.S$는 S의 실행중 X가 실행될 시점이 될 때마다, X가 $S(X)$로 치환되며 치환된 결과에 대해 실행이 계속됨을 뜻한다[8].

이 문장들은 논의의 편의를 위해 고안된 간단한 프로그래밍 언어로써 이 문장들에는 실제 프로그래밍 언어의 여러 가지 복잡한 특성들, 즉 임의의 제어 흐름, 배열, 새로운 정의를 허용하는 중첩 구조, 프로시져(procedure) 호출 등이 결여되어 있다. 하지만, 프로그래밍에 필수적인 요건들을 모두 포함하고 있으며, 명세 기반 슬라이싱의 유용성을 보이는 데 충분하다. 또, 이 논문의 후반부에 프로시져간의(interprocedural) 슬라이싱을 정의하기 위해 위 문장들에 프로시져 호출을 추가한다.

X를 포함하는 공식 $S(X)$의 문법적 축약(syntactic approximations)을 $S^0(X)=X, S^{n+1}(X)=S(S^n(X))$, 단, $n=0,1,\ldots,$.라고 하면 위 프로그램 문장들에 대한 최약 선행 조건과 최강 후행 조건은 다음과 같이 정의된다[7,8,9,10].

$wp($skip, $Q)=Q$
$wp($abort, $Q)=False$
$wp(x := e, Q)=Q[e/x]$
$wp(S_1; S_2, Q)=wp(S_1, wp(S_2,Q))$
$wp($ if B then S_1 else S_2 fi, $Q)$
$= B \wedge wp(S_1,Q) \vee \neg B \wedge wp(S_2,Q)$
$wp($ do B S od, $Q)$
$= \bigvee_{n=0}^{\infty} wp((\text{if } B \text{ then } S)^n(\text{abort}),Q)$
$sp($skip,$P)=P$
$sp($abort,$P)=False$
$sp(x := e,P)= \exists y\ (P[y/x] \wedge x=e[y/x])$
$sp(S_1; S_2,P)=sp(S_2,sp(S_1,P))$
$sp($if B then S_1 else S_2 fi$\},P)$
$= sp(S_1, B \wedge P) \vee sp(S_2, \neg B \wedge P)$
$sp($do B S od,$P)$
$= \bigvee_{n=0}^{\infty} sp((\text{if } B \text{ then } S)^n(\text{abort}),P)$

3. 명세 기반 슬라이싱의 정형적 정의

명세 $C=(P,Q)$에 대한 명세 기반 슬라이스를 정의하기 앞서, 이 논문에서는 선행 조건 기반 슬라이스(precondition-based slice)와 후행 조건 기반 슬라이스(postcondition-based slice)라는 명세 기반 슬라이스의 특수한 두 가지 형태를 정의한다.

3.1 선행 조건 기반 슬라이싱

선행 조건 P에 대한 선행 조건 기반 슬라이스는 전향적(forward), 정적(static) 슬라이스로써, 프로그램이 P를 만족하는 프로그램 상태에서 수행을 시작한 경우, 수행될 수 있고, 수행 된 후 프로그램의 상태를 변경할 수 있는 프로그램 문장과 제어 조건들의 집합으로 이루어진다.

만약 프로그램 S_i가 S_j로부터 영, 또는 하나이상의 문장들을 삭제함으로써 얻어질 수 있다면, S_i를 S_j의 부분(portion) 이라고 부르고, $S_i<S_j$와 같이 표현하기로 하자.

[정의 1] (P, *)-슬라이스

프로그램 S_1과 S_2이 주어졌을 때, 프로그램 S_1이 P에 관한 S_2의 슬라이스라는 것은 $sp(S_1,P)\equiv sp(S_2,P)$, $S_1<S_2$이고 $P\Rightarrow wp(S_2,true)$인 경우이며 $S_1 \lhd_{(P,*)} S_2$로 표현한다.

(P, *)-슬라이스는 선행 조건 P를 만족하는 초기 상태에 대해 프로그램의 행위를 보존하는 선행 조건 기반 슬라이스이다. 예를 들어, 다음과 같은 간단한 프로그램 "if ($i\geq 5$) then $n := n+i$ else $n:=10$ fi"을 살펴보자. 명세가 선행 조건 "$P\equiv(n=\eta \wedge i\geq 10)$, 이때 η는 임의의 정수값"으로 주어졌다고 하자. 이 프로그램에 대해 "(if 문장, $\{n, i\}$)"를 슬라이싱 기준으로 하여 전향적 슬라이스를 구하면 원래 프로그램 자체가 된다. 이는 프로그램의 모든 문

장이 주어진 슬라이싱 기준과 의존 관계가 있기 때문에 어떤 문장도 삭제될 수 없기 때문이다.

반면에, 선행 조건 기반 슬라이싱의 경우에는 else 부분이 skip으로 대치될 수 있다. 이는 조건문이 P를 만족하는 상태에서 수행된다면, else 부분이 절대 수행되지 않기 때문이다. 이 간단한 예를 통해, 선행 조건 기반 슬라이싱에서는 기존의 슬라이싱 기법과는 달리, 어떤 문장이 선행 조건에 나타나는 변수에 값을 대입하더라도, 경우에 따라서는 삭제될 수 있음을 알 수 있다.

규칙 1 $\sigma = \sigma' S_i \sigma''$, $sp(\sigma', P) \equiv p$, S_i가 대입문이고, $sp(S_i, p) \equiv p$이면, $\sigma[skip/S_i] \lhd_{(P,*)} \sigma$이다. 단, $\sigma[S_i/S_j]$ 는 문장 S_j이 S_i로 치환된 점을 제외하고는 σ와 같은 프로그램이다.

규칙 2 $\sigma = \sigma' S_i \sigma''$, $sp(\sigma', P) \equiv p$, S_i가 if B *then* $S_{i,1}$ *else* $S_{i,2}$ *fi*의 형태를 갖는 조건문이고, $p \Rightarrow \neg B$이면, $\sigma[S_{i,2}/S_i] \lhd_{(P,*)} \sigma$이다. 또, 비슷한 규칙이 $p \Rightarrow B$인 경우 $S_{i,1}$에 대해 성립한다.

규칙 3 $\sigma = \sigma' S_i \sigma''$, $sp(\sigma', P) \equiv p$, S_i가 *do* B S *do*와 같이 정의된 반복문이고, $sp(S, B \wedge p) \equiv p$이고 $p \Rightarrow wp$(do문, true)라면, $\sigma[skip/S_i] \lhd_{(P,*)} \sigma$ 이다.

규칙 4 $\sigma = \sigma' S_1; \cdots; S_{i-1}; S_i; \cdots; S_m \sigma''$, $sp(\sigma', P) \equiv p$이고, 어떤 i, $i \leq m$에 대해 $sp(S_1; \cdots; S_i, p) \equiv p$라면, $\sigma[skip/S_1; \cdots; S_i] \lhd_{(P,*)} \sigma$이다.

그림 1 선행 조건 기반 슬라이싱

그림 1은 주어진 선행 조건에 대하여 선행 조건 기반 슬라이스를 계산하기 위한 규칙들이다. 규칙 1은 대입문에 의해 프로그램 상태가 바뀌지 않는다면, 대입문을 skip으로 치환할 수 있다는 사실을 명시한다. 이때, 프로그램내의 각 문장은 고유하게 레이블되어 있다고 가정하며, 이러한 레이블을 이용하여 문법적으로 동일한 여러 개의 문장이 있을 경우에도 혼동없이 적절한 치환을 수행할 수 있다. 규칙 2는 조건문의 경우에 관한 규칙이다. 프로그램의 수행이 p를 만족하는 상태에서 수행되기 시작한 경우 만약 조건문의 분기 조건이 언제나 만족하거나 만족하지 않는다면, 조건문을 언제나 수행되는 쪽의 분기문으로 치환할 수 있다. 규칙 3은 만약 p가 반복문내의 문장들 S에 의해 변경되지 않으며, p에 의해 반복문의 종료가 보장된다면 반복문 전체가 삭제될 수 있음을 나타낸다.

이 경우 주어진 선행 조건 p는 S의 수행 전후에 언제나 만족하는 루프 인베리언트(loop invariant)이다. 어떤 경우에는 이보다 간단히 반복문의 선행 조건 기반 슬라이싱이 행해질 수 있다. 예를 들어, 반복문의 조건식이 주어진 선행 조건 p에 대해서 *False*의 값을 갖는다면, 즉 $p \Rightarrow \neg B$이 성립하는 경우, 반복문 내의 문장 S는 수행되지 않는다. 이러한 경우에 반복문 전체를 삭제하더라도 p에 대한 원 프로그램의 행위가 영향을 받지 않는다.

또, 이러한 규칙들 외에도 더욱 간결한 슬라이스를 구하기 위해서는 순차적 문장 결합에 의해 발생하는 상태 변화의 누적 효과를 고려하여야 한다. 예를 들어, 프로그램 S_1이 "x:=x-1"이고 S_2가 "x:=x+1"인 경우 "$S_1; S_2$"의 선행 조건 "$P \equiv (x \geq 0)$"에 대한 선행 조건 기반 슬라이스를 고려해보자. 문장 S_1의 최강 후행 조건 "$sp(S_1, P)$"가 문장 S_2의 선행 조건이 되며, "$P \neq sp(S_1, P)$"이며 "$sp(S_1, P) \neq sp(S_2, sp(S_1, P))$"이다. 따라서 규칙 1에 따르면, 두 문장 중 어느 것도 skip으로 치환될 수 없다. 하지만 P와 "$sp(S_2, sp(S_1, P)) = (x \geq 0)$"이 동치이므로 이러한 상황은 바람직하지 않다. 이 경우에는 순차적 문장 결합에 의해 주어진 선행 조건 P에 대한 두 문장의 효과가 서로 상쇄될 수 있기 때문이다. 규칙 4는 이러한 경우를 다루고 있다.

3.2 후행 조건 기반 슬라이싱

후행 조건 Q에 관한 후행 조건 기반 슬라이스는 후향적(backward), 정적(static) 슬라이스로써 주어진 후행 조건 Q에 대한 최약 선행 조건에 영향을 미칠 수 있는 프로그램 문장과 제어 조건들의 집합으로 이루어진다.

[정의 2] (*, Q)-슬라이스

프로그램 S_1과 S_2이 주어졌을 때, 프로그램 S_1이 Q에 관한 S_2의 슬라이스라는 것은 $wp(S_1, Q) \equiv wp(S_2, Q)$이고 $S_1 < S_2$인 경우이며 $S_1 \lhd_{(*,Q)} S_2$로 표현한다.

프로그램 S_1이 프로그램 S_2의 (*, Q)-슬라이스라는 것은 S_1이 S_2의 후행 조건 기반 슬라이스이며, 후행 조건 Q에 대해 같은 행위를 보임을 뜻한다. 이 경우에

규칙 5 $\sigma = \sigma' S_i \sigma''$, $wp(\sigma'', Q) \equiv q$, S_i가 대입문이고, $wp(S_i, q) \equiv q$이라면, $\sigma[skip/S_i] \lhd_{(*,Q)} \sigma$ 이다.

규칙 6 $\sigma = \sigma' S_i \sigma''$, $wp(\sigma'', Q) \equiv q$, S_i가 if B *then* $S_{i,1}$ *else* $S_{i,2}$ *fi*형태의 조건문이고 $B \Rightarrow wp(S_{i,1}, q) \equiv q$이라면, $\sigma[skip/S_{i,1}] \lhd_{(*,Q)} \sigma$이다. 또 비슷한 관계가 $\neg B \Rightarrow wp(S_{i,2}, q) \equiv q$인 경우 $S_{i,2}$에 대해 성립한다.

규칙 7 $\sigma = \sigma' S_i \sigma''$, $wp(\sigma'', Q) \equiv q$, S_i가 *do* B S *do*와 같이 정의된 형태의 반복문이고 $q \Rightarrow wp(S_i, true)$라면, $\sigma[skip/S_i] \lhd_{(*,Q)} \sigma$ 이다.

규칙 8 $\sigma = \sigma' S_1; \cdots; S_{i-1}; S_i; \cdots; S_m \sigma''$, $wp(\sigma'', Q) \equiv q$ 이고, 어떤 $i, (i \leq m)$에 대해 $wp(S_i; \cdots; S_m, q) \equiv q$라면, $\sigma[skip/S_i; \cdots; S_m] \lhd_{(*,Q)} \sigma$ 이다. }

그림 2 후행 조건 기반 슬라이싱

도 역시, S_1은 S_2의 문장과 조건들의 부분집합으로 이루어져 있다.

규칙 6에 따라 if 문장의 어떤 분기가 최약 선행 조건에 영향을 미치지 않는다면, 그 분기는 skip으로 치환될 수 있다. 또 규칙 7에 의해, 주어진 후행 조건이 루프 인베리언트(loop invariant)이고, 반복문의 종료가 보장되는 경우, 반복문 전체가 삭제될 수 있다. 또, 규칙 8은 순차적 문장 결합에 의해 발생하는 상태 변화의 누적 효과를 고려한 경우이다.

3.3 명세 기반 슬라이스의 추출

프로그램 S_1은 주어진 선행-후행 조건 쌍 (P,Q)에 대해, S_2로부터 추출될 수 있고, (P,Q)에 관한 S_2의 올바름을 보존하면 S_2의 명세 기반 슬라이스이다. 이때, 올바름을 보존한다는 것은 S_2가 주어진 명세 $C=(P,Q)$를 만족하면, S_2로부터 영, 또는 하나이상의 문장을 삭제함으로써 추출된 S_1 역시 명세를 만족한다는 것을 뜻한다. 이를 정형적으로 정의하면 다음과 같다.

정의 3 [(P, Q)-슬라이스]

프로그램 S_1과 S_2이 주어졌을 때, S_1이 $C=(P,Q)$에 대한 S_2의 슬라이스라는 것은, $(P \Rightarrow wp(S_2,Q)) \Rightarrow (P \Rightarrow wp(S_1,Q))$이고 $S_1 < S_2$ 인 경우이며, $S_1 \lhd_{(P,Q)} S_2$로 표현한다.

선행 조건 기반 슬라이스, 후행 조건 기반 슬라이스, 그리고 명세 기반 슬라이스는 다음 정리와 같이 서로 밀접하게 연관되어 있다[2].

정리 1 만약 $S_1 \lhd_{(P,*)} S_2$ 라면, $S_1 \lhd_{(P,Q)} S_2$ 이다.

정리 2 만약 $S_1 \lhd_{(*,Q)} S_2$ 라면, $S_1 \lhd_{(P,Q)} S_2$ 이다.

정리 1과 2에 따르면, 선행 조건 기반 슬라이스나 후행 조건 기반 슬라이스를 추출함으로써 명세 기반 슬라이스를 추출할 수 있다.

하지만, 비록 정리 1과 2에 따른 슬라이스들이 유효한 명세 기반 슬라이스들이지만, 더욱 간단하면서도 유효한 명세 기반 슬라이스를 구할 수 있다. 정리 1을 살펴보면, 주어진 정보 중에서 후행 조건을 이용하지 않고 명세 기반 슬라이스를 추출하며, 정리 2의 경우에는 선행 조건을 이용하지 않는 것을 알 수 있다. 만약, 선행 조건과 후행 조건을 동시에 이용하여 슬라이스를 구할 수 있다면 더욱 간결한 슬라이스를 구할 수 있을 것이다. 정리 1과 2로부터 다음과 같은 명세 기반 슬라이스를 추출하는 기본적인 정리들을 유도할 수 있다.

정리 3 만약 $S_2 \lhd_{(P,*)} S_3$이고, $S_1 \lhd_{(*,Q)} S_2$라면, $S_1 \lhd_{(P,Q)} S_3$이다.

정리 4 만약 $S_2 \lhd_{(*,Q)} S_3$이고 $S_1 \lhd_{(P,*)} S_2$라면, $S_1 \lhd_{(P,Q)} S_3$ 이다.

정리 3과 정리 4는 선행 조건 기반 슬라이싱을 적용한 결과에 다시 후행 조건 기반 슬라이싱을 적용하거나, 또는 이 둘의 적용 순서를 바꿔 적용함으로써 명세 $C=(P,Q)$에 대한 명세 기반 슬라이스를 추출할 수 있음을 보여준다.

4. 프로시져간 명세 기반 슬라이싱

이 절에서는, 명세 기반 슬라이싱을 프로시져에 대해서 확장한다. 지금까지의 명세 기반 슬라이스에 관한 논의는 프로시져 호출을 포함하지 않는다. 하지만, 명세 기반 슬라이싱이 실제적인 의미를 갖기 위해서는 프로시져 호출을 다룰 수 있어야 한다. 이를 위해 먼저 이 논문에서 다루는 프로그래밍 언어를 다음과 같이 확장한다.

S :: =skip (무행위)
| abort (비정상적 프로그램 종료)
| <variable>:=<expression>(대입문)
| $S_1;S_2$ (순차적 문장 결합)
| if B then S_1 else S_2 fi (조건문)
| do B S od (반복문)
| *procname*$(\overline{a}, \overline{b}, \overline{c})$ (프로시져 호출)

이때 프로시져 *procname*는 다음과 같이 정의된다고 가정한다.

```
proc procname(value <variable list> ;
              value result <variable list> ;
              result <variable list> );
        <Body>
endproc
```

이 정의에서, <variable list>는 변수 이름들과 이 변수들의 타입의 리스트이다. 또 <Body>는 하나 이상의 문장들이며, 각 프로시져는 지역적으로 정의된 변수를 포함할 수 있다.

인수 전달은 값에 의한 전달(pass-by-value), 결과에 의한 전달(pass-by-result), 값, 결과에 의한 전달(pass-by-value-result)등의 방법을 통해 이루어지며, 각각의 인수 전달 방법은 예약어 value, result, value result 등을 통해 명시된다[3].

2) 이 정리들에 대한 증명은 부록에 실려있다.

3) 이 논문에서는 논의의 집중을 위해, 변수의 상세한 정의 방법에 대한 논의를 생략하며, 전역변수의 이용을 금지한다. 또, 실제 프로시져의 정의가 프로그램중의 어디에 위치하는지 상관하지 않고, 프로시져의 정의가 필요한 시점에 그 정의 내용을 참조할 수 있다고 가정한다. 인수 전달의 다른 방법들, 예를 들면 주소에 의한 전달(pass-by-reference) 등은 고려하지 않는다.

프로시져가 호출되면, 세 단계를 거쳐 수행된다. 먼저, *value* 또는 *value result* 모드로 정의된 형식인수(formal parameter)에 대한 실인수(actual parameter)의 값들이 평가되며, 그 결과 값들이 해당되는 형식인수에 저장된다. 둘째, 첫번째 단계에서 정의된 형식인수를 이용하여 <Body>를 수행한다. 마지막으로, <Body> 수행이 끝난 후 result 또는 value result로 정의된 변수들의 값들이 실인수로 복사된다. 이때, result 모드로 정의된 변수들은 프로시져 내에서 값이 명시적으로 대입되기 전까지는 그 값이 정의되지 않는다.

프로시져 *procname()* 이 다음과 같이 정의되어 있다고 할때,

proc procname(value $\overline{x}$; value result $\overline{y}$; result $\overline{z}$);
<Body>
endproc,

프로시져 호출의 의미에 대한 위 정의에 따라, 프로시져 호출 문장의 최약 선행 조건과 최강 후행 조건을 정의하면 다음과 같다.

$$wp(procname(\overline{a}, \overline{b}, \overline{c}), Q) = wp(\text{``}\overline{x} := \overline{a}; \overline{y} := \overline{b}; \langle Body\rangle; \overline{b} := \overline{y}; \overline{c} := \overline{z}\text{''}, Q)$$

$$sp(procname(\overline{a}, \overline{b}, \overline{c}), P) = sp(\text{``}\overline{x} := \overline{a}; \overline{y} := \overline{b}; \langle Body\rangle; \overline{b} := \overline{y}; \overline{c} := \overline{z}\text{''}, P)$$

프로시져 호출의미에 대한 이러한 정의는 전적으로 2절에서 제시한 프로그래밍 언어로 이루어져 있다. 즉, 의미를 정의하는 데 있어서 새로운 개념을 추가하지 않았음으로, 3장의 프로시져내(intraprocedural) 명세 기반 슬라이싱에 관한 논의들을 수정 없이 프로시져간 명세 기반 슬라이싱에 이용할 수 있다.

procname()이 위와 같이 정의되어 있다고 할 때, 다음과 같은 규칙들을 통해 프로시져 호출 문장 전체의 슬라이싱을 다룰 수 있다.

규칙 9 S_i가 프로시져 호출 문장이고 그 형태가 $procname(\overline{a}, \overline{b}, \overline{c})$ 이고 $sp(\text{``}\overline{x} := \overline{a}; \overline{y} := \overline{b}; \langle Body\rangle; \overline{b} := \overline{y}; \overline{c} := \overline{z}\text{''}, p) \equiv p$라면, $\sigma[skip/S_i] \lhd_{(P,*)} \sigma$ 이다.

규칙 9는 프로시져가 수행되더라도 선행 조건이 변경되지 않는다면, 프로시져 호출 문장 자체가 skip으로 치환될 수 있음을 뜻한다.

규칙 10 S_i가 프로시져 호출 문장이고 그 형태가 $procname(\overline{a}, \overline{b}, \overline{c})$ 이고 $wp(\text{``}\overline{x} := \overline{a}; \overline{y} := \overline{b}; \langle Body\rangle; \overline{b} := \overline{y}; \overline{c} := \overline{z}\text{''}, q) \equiv q$라면, $\sigma[skip/S_i] \lhd_{(*,Q)} \sigma$ 이다.

규칙 10은 만약 프로시져의 내부에 명시된 일련의 문장들이 최약 선행 조건에 영향을 미치지 않는 경우, 프로시져 호출 문장 자체를 skip으로 치환할 수 있음을 보여준다.

규칙 9와 10은 프로시져 호출 전체를 skip으로 치환하는 경우를 다루고 있다. 하지만, 프로시져 호출 문장이 전체로써 skip으로 치환될 수는 없다고 하더라도, 프로시져 호출에 의해 수행되는 문장 즉, "$\overline{x} := \overline{a}; \overline{y} := \overline{b}; \langle Body\rangle; \overline{b} := \overline{y}; \overline{c} := \overline{z}$ "중의 일부가, 규칙 4와 8에 의해 skip으로 치환될 수 있다. 이러한 프로시져의 일부분의 삭제는 먼저 프로시져 호출 문장을 위에 설명한 것과 같은 동등한 의미를 갖는 프로시져가 없는 문장으로 확장을 하는 단계와 앞절에서 설명한 명세 기반 슬라이싱 규칙들을 적용하는 단계를 통해 이루어질 수 있다.

5. 명세 기반 슬라이싱의 응용

명세 기반 슬라이싱을 통해 추출된 컴포넌트는 주어진 명세를 만족하며, 또 명세와 연관된 문장들로만 이루어진다. 따라서 명세 기반 슬라이싱 기법은 프로그램의 복잡도를 관리하기 위한 여러 소프트웨어 공학의 활동들에 이용될 수 있다. 이 절에서는 명세 기반 슬라이싱 기법을 통해 소프트웨어 공학의 많은 문제 중, 소프트웨어 재사용(software reuse)과 소프트웨어 재구성(software restructuring)이 어떻게 향상될 수 있는지 살펴본다.

5.1 소프트웨어 재사용

소프트웨어 재사용은 소프트웨어를 제작하는 데 있어서, 개발의 초기단계부터 전단계를 수행하기보다는 기존의 소프트웨어 생성물로부터 도움을 받아 더 싸고, 빠르게 소프트웨어를 제작하는 방법이다[11]. 소프트웨어 재사용을 통해 시스템을 생성하기 위해서는 크게 세 가지 작업이 이루어져야한다. 먼저, 재사용 가능한 컴포넌트를 기존의 시스템으로부터 추출하거나, 새롭게 작성해서, 추후 재사용될 수 있도록 저장소(repository)에 적절한 구조로 저장해 두어야한다. 일련의 재사용 가능한 컴포넌트가 축적되었다면, 두 번째로 저장소(repository)로부터 적절한 재사용 가능한 컴포넌트를 판별하여야 한다. 마지막으로, 어떤 컴포넌트를 재사용할 것인지를 결정했다면, 그 컴포넌트를 적절히 변경하여 현재 개발중인 소프트웨어 제품에 결합(integrate)시켜야 한다. 이때, 효과적인 재사용이 이루어지기 위해서는 재사용에 필요한 변경이 최소한인 컴포넌트를 선택하여 재사용하여야 하며, 또 이 재사용 과정 역시 효율적으로 이루어져야한다.

명세 기반 슬라이싱 기법은 필요한 컴포넌트를 추출

하기 위한 적절한 컴포넌트를 판별하는 과정과, 이 판별된 컴포넌트로부터 필요한 컴포넌트를 추출하는 과정 모두에 응용될 수 있다.

5.1.1 명세 기반 슬라이싱 기법을 응용한 재사용 컴포넌트의 추출

이 절에서는 명세 기반 슬라이싱을 통해, 어떻게 실제 프로그램에서 재사용 가능한 컴포넌트를 추출할 수 있는지 예를 보이며 명세 기반 슬라이싱을 통해 추출된 재사용 컴포넌트와 기존의 기법에 따른 재사용 컴포넌트를 비교해 본다.

다음과 같은 예제를 고려해 보자.

"두 숫자가 입력으로 주어질 때, 절대최대값과 절대최소값을 계산한다. 두 입력이 값은 부호이면 절대최대값은 두 수중 절대값이 큰 값, 절대최소값은 절대값이 작은 값이 된다. 만약 두 수의 부호가 다르다면 절대최대값, 절대최소값 모두 0이다."

```
1:    if    (a>0 ∧ b>0) or (a<0 ∧ b<0) then
2:          if  a>0  then
3:              if  a>b  then
4:                  absmax: =a;
5:                  absmin: =b
6:              else
7:                  absmax: =b;
8:                  absmin: =a
9:              fi
10:         else
11:             if  a>b  then
12:                 absmax: = - b;
13:                 absmin: = - a
14:             else
15:                 absmax: = - a;
16:                 absmin: = - b
17:             fi
18:         fi
19:   else
20:         absmax: =0;
21:         absmin: =0
22:   fi
```

그림 3 절대최대값/절대최소값 프로그램

그림 3의 프로그램은 예제의 요구사항을 만족하는 프로그램이다. 이제, 이 프로그램으로부터 재사용 가능한 컴포넌트를 추출하는 경우를 살펴보자. 추출될 컴포넌트는 두 양수가 주어졌을 때 최대값을 계산하여야 한다고 가정하자. 이 새로운 컴포넌트의 요구사항은 다음과 같이 명세될 수 있다.

P: $p_0 \equiv a>0 \wedge b>0$
Q: $q_0 \equiv absmax = a>b>0 \vee absmax = b \geq a>0$

선행 조건 기반 슬라이스를 추출하기 위해, 그림 1에 설명된 선행 조건 기반 슬라이싱 규칙들을 그림 3에 적용한다[4]. 그림 4는 P에 대해 더 이상 삭제될 수 있는 문장이 없는 최종 슬라이스이다. 이 선행 조건 기반 슬라이스는 명세 $C=(P,Q)$를 만족시킴으로써 (P,Q)에 대해 원 프로그램의 행위를 보존한다.

```
3:    if  a>b  then
4:            absmax: =a;
5:            absmin: =b
6:    else
7:            absmax: =b;
8:            absmin: =a
9:    fi
```

그림 4 선행 조건 기반 슬라이스

또, 정리 2에 따르면 Q에 관해 후행 조건 기반 슬라이스를 구함으로써 명세 기반 슬라이스를 구할 수도 있다. 그림 5는 후행 조건 기반 슬라이싱의 결과이다. 후행 조건 기반 슬라이스 역시 명세 $C=(P,Q)$를 만족함을 알 수 있다.

```
1: if    (a>0 ∧ b>0) or (a<0 ∧ b<0) then
2:       if  a>0 then
3:           if  a>b then
4:               absmax: =a
6:           else
7:               absmax: =b
9:           fi
10:      else
11:          if  a>b then
12:              absmax: = - b
14:          else
15:              absmax: = - a
17:          fi
18:      fi
19: else
20:      absmax: =0
22: fi
```

그림 5 후행 조건 기반 슬라이스

그림 4와 그림 5가 적합한 명세 기반 슬라이스이기는 하지만, 정리 3과 4에 따라 더욱 간결한 슬라이스를 구할 수 있다. 정리 3에 따라 선행 조건 기반 슬라이스(그림 4)에 후행 조건 기반 슬라이싱을 적용한 결과로 얻을 수 있는 명세 기반 슬라이스가 그림 6에 나타나 있다.

이 슬라이스가 명세 $C=(P,Q)$를 만족함은 명확하다.

4) 구체적인 슬라이싱 과정은 그림 1과 그림 2의 슬라이싱 규칙들을 적용하는 과정으로써, 최약 선행 조건과 최강 선행 조건의 유도와 슬라이싱 규칙에의 부합여부를 반복적으로, 더 이상 슬라이싱 되는 문장이 없을 때까지 수행하는 과정이다. 구체적인 적용의 내용은 지면관계상 생략하도록 한다.

또, 정리 4에 따라 그림 5의 선행 조건 기반 슬라이스에 후행 조건 기반 슬라이싱을 적용함으로써도 역시 더욱 간결한 명세 기반 슬라이스를 구할 수 있다.

```
3:      if  a>b then
4:          absmax: =a
6:      else
7:          absmax: =b
9:      fi
```

그림 6 더욱 간결한 슬라이스

이 논문에서 제시한 명세 기반 슬라이싱 기법과 기존의 프로그램 슬라이싱 기법을 비교하기 위해, 그림 7을 살펴보자. 그림 7은 그림 3에 나타난 예제 프로그램의 프로그램 의존 그래프(Program Dependence Graph:PDG)를 나타낸다[12]. 슬라이싱 기준 (ENTRY, *{a,b}*)[5]에 대한 전향적(forward) 슬라이스를 고려해보자. ENTRY는 프로시져의 모든 다른 위치보다 선행하는 가상의 위치이다. PDG의 모든 노드들이 ENTRY 노드에 제어 의존적(control dependent)이기 때문에 슬라이싱을 통해 제거할 수 있는 노드는 없다.

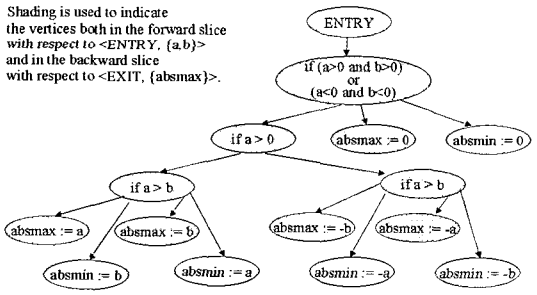

그림 7 기존의 기법에 의한 슬라이스

또, (EXIT, *{absmax}*)에 대해 후향적(backward) 슬라이스를 고려해보자. EXIT는 다른 모든 문장의 뒤에 오는 위치로, 후향 슬라이스의 경우에는 그림에서 음영이 들어 있는 노드들이 포함된다. 따라서, 이 논문에서 제시한 슬라이싱 기법처럼, 전향적 슬라이싱과 후향적 슬라이싱을 모두 고려한다고 하더라도, 기존의 슬라이싱 기법에 따른 슬라이스는 그림 7과 같게 된다. 즉, 기존의 슬라이싱 기법에 따른 슬라이스는 명세 기반 슬라이싱에 의한 슬라이스보다 부정확하다.

또, 프로시져간 명세 기반 슬라이싱의 적용 예로써, 절대최대값/절대최소값 프로그램을 살펴보자. 그림 8은 절대최대값/절대최소값 프로그램을 프로시져 호출을 포함하도록 변경한 것이다. 이를 위해 원 프로그램중 내포된 조건 문장을 *minmax()*라는 이름의 프로시져로 추출하였다. 이 프로시져는 입력 인수들의 부호를 고려하지 않고, 수학적인 크고 작음만을 비교하는 역할을 한다. 주 프로시져는 절대최대값/절대최소값을 판별하기 위해 *minmax()*에의 인수의 부호를 적절히 변경하게 된다.

```
1   proc abs-minmax(value a,b : int; result absmin,absmax : int);
2       if (a>0 and b>0) or (a<0 and b<0) then
3           if a>0 then
4               minmax(a,b,absmin,absmax)
5           else
6               minmax(-a,-b,absmin,absmax)
7           fi
8       else
9           absmax := 0;
10          absmin := 0
11      fi
12  endproc

a   proc minmax({value} aa,bb : int; result min,max : int);
b       if aa>bb then
c           max := aa;
d           min := bb
e       else
f           max := bb;
g           min := aa
h       fi
i   endproc
```

그림 8 절대최소값/절대최대값의 다중 프로시져 버전

이제, 그림 8의 프로그램으로부터 두 양수가 주어졌을 때 최대값을 찾는 재사용 가능한 컴포넌트를 추출하는 경우를 생각해보자. 정리 1에 따르면, 먼저 P에 관해 선행 조건 기반 슬라이스를 계산하고 이 결과에 대해, Q에 관한 후행 조건 기반 슬라이싱을 수행함으로써 명세 기반 슬라이싱을 수행할 수 있다. 프로시져가 여럿인 프로그램의 슬라이싱은, 위에서 정의한 방법에 따라 프로시져가 없는 슬라이싱으로 변환가능하며, 그 결과는 그림 9와 같다.

```
1   proc abs-minmax(value a,b : int; result absmin,absmax : int);
4           minmax(a,b,absmin,absmax)
12  endproc

a   proc minmax({value} aa,bb : int; result min,max : int);
b       if aa>bb then
c           max := aa;
e       else
f           max := bb;
h       fi
i   endproc
```

그림 9 프로시져간 명세 기반 슬라이싱의 예

5) 슬라이싱 기준은 프로그램내의 위치와 변수 집합의 쌍으로 주어진다.

한편, 슬라이싱과정에서 원래 프로그램의 많은 부분이 사라지고, 따라서 최종 프로그램 구조가 불필요하게 복잡할 수 있기 때문에 구해진 슬라이스를 재성형할 필요가 있을 수도 있다. 그림 9의 슬라이스가 그 좋은 예이다. 이 슬라이스는 두개의 프로시져로 이루어져 있으며 호출하는 프로시져는 오직 하나의 프로그램 문장만을 포함하고 있다. 이 문장은 두개의 프로시져중 다른 하나에 대한 호출 문장이다. 이 예처럼 프로그램의 이해에 도움을 주지 않는, 불필요하게 복잡한 중첩된 프로시져구조는 바람직하지 않으며, 이러한 관점에서 중첩된 구조를 간단하게 만드는 것이 프로그램의 이해에 도움을 줄 수 있다.

재성형 과정은 크게 세 단계를 거쳐 진행된다. 먼저, 프로시져 호출 문장을 앞에서 정의한 프로시져 호출의 의미에 따른 문장들로 대체한다. 이중에는 실인수의 형식인수에의 대입도 포함된다. 둘째, 프로시져 정의에 나타난 형식인수의 사례들을 실질인수로 치환하고, 인수전달에 나타나는 대입문들을 삭제함으로써 인수전달에 따라 필요한 문장들을 없앤다. 마지막으로, 프로시져 이름과 변수 이름들을 더욱 의미있는 것들로 바꿈으로써 새로운 프로시져가 쉽게 이해될 수 있도록 한다. 그림 10은 이러한 세 단계에 걸쳐 명세 기반 슬라이스를 재성형한 결과로써, 이 프로그램과 이전 단계의 프로그램 그림 9를 비교해 보면 재성형을 통해 프로그램이 더욱 이해하기 쉽게 되었다는 것을 알 수 있다.

```
1 proc abs-minmax(value a,b : int; result absmin,absmax : int);
4a   aa := a; bb :=b;
4b   if aa>bb then
4c    max := aa
4e   else
4f    max := bb
4h   fi;
4i   absmin := min; absmax := max
12 endproc

1 proc abs-minmax(value a,b : int; result absmin,absmax : int);
4b   if a>b then
4c    absmax := a
4e   else
4f    absmax := b
4h   fi
12 endproc

1 proc positive-max(value a,b : int; result max : int);
2   if a>b then
3    max := a
4   else
5    max := b
6   fi
7 endproc
```

그림 10 명세 기반 슬라이스의 재성형

5.1.2 재사용 컴포넌트의 판별

앞절에서는 기존의 시스템으로부터 재사용 가능한 컴포넌트를 추출하는 데 명세 기반 슬라이싱 기법이 어떻게 이용될 수 있는지를 살펴보았다. 이 절에서는 저장소의 여러 재사용 가능한 컴포넌트 후보로부터 최적의 재사용 컴포넌트를 판별하는 문제와 명세 기반 슬라이싱이 어떻게 연관되며, 어떻게 전체 재사용 과정을 효과적으로 만들 수 있는지 논의한다.

지금까지 논의했던 명세 기반 슬라이싱에서, 우리는 원래의 프로그램이 추출하고자 하는 슬라이스의 상세화(refinement)라고 가정하였다. 즉, 주어진 특정 (P,Q)에 대하여, 구하고자하는 슬라이스 S_1과 원래의 프로그램 S_2 사이에는 $P[S_2]Q \Rightarrow P[S_1]Q$가 성립한다고 가정하였다. 이러한 가정을 명세간의 상세화 순서 관계(refinement ordering relation)를 이용하여 정형적으로 표현하면 다음과 같다.

[정의 4]

명세 $C_1=(P_1, Q_1)$이 $C_2=(P_2, Q_2)$의 상세화라는 것은, $P_2 \Rightarrow P_1$이고 $Q_1 \Rightarrow Q_2$이며, $C_2 \sqsubseteq C_1$로 표현한다.

즉, 명세 기반 슬라이싱을 통해 원 프로그램으로부터 명세 C에 관한 슬라이스를 구하기 위해서는, 원 프로그램의 명세가 구하고자하는 컴포넌트의 명세 C의 상세화이어야 한다.

하지만, 이러한 경우는 특수한 경우로써, 재사용을 위한 소프트웨어 저장소의 어떤 프로그램도 C의 상세화가 아니며, 단지 C의 요구사항중의 일부만을 만족할 수도 있다. 이러한 경우에는 저장소의 프로그램중 명세 C의 가장 큰 부분을 만족시키는 프로그램을 찾을 수 있어야 한다. 이러한 프로그램을 찾은 후, 이 프로그램으로부터 C를 만족시키기 위한 노력이 가장 적게 드는 슬라이스를 구할 수 있다. 이러한 과정은 명세들간의 상세화 관계가 (세미)래티스(semi lattice)구조라는 사실을 활용함으로써 수행될 수 있다[13].

$(\Sigma, \sqsubseteq)$가 래티스 구조라고 하자. 이때, Σ는 명세의 집합이며 $C_1, C_2 \in \Sigma$의 미트(meet, 또는 greatest lower bound)는 다음과 같이 정의된다.

[정의 5]

$C_1=(P_1, Q_1)$과 $C_2=(P_2, Q_2)$의 미트(meet, 또는 greatest lower bound)는 $C_1 \sqcap C_2$로 표현되는 명세 (P,Q)로서 $P=P_1 \wedge P_2$이며 $Q=Q_1 \vee Q_2$이다.

위의 정의로부터, C_1과 C_2의 미트는 두 명세에 공통된 정보를 나타냄을 알 수 있다. 따라서 명세의 래티스

구조와 미트의 개념을 바탕으로, 소프트웨어 저장소의 프로그램중 주어진 명세와 기능을 최대한 공유하는 프로그램을 찾을 수 있다. 주어진 두 명세의 미트를 통해 두 명세간의 공통된 정보의 크고 작음을 비교할 수 있으므로, 더욱 많은 정보를 공유하는 프로그램을 찾을 수 있고, 따라서 한 명세에 관련된 프로그램을 변경해서 다른 명세를 만족시키는 작업의 비용도 줄어들 것이다. 이런 관점에서, 소프트웨어 저장소에 있는 재사용 가능 소프트웨어의 명세 C_i와 슬라이싱 기준으로 주어지는 원하는 소프트웨어의 명세 C의 미트 $C_i \sqcap C$를 계산함으로써 앞에서 설명했던 재사용의 판별 과정을 도와줄 수 있다.

Σ를 소프트웨어 저장소의 재사용 가능 소프트웨어의 명세의 집합이라고 하자. 명세 c는 다음과 같은 조건을 만족시킬 때 주어진 명세 C와의 미트를 최대화시킨다고 한다.

$\forall C_i \in \Sigma$ 에 대해, $C \sqcap C_i \sqsupseteq C \sqcap c \Rightarrow C \sqcap C_i = C \sqcap c$.

이처럼 C와의 미트를 최대화시키는 명세를 c라고 했을 때, c를 명세로 갖는 프로그램에 대해 $C \sqcap c$을 슬라이싱 기준으로 하여 명세 기반 슬라이싱을 적용할 수 있다. 이러한 최대화 미트를 사용함으로써, 상세화 관계에 있지 않은 프로그램에 대한 명세 기반 슬라이싱을 통하여, 원하는 기능을 최대한 지원하는 프로그램을 효과적으로 재사용할 수 있다.

명세의 구조가 래티스구조이기 때문에, 일반적으로 두 명세의 미트는 하나 이상일 수 있다. 따라서 이러한 경우에, 즉 하나 이상의 저장소에 저장된 프로그램이 판별 기준을 만족하는 경우에, ''여러 개의 후보들 중에서 어떤 것을 재사용의 대상으로 결정할 것인가?" 라는 문제가 생길 수 있다. 만약 하나 이상의 후보가 선택된다면, 고려하여야 할 후보의 수를 줄이기 위해 다음과 같은 조건을 추가로 제시할 수 있다. $\gamma(C)$를 명세의 집합으로써, $\gamma(C) = \{d \mid \forall C_i \in \Sigma, C \sqcap C_i \sqsupseteq C \sqcap c \Rightarrow C \sqcap C_i = C \sqcap c\}$라고 하자. 그러면, 우리가 원하는 명세 γ'는,

$\gamma' = \{d \mid \nexists x \in \gamma(C)$ such that $c \sqsupseteq x\}$

와 같이 정의될 수 있다.

위의 기준은 사용자가 원하는 기능, 즉 명세와 최대한 많은 부분을 공유하고 있으며, 동시에 최소인 명세가 사용자가 원하지 않는 부분을 가장 조금 갖고 있을 것이라는 가정을 바탕으로 하고 있다.

5.2 소프트웨어 재구성

소프트웨어 재구성(software restructuring)이란 "소프트웨어를 더 쉽게 이해될 수 있고, 변경될 수 있으며 미래의 변경시 오류가 더 일어나지 않도록 변경하는 행위"이다[14]. 소프트웨어 재구성과 소프트웨어 재사용은 모두 기존의 프로그램으로부터 새로운 프로그램을 유도한다는 점에서는 유사하다. 하지만, 소프트웨어 재사용이 원래 프로그램이 가정하고 있는 환경과는 다른 환경에서 사용되도록 프로그램을 변경하는 반면, 재구성에 의한 프로그램은 원래의 프로그램이 가정하고 있는 환경과 동일한 환경에서 사용되기 위한 프로그램이다. 재구성된 프로그램은 원래의 프로그램과 같은 요구사항을 만족하며, 그 외에도 더욱 향상된 응집도(cohesion), 결합도(coupling), 가독성(readability), 이해가능성(understandability) 등을 갖는다.

명세 기반 슬라이싱을 통해 복잡한 소프트웨어를 재구성하는 예로써, 그림 11의 프로그램을 살펴보자. 그림 11의 프로그램은 $\sum_{i=1}^{i=n} i$과 $\prod_{i=1}^{i=n} i$을 동시에 계산하는 프로그램이다. 그림 11의 프로그램의 명세 $C_{PS} = (P_{PS}, Q_{PS})$는 다음과 같다.

$P_{PS} \equiv n \geq 1$

$Q_{PS} \equiv prod = \prod_{i=1}^{i=n} i \wedge sum = \sum_{i=1}^{i=n} i$

```
1:  proc ProdAndSum( value n : int; result prod,sum : int);
2:    prod := 1;
3:    sum := 0;
4:    i := 1;
5:    do i ≤ n
6:      prod := prod * i;
7:      sum := sum + i;
8:      i := i + 1
9:    do
10: endproc
```

그림 11 소프트웨어 재구성의 예-ProdAndSum()

그림 11로부터 명세 기반 슬라이싱을 $C_{Prod} = (n \geq 1, prod = \prod_{i=1}^{i=n} i)$과 $C_{\Sigma} = (n \geq 1, sum = \sum_{i=1}^{i=n} i)$의 두 명세에 대해 적용하면, 합 프로그램과 곱 프로그램을 분리할 수 있다. 먼저, 두 명세 C_{Prod}, C_{Sum}의 선행 조건이 C_{PS}의 선행 조건과 동일하므로, 후행 조건 기반 슬라이싱을 적용한다. 그림 12에 그 결과가 정리되어 있다. 각각의 슬라이스가 오직 하나만의, 명확히 정의된 기능을 수행하므로 원 모듈의응집도가 기능적(functional) 수준으로 향상되었다.

지금까지 예를 통하여, 원래의 모듈이 여러 개의 연관되지 않은 기능들을 수행하는 경우, 명세 기반 슬라이싱을 통하여 이 모듈이 어떻게 하나의 기능만을 수행하도

```
p1:  proc Prod( value n : int; result, prod : int);
p2:    prod := 1;
p3:    i := 1;
p5:    do i ≤ n
p6:      prod := prod * i;
p7:      i := i + 1
p8:    do
p9:  endproc

s1:  proc Sum( value n : int; result sum : int);
s2:    sum := 0;
s3:    i := 1;
s5:    do i ≤ n
s6:      sum := sum + i;
s7:      i := i + 1
s8:    do
s9:  endproc
```

그림 12 재구성된 프로그램 - Prod()와 Sum()

록 재구성될 수 있는지를 살펴보았다.

6. 관련 연구

프로그램 슬라이스에 관한 개념이 Weiser [1]에 의해 처음 도입된 이후 다양한 슬라이스에 대한 연구들이 제시되었다. 이 절에서는 정적(static) 슬라이싱 [1], 동적(dynamic)슬라이싱 [15], 유사-정적(quasi-static) 슬라이싱 [16], p-슬라이싱 [17]과 조건부(conditioned) 슬라이싱 [3] 등에 관해 살펴본다. 또 명세 기반 슬라이싱이 다른 슬라이싱 개념들을 설명할 수 있는 일반적인 프레임웍으로도 이용될 수 있음을 보인다.

정적 슬라이싱과 동적 슬라이싱의 개념은 다른 많은 슬라이싱에 대한 정의들의 바탕을 이루는 개념이다. 이 둘의 차이는 동적 슬라이싱이 프로그램의 입력으로 고정된 입력을 가정하는 반면 정적 슬라이싱은 입력에 대해 별다른 가정을 하지 않는다는 사실이다[15,18]. 동적 슬라이싱 기준은 입력을 포함하며, 프로그램의 위치뿐만이 아니라, 실행 기록에 나타나는 한 문장의 서로 다른 수행 예까지도 구분한다. 실행 시점의 정보를 이용함으로써 동적 슬라이스는 일반적으로 정적 슬라이스보다 더욱 간결할 수 있다. 하지만 결과로 얻어지는 슬라이스가 슬라이싱 기준에 포함된 입력에만 유효하게 되는 적용상의 단점을 갖게 된다.

동적 슬라이스가 프로그램의 마지막 문장에 대해 계산된다면, 동적 슬라이싱 기준을 명세 기반 슬라이싱 기준으로 표현할 수 있다. 예를 들어, 프로그램 S가 두 입력 변수, in_1과 in_2, 하나의 출력 변수 out를 갖는다고 하자. 이제 이 프로그램으로부터 입력 값들이 $in_1=3$과 $in_2=0$일 때, 프로그램의 마지막 문장에서 출력되는 출력 변수 out의 값에 영향을 미치는 문장들로 이루어진 동적 슬라이스를 구하는 경우를 살펴보자. 이러한 동적 슬라이스는 슬라이싱 기준 $C=(in_1=3 \wedge in_2=0, out)$을 이용하여 명세 기반 슬라이싱을 통해 얻는 슬라이스와 동일하다.

유사-정적(quasi-static) 슬라이싱은 정적 슬라이싱과 동적 슬라이싱의 특성을 함께 갖는 슬라이싱의 대표적인 예이다[16]. 유사-정적 슬라이스는 프로그램의 특정 위치의 변수의 값에 영향을 미치는 문장들로 구성된다. 다만, 정적 슬라이스와는 달리, 여러 입력중 어떤 입력들은 고정되며, 다른 입력 값들은 변경될 수 있다. 만약 모든 입력 값들이 고정된다면, 유사-정적 슬라이싱은 동적 슬라이싱과 같게 된다. 또, 모든 입력 값들이 변화될 수 있다면, 유사-정적 슬라이싱은 정적 슬라이싱과 같게 된다. 또, 이외에도 동적 슬라이싱 특성과 정적 슬라이싱 특성을 함께 이용하는 슬라이싱 기법도 연구되고 있다[19,20]. 하지만 이 기법들의 효용성은 프로그램의 호출 패턴에 의존적인 단점이 있다[20].

동적 슬라이싱의 경우와 유사하게, 유사-정적 슬라이스 역시 명세 기반 슬라이싱을 통해 설명될 수 있다. 예를 들어, 위에서 설명한 프로그램 S를 다시 살펴보자. 프로그램 S로부터 두 입력 변수중 하나의 입력 값이 고정된 경우, 즉 $in_1=5$인 경우, 출력 변수 out의 값에 대한 S의 행위를 보존하는 유사-정적 슬라이스를 추출한다고 가정해보자. 이 경우에 구해지는 유사-정적 슬라이스는 슬라이싱 기준 $C=(in_1=5, out)$에 대한 명세 기반 슬라이스와 같게 된다.

조건부(conditioned) 슬라이싱 [3]과 p-슬라이싱 [17]은 명세 기반 슬라이싱과 유사한 슬라이싱 기법들이다. 조건부 슬라이싱은 입력에 대한 조건을 명시하는 명세를 필요로 하며, 이 논문에서 제시하고 있는 선행 조건 기반 슬라이스와 대응된다. 하지만, 조건부 슬라이싱은 슬라이스가 만족해야만 하는 바람직한 상태에 대한 조건인 후행 조건을 고려하지 않고 있다. 반면에 p-슬라이싱은 최약 선행 조건에 기반하고 있으며, 이 논문에서 제시한 후행 조건 기반 슬라이스의 정의와 같다. 하지만, p-슬라이싱은, 슬라이싱 기준으로써 후행 조건만을 고려하고 있으며 프로시져간 슬라이싱에 대한 고려 역시 빠져 있다. 조건부 슬라이싱과 p-슬라이싱은 명세의 한 면만을 이용하여 슬라이스를 구하기 때문에 명세 기반 슬라이싱보다 더 부정확한 슬라이스를 구하게 된다.

7. 결론 및 향후 연구 방향

이 논문에서는 명세 기반 프로그램 슬라이싱에 관한 개념 및 정형적 정의를 제시하였다. 이를 위하여 명세 기반 슬라이스와 이를 추출하기 위한 프로시져간 호출을 고려하는 정형적 슬라이싱 규칙들을 제시하였다.

명세 기반 슬라이싱은 명세에 나타나 있는 정보를 모두 활용함으로써 기존의 슬라이싱 기법들보다 더욱 정확하고 간결한 슬라이스들을 추출할 수 있다. 명세 기반 슬라이싱 기법의 유용성을 보이기 위해, 명세 기반 슬라이싱을 이용하여 어떻게 기존의 프로그램으로부터 재사용 가능한 컴포넌트를 추출하는 과정과 복잡한 소프트웨어를 재구성하는 과정을 향상시킬 수 있는지를 살펴보았다. 특히, 이 논문에서는 명세들간의 상세화 관계의 래티스 구조를 이용하여, 소프트웨어 재사용이 가능한 경우를 좀 더 일반화하였다. 또, 명세 기반 슬라이싱과 다른 슬라이싱 개념들의 비교를 통해 명세 기반 슬라이싱이 다른 슬라이싱 개념들을 설명할 수 있는 프레임웍으로 이용될 수 있음을 보였다.

명세 기반 슬라이스의 개념이 기존의 슬라이스의 정의들과는 다른 다양한 특징들을 포함하고 있기 때문에 이에 관한 많은 연구가 필요하다. 이론적인 면에서는, 명세 기반 슬라이스의 많은 다른 변종들과 그들의 특징 및 그들간의 관계 등에 관한 연구가 이루어져야 할 것이다. 특히, 명세 기반 슬라이스를 좀 더 일반적으로 적용할 수 있기 위해서는 명세 기반 슬라이스의 정의에서 가정하고 있는 $P \Rightarrow wp(\sigma, Q)$의 조건을 완화할 수 있는 방안이 연구되어야 할 것이다.

이 논문에서 제시된 명세 기반 슬라이스 기법이 실제로 널리 이용되기 위해서는 몇가지 연구가 수행되어야 한다. 먼저, 이 논문에서 지원하지 못하고 있는 다양한 고급 프로그래밍 언어의 특성들을 지원할 수 있는 방안이 연구되어야 할 것이다. 실제 프로그램에 자주 이용되는 배열, 포인터, 전역 변수등의 지원은 필수적이다. 두번째로, 임의의 반복문을 지원하기 위해서는 최약 선행 조건과 최강 선행 조건에 대한 의미있는 근사(approximation)가 정의되어야 한다. 최약 선행 조건과 최강 선행 조건을 구하기 위해서는, 정의에 따르면 모든 가능한 반복을 고려하여야 하지만, 많은 수의 반복을 포함하고 있는 실제 프로그램에서 이를 구하는 것은 비실용적이기 때문이다. 또, 반복문의 최약 선행 조건과 최강 선행 조건에 대한 근사를 제외한 부분의 시간 복잡도는 프로그램의 크기에 비례하므로, 이 근사의 유효성 및 복잡도에 따라 전체 명세 기반 슬라이스 기법의 시간 복잡도가 영향을 받는다.

마지막으로 소프트웨어 컴포넌트 명세의 체계적 작성을 지원하며, 소프트웨어의 검증 등 명세와 연관된 소프트웨어 공학의 활동들과, 본 논문에서 제시하고 있는 슬라이싱 과정을 통합하여 지원할 수 있는 도구의 개발이 필요하다.

참 고 문 헌

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

[2] H. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," IEEE Trans. on Software Engineering, vol. 21, no. 6, pp. 528-562, Jun 1995.

[3] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca, "Software Salvaging based on Conditions," In Proceedings of International Conference on Software Maintenance, pp. 424-433, 1994.

[4] 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. 1981.

[5] 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.

[6] M. J. Harrold, N. Ci, "Reuse-Driven Interprocedural Slicing," In Proceedings of International Conference on Software Engineering, pp. 74-83, April 1998.

[7] J. de Bakker, Mathematical Theory of Program Correctness, Prentice/Hall Int'l. Inc., 1980.

[8] R. J. R. Back, "A Calculus of Refinements for Program Derivations," Acta Informatica, vol. 25, pp. 593-624, 1988.

[9] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Communications of the ACM, vol. 18, no. 8, pp. 453-457, Aug. 1975.

[10] D. Gries, The Science of Programming, Springer-Verlag New York Inc., 1981.

[11] C. W. Krueger, "Software Reuse," ACM Computing Surveys, vol. 24, no. 2, pp. 131-183, Jun 1992.

[12] S. Horwitz, J. Prins, and T. Reps, "Integrating Non-interfering Versions of Programs," ACM Trans. on Programming Languages and Systems, vol. 11, no. 3, pp. 345-387, July 1989.

[13] R. Mili, A. Mili, and R. T. Mittermeir, "Storing

and Retrieving Software Components: A Refinement Based System," IEEE Trans. on Software Engineering , vol. 23, no. 7, pp.445-460, July 1997.
[14] R. S. Arnold, "Software Restructuring," Proc. IEEE, pp. 607-617, 1989.
[15] B. Korel and J. Laski, "Dynamic Slicing of Computer Programs," Journal of Systems and Software, vol. 13 pp. 187-195, 1990.
[16] G. Venkatesh, "The Semantic Approach to Program Slicing," In Proc. of the ACM SIGPLAN91 Conf. on Programming Languages Design and Implementation, Toronto, Ontario, pp. 26-28, June 1991.
[17] J. J. Comuzzi and J. M. Hart, "Program Slicing Using Weakest Preconditions," In FME'96, vol. 1051 of Lecture Notes in Computer Science, pp. 557-575, Mar. 1996.
[18] B. Korel and J. Laski, "Dynamic Program Slicing," Information Processing Letters, vol. 29, no. 3, pp. 155-163, 1988.
[19] F. Tip, "A Survey of Program Slicing Techniques," TR CS-R9438, Univ. of Amsterdam, 1994.
[20] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue, "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice," In Proceedings of International Conference on Software Engineering, pp. 422-431, May 1999.

부록 A. 정리의 증명

S를 프로그램 문장, P,Q를 프로그램 상태에 대한 술어(predicate)라고 하면, 다음과 같은 성질들이 성립한다[10,9].

$wp(S, false) \Leftrightarrow false$ (1)

$(P \Rightarrow Q) \Rightarrow (wp(S, P) \Rightarrow wp(S, Q))$ (2)

$wp(S, P) \wedge wp(S, Q) \Leftrightarrow wp(S, P \wedge Q)$ (3)

$wp(S, P) \vee wp(S, Q) \Rightarrow wp(S, P \vee Q)$ (4)

$sp(S, false) \Leftrightarrow false$ (5)

$(P \Rightarrow Q) \Rightarrow (sp(S, P) \Rightarrow sp(S, Q))$ (6)

$sp(S, P) \wedge sp(S, Q) \Rightarrow sp(S, P \wedge Q)$ (7)

$sp(S, P) \vee sp(S, Q) \Leftrightarrow sp(S, P \vee Q)$ (8)

$sp(S, wp(S, Q)) \Rightarrow Q$ (9)

$P \Rightarrow wp(S, true)$라면, $P \Rightarrow wp(S, sp(S, P))$ (10)

A.1 정리 1의 증명

$P \Rightarrow wp(S_2, Q)$

$\models sp(S_2, P) \Rightarrow sp(S_2, wp(S_2, Q))$ {성질 6}

$\models sp(S_2, P) \Rightarrow Q$ {성질 9}

$\models sp(S_1, P) \Rightarrow Q$ {가정으로부터, $sp(S_1, P) \equiv sp(S_2, P)$}

$\models wp(S_1, sp(S_1, P)) \Rightarrow wp(S_1, Q)$ {성질 2}

$\models P \Rightarrow wp(S_1, Q)$ {성질 10}

Q.E.D.

A.2 정리 2의 증명

$P \Rightarrow wp(S_2, Q)$

$\models P \Rightarrow wp(S_1, Q)$

{가정과 정의 2로부터, $wp(S_1, P) \equiv wp(S_2, P)$}

Q.E.D.

A.3 정리 3의 증명

$S_1 \lhd_{(*, Q)} S_2, S_2 \lhd_{(P, *)} S_3$

$\models S_1 \lhd_{(P', Q)} S_2, S_2 \lhd_{(P, Q')} S_3$

{정의 3으로부터, $P' \Rightarrow wp(S_2, Q)$, $P \Rightarrow wp(S_3, Q')$}

$\models ((P \Rightarrow wp(S_3, Q)) \Rightarrow (P \Rightarrow wp(S_2, Q)))$

$\wedge ((P \Rightarrow wp(S_2, Q)) \Rightarrow (P \Rightarrow wp(S_1, Q)))$

{$P'=P$, $Q'=Q$이라고 하면, 정의 3으로부터. }

$\models ((P \Rightarrow wp(S_3, Q)) \Rightarrow (P \Rightarrow wp(S_1, Q)))$

$\models S_1 \lhd_{(P, Q)} S_3$ {정의 3}

Q.E.D.

A.4 정리 4의 증명

정리 3의 증명 방법에 따라 증명할 수 있다. *Q.E.D.*

정 인 상

1983년 ~ 1987년 서울대학교 컴퓨터공학과(학사). 1987년 ~ 1989년 한국과학기술원 전산학과(석사). 1989년 ~ 1993년 한국과학기술원 전산학과(박사). 1994년 ~ 1998년 한림대학교 부교수. 1997년 ~ 1998년 미국 purdue 대학 방문교수. 1999년 ~ 현재 한성대학교 컴퓨터 공학부 부교수. 관심분야는 프로그램 분석, 병렬 및 분산 프로그램 테스팅, 테스트 데이타 자동생성, 모형 검사(model checking)

윤 광 식

1990년 ~ 1995년 한국과학기술원 전산학과(학사). 1995년 ~ 1997년 한국과학기술원 전산학과(석사). 1997년 ~ 현재 한국과학기술원 전자전산학과 전산학전공(박사과정). 2002년 ~ 현재 매크로임팩트(주) 시스템 소프트웨어연구소 주임연구원. 관심분야는 실시간 소프트웨어 설계 및 분석, 병렬 및 분산 프로그램 테스팅, 테스트 자동화 도구 개발

이 완 권

1987년 2월 서울대학교 컴퓨터공학과(학사). 1990년 2월 한국과학기술원 전산학과(석사). 2000년 2월 한국과학기술원 전산학과(박사). 1994년 ~ 현재 전주대학교 정보기술컴퓨터공학부 부교수. 관심분야는 소프트웨어 유지보수, 디자인 메트릭, 객체지향 모델링, 객체지향 테스팅 등.

권 용 래

1969년 서울대학교 문리대학 이학사. 1971년 서울대학교 대학원 이학석사. 971년 ~ 1974년 육군사관학교 전임강사. 1978년 미국 피츠버그대학 이학박사. 1978년 ~ 1983년 미국 Computer Science Corporation 연구원. 1983년 ~ 현재 한국과학기술원 전자전산학과 전산학전공 교수. 2002년 ~ 현재 한국정보과학회 수석부회장. 관심분야는 실시간 병렬 소프트웨어 검증, 실시간 시스템의 객체지향 기술, 고신뢰도 소프트웨어의 품질 보증, 소프트웨어 시험 기법