論文
2-9-8

객체지향 프로그램의 슬라이싱에 관한 연구 (A Study on Program Slicing of Object-Oriented Programs)

김 희 천*
(Hee-Chern Kim)

요 약

프로그램 슬라이싱은 테스팅 및 유지보수 작업의 효율성을 높이기 위한 프로그램 분해 기술이다. 본 논문에서는 객체지향 프로그램 실행부의 슬라이싱 방법을 제안하고 정확성을 분석하였다. 또한 프로그램이 주어졌을 때 클래스 선언부에서 필요 없는 함수와 사용하지 않는 데이터멤버 및 서브객체를 제거하는 클래스 인터페이스 슬라이싱 방법을 제안하고 정확성을 증명하였다. 또 클래스 선언부만을 가지고 주어진 데이터멤버에 영향을 주는 또는 영향을 받는 부분을 추출해 내는 클래스 계층구조의 슬라이싱 방법을 제안하였다. 이러한 기술은 클래스 테스팅 같은 분야에서 정보분석 작업에 이용될 수 있다.

ABSTRACT

Program slicing is a program decomposition technique to improve the effectiveness of program testing and maintenance. In this paper I propose a method for slicing object-oriented programs and show its correctness. Also, I propose the class interface slicing technique which can eliminate irrelevant data members, subobjects, and methods from the class declaration parts of the given program. And I show its correctness. Finally, I propose the class hierarchy slicing technique which can compute the declarative parts affecting or affected by the given data member. These techniques may be used for information analysis in some area such as class testing.

1. 서론

재사용성을 높이기 위하여 객체지향 프로그래밍은 고수준의 기능을 포함하는 다기능 클래스를 사용하도록 권장하고 있다. 재사용성은 바람직하긴 하지만 이로 인한 몇 가지 단점을 가지고 있다[1]. 단점 가운데 하나는 다기능의 클래스를 사용하는 프로그램은 그것이 사용하지 않는 기능에 대한 댓가를 지불해야 한다는 점이다.

확실한 단점은 사용하지 않는 함수를 링크함으로써 생기는 코드 사이즈의 증가이다. 또 다른 단점은 불필요한 데이터멤버와 서브객체를 가지는 객체의 발생이다. 대형 객체는 프로그램의 공간 요건을 증가시키고 또 객체의 생성과 소멸, 페이징 등으로 인한 불필요한 시간 때문에 실행 속도를 감소시킨다.

* 정회원 : 한세대학교 컴정학부 조교수

논문접수 : 2001. 9. 10.
심사완료 : 2001. 9. 21.

※ 본 논문은 '00 한세대학교 교내 학술연구비 지원에 의해 이루어짐.

클래스 재사용과 관련된 이러한 사항들은 클래스 설계자에게 기능과 성능 사이의 선택을 강요한다.

이러한 문제의 해결책은 필요한 많은 기능들을 클래스가 가지도록 설계하고 프로그래머가 프로그램에 맞도록 클래스를 특성화하여 응용 프로그램을 구성하는 것으로 이 과정에 프로그램 슬라이싱 방법이 적용되어 진다. 프로그램 슬라이싱은 특정한 계산에 필요한 프로그램의 부분들을 추출하는 일종의 프로그램 분해 기술이다. 간단히 생각하면 "어떤 프로그램의 문장 s에서 사용된 변수 v의 값에 영향을 줄 수 있는 프로그램 문장들은 무엇인가?"에 대한 답을 제공한다. 프로그램 슬라이싱은 프로그램 테스팅, 프로그램 통합(integration), 프로그램 이해, 소프트웨어 유지보수 및 소프트웨어 메트릭(metric)과 같은 지루하며 실수하기 쉬운 작업들을 도와주는데 그 유용성과 가치가 있다.

본 논문에서 제안하는 슬라이싱 연구는 객체지향 프로그램의 종속성 표현에 기반을 두고 있는데 3.1절에서 객체지향 프로그램의 슬라이싱에 관해, 3.2절에서는 클래스 인터페이스 슬라이싱에 관해, 3.3절에서는 클래스 계층구조 슬라이싱에 관해 제안을 하며 그 내용을 설명한다.

2. 관련 연구

Weiser는 처음으로 슬라이스 개념을 소개하였다[2]. 그의 슬라이싱 알고리즘은 제어 흐름 그래프(control flow graph)상에서 데이터 흐름 분석을 이용한다. 프로그램 P의 슬라이스는 슬라이싱 기준(criterion) C=<s, V>과 관계가 있다. 여기서 s와 V는 P에 있는 프로그램 포인트(또는 문장)와 변수 집합이다.

Ottenstein 등은 그래프 도달성 알고리즘(graph reachability algorithm)을 이용하여 즉, 프로그램을 프로그램 종속성 그래프로 표현 한 후 이것의 종속성 에지를 순회(traverse)함으로써 효율적으로 슬라이스를 계산할 수 있음을 보였다[3].

Harrold등은 시스템 종속성 그래프에 그래프 도달성 알고리즘을 적용한 다중 프로시저 슬라이싱 알고리즘을 연구하였다[4]. Harrold 등이 제안한 알고리즘은 호출 문맥을 반영하기 위해 호출 부분의 요약정보(summary information)를 이용하므로 정확한 슬라이스를 계산해 낸다. 알고리즘은 2 단계로 되어 있으며 단계 1에서는 슬라이싱 기준 문장 s에서 시작하여 에지를 순회하는데, 역방향 슬라이싱에서는 인자-출력에지를 제외한 모든 에지를, 순방향 슬라이싱에서는 인자-입력에지와 호출에지를 제외한 모든 에지를 따라 순회한다. 단계 2에서는 단계 1에서 도달된 노드들에서 시작하여, 역방향 슬라이싱은 인자-입력에지와 호출에지를 제외한 모든 에지를, 순방향 슬라이싱은 인자-출력에지를 제외한 모든 에지를 따라 순회한다.

Harrold등의 연구[4]가 절차적 프로그램을 표현하고 이것에 대한 슬라이싱 방법을 연구한 것인데 반해 Larsen등의 연구[6]는 Harrold 등의 연구를 토대로 프로그램 표현을 객체지향 프로그램을 지원하도록 확장하고 여기에 슬라이싱을 적용한 것이다.

Harrold등의 연구에서 슬라이싱 기준은 <p,x>로 p는 프로그램 문장이며 x는 p에서 참조되거나 정의되는 변수이다. 객체지향 소프트웨어의 개발자는 객체 상태의 정의와 참조를 위한 인터페이스를 제공하여 사용자로 하여금 객체의 상태를 규정짓는 데이터멤버들을 직접 다루지 않게 하고 있다. 따라서 객체지향 소프트웨어에서 단순히 값을 리턴하는 메소드의 호출은 변수의 참조라고 볼 수 있다. 이러한 점을 고려하여 Larsen등의 연구에서 슬라이싱 기준은 Harrold등의 연구와는 약간 다른데, 슬라이싱 기준은 <p,x>이며 여기서 p는 문장이고 x는 p에서 정의 또는 참조된 변수이거나 p에서 호출되는 메소드이다. Harrold등의 연구와 마찬가지로 시스템 종속성 그래프가 호출 문맥을 위해 표시되는 요약에지를 가지므로 슬라이스를 계산하기 위해 2 단계 알고리즘을 이용한다.

역방향 슬라이싱의 첫 번째 단계에서는 요약에지를 통해 호출노드와의 이행 종속성을 추적할 수 있으며, 두 번째 단계에서는 인자-출력에지를 따라 호출되는 메소드로 내려간다. 순방향 슬라이싱도 Harrold등의 연구와 동일하다.

3. 객체지향 프로그램의 슬라이싱 방법

프로그램 실행부의 슬라이싱은 기존 절차적 프로그램의 슬라이싱과 동일한 의미를 가진다. 간단히 생각하면 "어떤 프로그램의 문장 s에 있는 변수 v의 값에 영향을 줄 수 있는 프로그램 문장들은 무엇인가?"에 대한 답을 제공하여 필요 없는 코드를 제거하는 것이다. 본 논문에서는 프로그램 실행부의 슬라이싱을 간단히 "프로그램 슬라이싱"이라 지칭하겠다.

클래스 선언부의 슬라이싱은 두 가지로 나누어 생각할 수 있다.

첫 번째는 프로그램실행부가 주어졌을 때 사용되지 않는 함수와 데이터멤버 및 서브객체를 클래스 선언부에서 제거하는 것이다. 기존의 문헌을 살펴보면 절차적 프로그램에서 사용하지 않는 함수를 제거하는 방법으로 "인터페이스 슬라이싱"이라는 용어를 사용한 예가 있다[7]. 본 논문에서는 "클래스 인터페이스 슬라이싱"이라고 지칭하겠다.

두 번째는 클래스 선언부만을 가지고 주어진 데이터멤버에 영향을 주는 또는 영향을 받는 부분을 추출해 내는 것으로 본 논문에서 "클래스 계층구조 슬라이싱"이라 지칭하겠다.

3.1 프로그램 슬라이싱

아래에 프로그램 역방향 슬라이싱을 위한 알고리즘 3.1을 제안하였다. [8]에서 제안한 객체지향 프로그램 종속성 모델을 이용하며 다중 프로시저 슬라이싱을 지원한다. 본 종속성 그래프는 변수 단위의 종속성을 표현하며 모듈화 되어있다. 알고리즘 3.1의 슬라이싱 기준은 포트 집합이며 포트는 문장에서 사용된 변수를 의미한다. 변수의 사용은 곧 문장의 사용을 의미하므로 다음을 가정한다.

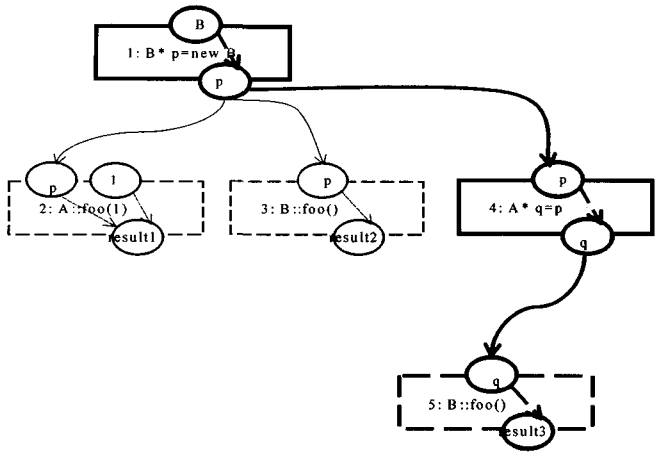
(가정 1) 임의의 노드 n_1에 대해서 $USE(n_1) \cup DEF(n_1)$에 속하는 한 개 이상의 포트가 마크되면 노드도 마크한다.

(가정 2) 통합된 표현을 지원하기 위해 다음을 가정한다. 호출노드 n_1의 $USE(n_1)$에 속하는 포트로부터 호출되는 메소드헤더노드 n_2의 $USE(n_2)$에 속하는 해당 포트로, 또 $DEF(n_2)$에 속하는 포트로부터 $USE(n_1)$에 속하는 해당포트로 도달 가능하다.

[알고리즘 3.1] 객체지향 프로그램의 슬라이싱 알고리즘

```
algorithm Program_Slicing(G(P), C, flag)
input : G(P), program(or method) P's dependence
graph
        C, a set of ports in G(P)
output : S, Slice(P,C)'s dependence graph
{
  /* 단계 1 : P 내부에서의 슬라이싱 */
  Traverse edges backward using cd, du, or ud
   edges and mark all ports and nodes of G(P)
   which can (reflexive and transitively) reach any
   unmarked port in C ;
  Let the symbols marked be called S;
  /* 단계 2 :P를 호출한 부분의 슬라이싱 */
  if flag and P is not main program {
     Get G(m1 which calls P);
     Program_Slicing(G(m1), use_port corresponding
      marked initial_define_port, 1);
     Add returned graph to S;
  }
  /* 단계 3 :P에서 호출되는 부분의 슬라이싱 */
  for each marked call node m1 which is called by P {
     Get G(m2);
     Program_Slicing(G(m2), final_use_ports
      corresponding marked def_port, 0);
     Add returned graph to S;
  }
  return S;
}
```

종속성은 모듈 단위로 표현된다. 따라서 알고리즘은 3 단계로 이루어진다. 첫 번째 단계는 메소드 P 내부의 슬라이싱이며 두 번째와 세 번째 단계는 P를 호출하는 메소드 a에서의 슬라이싱과 P에 의해 호출되는 메소드 b에서의 슬라이싱이다. P 메소드헤더에서 마크된 초기정의포트가 있을 때, 호출하는 모듈 a에서의 슬라이싱은 a에 존재하는 호출노드에서 해당 참조포트를 기준으로 역방향 슬라이싱을 한다. P의 호출노드에서 마크된 정의포트가 있을 때, 호출되는 메소드 b의 슬라이싱은 b 메소드헤더의 해당 최종참조포트를 기준으로 역방향 슬라이싱을 한다. [그림 1]은 포트 (result3, 5)를 기준으로 한 역방향 슬라이



[그림 1] 객체지향 프로그램의 슬라이싱
[Fig. 1] Slicing of a object-oriented program

싱을 설명한 것으로 호출노드 5에서 마크된 정의포트에 해당하는 즉, 호출되는 B::foo() 메소드헤더의 최종참조포트를 기준으로 역방향 슬라이싱을 한다.

프로그램 테스팅 과정에서 프로그램 포인트 p에서 사용된 변수 x의 오류를 발견했다고 하자. 이 때 (x, p)를 기준으로 역방향 슬라이싱을 수행하면 오류 발견에 필요한 부분만을 고려할 수 있다.

순방향 슬라이싱은 역방향 슬라이싱의 반대로 생각하면 되는데 예를 들어 프로그램에서 문장 p를 삭제하려 한다고 하자. 그러면 p의 영향력에 관한 이해가 우선되어야 한다. 이 때 순방향 슬라이싱을 수행하면 p에 종속되는 문장과 변수만을 계산할 수 있다.

Larsen과 Harrold는 객체지향 프로그램의 시스템 종속성 그래프와 슬라이싱 알고리즘(이것을 "LH알고리즘"이라 하자)을 연구하였다[6]. LH알고리즘은 Horwitz, Reps와 Binkley의 2 단계 다중 프로시쥬어 알고리즘[4]을 그대로 적용시킨다. LH알고리즘에 의해 계산된 슬라이스가 있을 때, 여기에 포함되는 모든 노드들이 알고리즘 3.1에 의해 계산된 슬라이스에 포함됨을 보이겠다.

[정의 3.1] LH알고리즘을 이용하여 <n>(n은 프로그램 문장)을 기준으로 계산한 슬라이스에 속하는 노드 집합을 H라 하고 단계 1에서 마크된 노드 집합을 H1, 단계 2에서 마크된 노드 집합을 H2라 하자.

그러면 $H_1 \cap H_2 = \varnothing$, $H_1 \cup H_2 = H$, $\{n\} \subseteq H_1$의 성질을 가진다.

[정의 3.2] 알고리즘 3.1을 이용하여 <V, n>(V는 n에서 정의 또는 참조된 변수 집합)을 기준으로 계산한 슬라이스에 속하는 심볼 중 노드 집합만을 K라 하고 단계 1, 2, 3에서 마크된 노드 집합을 각각 K_1, K_2, K_3라 하자.

그러면 $K_1 \cap K_2 \cap K_3 = \varnothing$, $K_1 \cup K_2 \cup K_3 = K$, $\{n\} \subseteq K_1$의 성질을 가진다. 따라서 $K_1 \cup K_2 = H_1$과 $K_3 = H_2$임을 보임으로써 K=H임을 증명하도록 한다.

[정리 3.1] 알고리즘 3.1의 단계 1과 단계 2에서, 포트p로부터 <V, n>에 도달 가능하면 $p \in K_1 \cup K_2$이다. 또한 역이 성립한다.

(증명) 자명하다. □

[정리 3.2] 알고리즘 3.1의 단계 3에서, 포트 $p' \in K_1 \cup K_2$가 존재하여 포트p로부터 p'에 도달가능하고 $p \notin K_1 \cup K_2$이면 $p \in K_3$이다.

(증명) 자명하다. □

[정리 3.3] $K_1 \cup K_2 = H_1$

(증명) $p \in H_1$이라 가정하자. HRB 알고리즘은 단계 1에서 호출되는 부분의 형식인자-출력노드로부터 호출하는 부분의 실인자-출력노드로 향하는 인자-출력 에지를 제외한 에지를 이용하여 n에 도달 가능한 p를 마크한다. 이것은 알고리즘 3.1의 단계 1과 단계 2에서 포트p로부터 <V, n>에 도달 가능하다는 것과 일치한다. 그러므로 정리3.1로부터 $p \in K_1 \cup K_2$임을 알 수 있다. □

[정리 3.3] $K_3 = H_2$

(증명) $p \in H_2$라 가정하자. HRB 알고리즘은 단계 2에서 호출 에지와 호출하는 부분의 실인자-입력노드로부터 호출되는 부분의 형식인자-입력노드로 향하는 인자-입력 에지를 제외한 에지를 이용하여 H_1에 도달 가능한 p를 마크한다. 따라서 슬라이스에 새로 추가되는 노드 p는 인자-출력에지를 이용하여 H_1에 도달 가능하다. 그렇지 않다면 $H_1 \cap H_2 = \emptyset$이므로 $p \in H1$이 되기 때문이다. 즉 $p' \in H_1$이 존재하여 포트p로부터 p'에 도달가능하고 $p \notin H_1$이라는 사실을 의미한다. 이것은 알고리즘 3.1의 단계 3에서 포트 $p' \in K_1 \cup K_2$가 존재하여 포트 p로부터 p'에 도달가능하고 $p \notin K_1 \cup K_2$이라는 사실과 일치한다. 따라서 정리3.2로부터 $p \in K_3$임을 알 수 있다. □

[정리 3.3] $K = H$

(증명) 정리3.3으로부터 $K_1 \cup K_2 = H_1$이다. 또 정리3.4로부터 $K_3 = H_2$이다.

정의 3.1과 3.2로부터 $K = K_1 \cup K_2 \cup K_3 = H_1 \cup H_2 = H$임으로 증명된다. □

3.2 클래스 인터페이스의 슬라이싱 방법

여기서는 클래스 선언부에 관한 두 가지 슬라이싱 방법 가운데 프로그램이 주어졌을 때, 클래스 선언부에서 프로그램이 사용하지 않는 메소드와 데이터멤버 및 서브객체를 제거하는 방법을 제안한다. 본 논문에서는 이것을 클래스 인터페이스 슬라이싱이라고 하였다.

3.2.1 클래스 인터페이스 슬라이싱 알고리즘

클래스 인터페이스 슬라이싱은 주어진 프로그램에서 사용되지 않는 데이터 또는 멤버함수들을 제거하고자 하는 것으로 프로그램에서 d.m 또는 d->m과 같이 어떤 멤버에 대한 접근이 있을 때 클래스 구조에서 이것을 해결하는 경로를 파악함으로써 슬라이스를 구성해 간다. 먼저 필요한 정의를 기술하였으며 클래스 인터페이스 슬라이싱을 위한 알고리즘 3.2를 제안한다. 알고리즘의 정확성에 대해서는 3.2.2에서 설명하였다.

클래스 계층구조에서의 계승 경로는 클래스 구조로부터 인스탄스화(instantiation)된 객체가 가질 수 있는 모든 계승경로로 이루어진 집합이다. 클래스 계층구조 α의 그래프에서 메소드 요약 표현은 클래스 인터페이스 슬라이싱에 사용되지 않는다. 그러면 $CHG(\alpha) = <C, M, \emptyset, D, \rightarrow^{member}, \Rightarrow, \emptyset>$라 할 수 있으며, 계승 경로의 집합은 다음과 같이 정의된다.

[정의3.3] 클래스 계층구조에서의 계승경로 집합, IP_Set(α)
(i) for all $c \in C$, $c \in IP\_Set(\alpha)$
(ii) $(x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n \rightarrow y) \in IP(\alpha)$ if $(x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n) \in IP\_Set(\alpha)$, $n \geq 0$, $x_i, y \in C$ and $(x_n, y) \in \rightarrow$
(iii) $(x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n \dashrightarrow y) \in IP(\alpha)$ if $(x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n) \in IP\_Set(\alpha)$, $n \geq 0$, $x_i, y \in C$ and $(x_n, y) \in \dashrightarrow$

멤버의 접근을 해결하기 위해선 객체의 정적타입 및 동적타입과 관련된 계승 경로가 분석되어야 하므로 다음을 정의한다.

[정의3.4] 객체의 계승 경로
클래스 계층구조α를 사용하여 인스탄스화된 객체 d의 동적 타입을 x_0, 정적 타입을 x_n이라 할 때 d의 계승 경로 $IP(d) \in IP_Set(\alpha)$는 $x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n$이다.

$(x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n) \in IP\_Set(\alpha)$, $n \geq 0$을 간단히 $(x_0 \Rightarrow {}^{*}x_n)$라 표기하고 $(x_0 \Rightarrow x_1 \Rightarrow ... \Rightarrow x_n) \in IP\_Set(\alpha)$, $n \geq 1$을 $(x_0 \Rightarrow {}^{+}x_n)$라 표기하자.

[정의3.5] $<(x_0 \Rightarrow {}^{*}x_n) < (x_0 \Rightarrow {}^{*}x_n \Rightarrow {}^{+}y) \in IP_Set(\alpha)$라 정의한다.

멤버의 정적 경로는 정적으로 결정되어지는 멤버를 해결하기 위한 클래스 계층구조의 계승 경로이며 아래와 같이 정의되어 진다.

[정의3.6] 멤버의 정적 경로 (Static_Path)
α를 클래스 계층구조, CHG(α)=<C, M, ∅, D, $\rightarrow^{member}$, ⇒, ∅>를 클래스 계층구조 그래프, d를 인스탄스화된 객체, $IP(d)=x_0 \Rightarrow {}^{*}x_n \in IP\_Set(\alpha)$, $m \in D \cup M$을 메소드 또는 데이터멤버라 할 때, $(x_n, m) \in \rightarrow^{member}$ 일 때, $Static\_Path(\alpha, IP(d), m) = (x_0 \Rightarrow {}^{*}x_n)$
$(x_n, m) \notin \rightarrow^{member} \wedge x_n \Rightarrow {}^{*}y \wedge (y, m) \in \rightarrow^{member}$ 일 때, $Static\_Path(\alpha, IP(d), m) = min(x_0 \Rightarrow {}^{*}x_n \Rightarrow {}^{*}y)$

동적 경로는 멤버가 동적으로 결정되어지는 경우 그것을 해결하기 위한 클래스 계층구조의 계승 경로이며 아래와 같이 정의되어 진다.

[정의3.7] 멤버의 동적 경로 (Dynamic_Path)
α를 클래스 계층구조, CHG(α)=<C, M, ∅, D, $\rightarrow^{member}$, ⇒, ∅>를 클래스 계층구조 그래프, d를 인스탄스화된 객체, $IP(d)=x_0 \Rightarrow {}^{*}x_n \in IP\_Set(\alpha)$, $m \in M$을 메소드라 하고 $Static_Path(\alpha, IP(d), m)$에 의해 결정된 m이 가상 메소드이면
(i) $(x_0, m) \in \rightarrow^{v}$ 일 때, $Dynamic\_Path(\alpha, IP(d), m) = (x_0)$
(ii) $(x_0, m) \notin \rightarrow^{v} \wedge x_0 \Rightarrow {}^{*}y \wedge (y, m) \in \rightarrow^{v}$ 일 때, $Dynamic\_Path(\alpha, IP(d), m) = min(x_0 \Rightarrow {}^{*}y)$

클래스 계층구조 α의 부계층구조 α'는 α에서 클래스, 메소드, 데이터멤버, 및 계승 관계가 삭제된 형태로 직관적으로 볼 때 α의 부분집합이라 할 수 있다.

[정의3.8] 클래스 부계층구조(Class Subhierarchy)
α와 α'를 클래스 계층구조라 하고 CHG(α)=<C, M, ∅, D, $\rightarrow^{member}$, ⇒, ∅>와 CHG(α')=<C', M', ∅, D', $\rightarrow^{member}$, ⇒'>를 각각 α와α'의 클래스 계층구조 그래프라 하자. α'가 다음의 성질을 가지면 α'를 α의 클래스 부계층구조라 한다.
(i) $C' \subseteq C$, (ii)$M' \subseteq M$, (iii)$D' \subseteq D$, (iv)$\rightarrow^{member}$, $\subseteq \rightarrow^{member}$, (v)$\Rightarrow' \subseteq \Rightarrow$

[알고리즘 3.2] 클래스 인터페이스 슬라이싱 알고리즘

α를 프로그램 P에서 사용하는 클래스 계층구조라 하면

```
1  algorithm Hierarchy_Interface_Slice(CHG(α), P) :
   CHG(α');
2  input : CHG(α) and Program P
3  output : CHG(α'), (CHG(α), P)'s Interface Slice
4  {
5  for each implicit or explicit object creation
   construct e in P do
6    mark Class;
7  for each implicit or explicit typecast expression
   (T*)d, (T&)d or (T)d in P do
8    mark inheritance path such that StaticType(d)⇒
     *T;
9    for each implicit or explicit expression d.m or
     d->m in P do {
10     compute IP(d);
11     for each possible IP(d) {
12       compute Static_Path(α, IP(d), m);
13       mark generated path p and corresponding
         method m;
14       mark compared path p'such that p'>p;
15       if m is virtual method {
16         compute Dynamic_Path(α, IP(d), m);
17         mark generated path p and corresponding
           method m;
18         mark compared path p' such that p'> p;
19       }
20     }
21   }
22   return CHG(α') which consists of marked
     components in CHG(α);
23 }
```

3.2.2 분석

프로그램 P에서 사용하는 클래스 계층구조 α의 인터페이스 슬라이스 α'는 α의 부계층구조이며 다음의 성질을 가져야한다.

(ⅰ) α'는 P에서 인스탄스화되는 객체가 가지는 모든 계승 경로를 가진다.
(ⅱ) α에서의 객체 d의 멤버 m을 결정할 때, 그 결정 결과가 α'에서 그대로 유지된다.

다음의 정리로부터 3.2.1에서 기술한 알고리즘의 정확성을 증명한다.

[정리3.6] α'는 P에서 인스탄스화되는 객체가 가지는 모든 계승 경로를 가진다.

(증명) 멤버의 정적 경로에 대한 정의3.6으로부터 Static_Path(α, IP(d), m) ≥ IP(d)라는 사실을 알 수 있다. 라인13에서 결정된 객체의 정적 경로에 포함되는 클래스와 계승 에지들이 인터페이스 슬라이스에 포함되므로 라인10에서 계산되는 모든 IP(d)가 인터페이스 슬라이스에 포함된다. 따라서 α'는 모든 IP(d)의 계승 경로를 가진다. □

[정리3.7] Static_Path(α, IP(d), m) = Static_Path(α', IP(d), m)

(증명) 먼저 Static_Path(α, IP(d), m)∈IP_Set(α')이며 α'에서 (xn, m)→$^{member'}$를 만족함을 보인다.
α에서 IP(d)=x_0⇒*x_i라 하자. x_0⇒*x_i∈IP_Set(α')이다. Static_Path(α, IP(d), m)=x_0⇒*x_i⇒*x_n라 하면 정적 경로의 정의3.6으로부터 α에서 (x_n, m)→^{member}이다. 라인13으로부터 x_0⇒*x_i⇒*x_n∈IP_Set(α')이며 α'에서 (x_n, m)→$^{member'}$이다.
x_0⇒*x_i⇒*x_n이 Static_Path(α', IP(d), m)임을 보인다. 라인14로부터 α'에서 x_0⇒*x_i⇒*y이고 (y, m)→$^{member'}$인 y'들이 존재하고 이 가운데 α에서 (y, m)→member를 만족하는 가장 작은 것을 min(x_0⇒*x_i⇒*y)∈IP_Set(α')라 하자. 그러면 α'⊂α이므로 min(x_0⇒*x_i⇒*y)∈IP_Set(α)이고 (y, m)→^{member}이다. 정적 경로의 정의3.6에 의해 α에서 x_0⇒*x_i⇒*x_n≤min(x_0⇒*x_i⇒*y)이다. 따라서 x_0⇒*x_i⇒*x_n∈IP_Set(α')이며 α'에서 (x_n, m)→$^{member'}$이므로 Static_Path(α', IP(d), m)=x_0⇒*x_i⇒*x_n이다. □

[정리3.8] Dynamic_Path(α,IP(d), m) =Dynamic_Path(α', IP(d), m)

(증명) 정리3.7와 같은 방법으로 증명될 수 있다. □

[정리3.9] α에서의 객체 d의 멤버 m을 결정할 때, 그 결정 결과가 α'에서 그대로 유지된다.

(증명) 정리3.7와 3.8로부터 객체 d의 멤버 m을 결정할 때 m이 가상 함수이면 ① Static_Path(α, IP(d), m) = Static_Path(α', IP(d), m)와 ② Dynamic_Path(α, IP(d), m) = Dynamic_Path(α', IP(d), m)를 만족하며 그렇지 않은 경우 ①을 만족하므로 증명된다. □

3.3 클래스 계층구조의 슬라이싱 방법

메소드들은 명시적인 호출 관계가 없더라도 데이터멤버를 통해 서로 영향을 주거나 받을 수 있으므로 데이터멤버를 통한 상호 종속성이 존재한다. 본 절에서는 클래스 선언부만을 가지고 주어진 데이터 멤버에 영향을 주는, 또는 영향을 받는 부분을 추출해 내는 방법을 제안하며 이것을 클래스 계층구조 슬라이싱이라 지칭한다. 데이터멤버를 통한 상호 종속성을 분석하기 위해서는 기존의 클래스 계층구조 그래프를 확장하여 메소드들이 데이터멤버들을 참조하거나 정의하는 흐름을 추가하여야 한다. 클래스 계층구조 슬라이싱 알고리즘 3.3에서 사용되는 클래스 데이터 흐름 그래프(CDFG)는 [8]에서 정의된 것이다.

클래스 계층구조의 슬라이싱은 주어진 프로그램 없이 클래스 선언부만을 가지고 수행되는 슬라이싱 방법이다. 이 방법은 클래스에서 정의된 어떤 데이터멤버에 영향을 주는 또는 그 데이터멤버로부터 영

향을 받는 메소드와 데이터멤버를 분석하기 위한 것으로 각각 클래스 데이터 흐름 그래프의 데이터멤버 포트에서 역방향 또는 순방향 슬라이싱을 수행한다.

아래에 역방향 슬라이싱을 알고리즘을 제안한다.

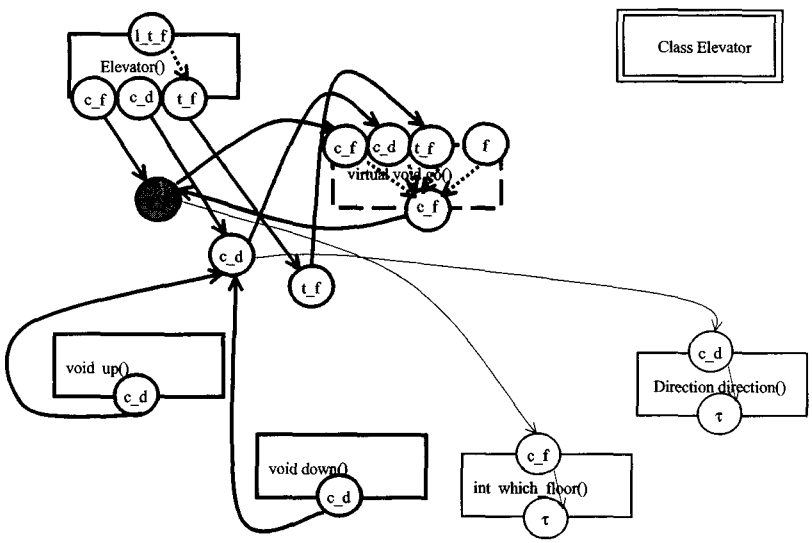
[알고리즘 3.3] 역방향 클래스 계층구조 슬라이싱 알고리즘

algorithm Hierarchy_Slicing(CDFG(α), D) : CDFG (α');
input : CDFG(α)
D, a set of data member in CDFG(α)
output : CDFG(α'), (CDFG(α),D)'s Class Hierarchy Slice
{
Mark all ports and methods of CDFG(α) which can reach data port in D using du or td edges;
Mark traversed edges;
Return marked symbols;
}

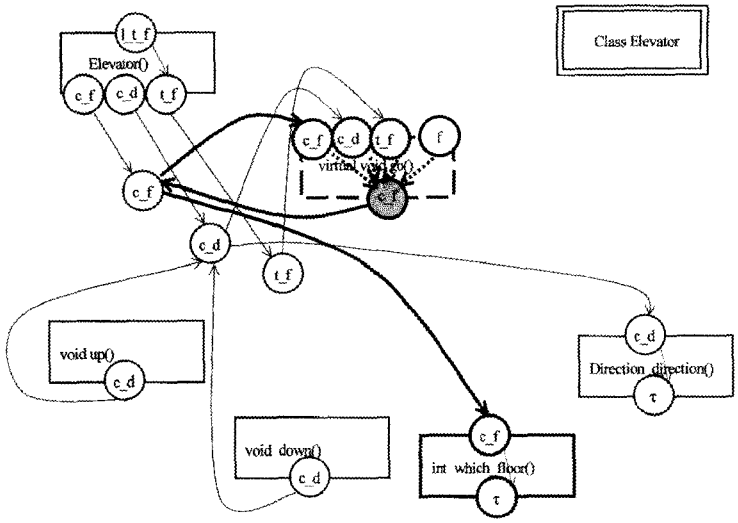
data를 클래스에서 정의된 데이터멤버라 할 때, 역방향 슬라이싱의 슬라이싱 기준을 <data>라 제안하며 이것은 객체의 상태를 결정하는 데이터멤버에 직접 또는 간접적으로 영향을 주는 메소드 집합을 구하는 것이다. 이러한 정보들은 클래스 테스팅을 위한 테스트케이스 생성에 이용될 수 있다.

예를 들어 [그림 2]에서 클래스 Elevator에서 정의된 데이터멤버 c_f 포트를 기준으로 역방향 슬라이싱을 수행하면 메소드 go()가 데이터멤버 c_f에 영향을 줄 수 있다는 사실을 알 수 있고, 계속해서 up(), down()이 go()에 영향을 미침으로써 간접적으로 c_f에 영향을 주고 있음을 알 수 있다.

3.1절에서 기술한 객체지향 프로그램의 순방향 슬라이싱과 마찬가지로 클래스 계층구조의 순방향 슬라이싱을 정의할 수 있다. 순방향 슬라이싱에서 슬라이싱 기준은 <m, def_port>로 제안한다. m은 메소드, def_port는 m에서 정의하는 데이터이다. 순방향 슬라이싱은 메소드 m에 종속되는 메소드 집합을 구하는 것이다. 클래스 회귀 테스팅 분야에서는 메소드 수정시 클래스 계층구조의 순방향 슬라이싱을 이용하여 메소드간의 종속관계를 분석함으로써 테스트케이스의 재사용성을 고려하게 된다.



[그림 2] 역방향 클래스 계층구조 슬라이싱
[Fig. 2] Backward slicing of a class hierarchy



[그림 3] 순방향 클래스 계층구조 슬라이싱

[Fig. 3] Forward slicing of a class hierarchy

[그림 3]의 예를 보자. 클래스 Elevator에서 정의된 메소드 go()가 정의하는 c_f 포트를 기준으로 슬라이싱을 수행하면 메소드 go(), which_floor()가 슬라이스에 포함된다. 이 사실은 메소드 go()를 수정할 때, go()의 정확성을 다시 검증하여야 하며 또한 which_floor()에 미치는 영향도 함께 검증하여야한다는 점을 의미한다.

4. 결론

객체지향 프로그램의 종속성 모델에 기초하여, 본 논문에서는 특정한 계산에 필요한 부분들을 추출하는 프로그램 분해 기술인 슬라이싱에 관해 연구하였으며 그 결과와 유용성은 다음과 같다.

객체지향 프로그램은 프로그램 실행부와 클래스 선언부로 나뉜다. 본 논문에서는 프로그램이 주어졌을 때 클래스 선언부에서 필요 없는 함수와 사용하지 않는 데이터멤버 및 서브객체를 제거하는 클래스 인터페이스 슬라이싱 방법을 제안하였고 정확성을 증명하였다. 또 클래스 선언부만을 가지고 주어진 데이터멤버에 영향을 주는 또는 영향을 받는 부분을 추출해 내는 클래스 계층구조의 슬라이싱 방법을 제안하였다.

고수준의 기능을 포함하는 클래스는 재사용성을 시키나 이러한 클래스를 사용하는 프로그램은 사용하지 않는 함수를 링크하며 불필요한 데이터를 가지는 객체를 생성하는 단점을 가진다. 대형 객체는 프로그램의 공간 요건을 증가시키고, 객체의 생성과 소멸로 인한 불필요한 시간 때문에 실행 속도를 감소시킨다. 이러한 문제의 해결책이 클래스 인터페이스 슬라이싱이다. 한편 클래스 계층구조의 슬라이싱 방법을 이용하면 객체 상태에 영향을 미치는 메소드 간의 종속성 관계를 밝힐 수 있어 클래스 테스팅 분야 유용하게 쓰일 수 있을 것이다.

※ 참고문헌

[1] F. Tip, J. Choi, J. Field, and G. Ramalingam, "Slicing Class Hierarchy in C++," *Proceedings of OOPSLA* '96, 1996.

[2] M. Weiser, "Program slicing," *IEEE Transaction on Software Engineeing*, July 1984.

[3] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *ACM SIGPLAN Notices*, May 1984.

[4] S. Horwitz, Thomas Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transaction on Programming Languages and Systems*," Jan. 1990.

[5] S. Horwitz and Thomas Reps, "The Use of Program Dependence Graphs in Software Engineering," *Proceedings of the 14th International Conference on Software Engineering*, May 1992.

[6] L. Lasen and M. J. Harrold, "Slicing Object-Oriented Software," *Proceedings of the 18th International Conference on Software Engineering*, May 1996.

[7] J. Beck and D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *Proceedings of '93 International Conference on Software Maintenance*, 1993.

[8] 김희천, "객체지향 프로그램의 종속성 모델과 프로그램 슬라이싱을 이용한 클래스 테스팅 방법", 1998.3, 소프트웨어공학회지, 제11권, 1호, pp. 57~73

[9] Ian Sommerville, Software Engineering, Addison-Wesley, 1996.

[10] H. Kim and C. Wu, "A Class Testing Technique Based on Data Bindings," *Proceedings of '96 APSEC*, Dec. 1996.

김 희 천

서울대학교대학원
전산과학전공 박사
㈜고구려멀티미디어통신
연구원
㈜E&E 책임연구원 역임
현 한세대학교 컴정학부
조교수
현 ㈜엘로우위즈 기술고문
관심분야 : XML, UML,
소프트웨어공학
khc@hansei.ac.kr