슬라이스 기반 복잡도 척도

문 유 미[†] · 최 완 규[††] · 이 성 주[†††]

요 약

본 논문은 데이터 슬라이스에서의 데이터 토큰들의 정보 흐름에 기초하여 프로그램에서의 정보 흐름을 모델링하는 SIFG(Slice-based Information Graph)를 개발하였다. 다음으로, SIFG에서의 정보 흐름의 복잡도 측정를 통해서 프로그램의 복잡도를 측정하기 위해 SCM(Slice-based Complexity Measure)을 정의하였다. SCM은 Briand가 제시하는 복잡도 메트릭에 필요한 특성들을 만족하였고, 그리고 기존 척도들과는 달리, SCM은 프로그램 내에서의 제어와 데이터 흐름뿐만 아니라 프로그램의 물리적 크기를 반영하는 측정이 이루어졌다.

A Slice-based Complexity Measure

Yu-Mi Moon[†] · Wan-Kyoo Choi[††] · Sung-Joo Lee[†††]

ABSTRACT

We developed a SIFG (Slice-based Information Graph), which modelled the information flow on program on the basis of the information flow of data tokens on data slices. Then we defined a SCM (Slice-based complexity measure), which measured the program complexity by measuring the complexity of information flow on SIFG. SCM satisfied the necessary properties for complexity measure proposed by Briand et al. SCM could measure not only the control and data flow on program but also the physical size of program unlike the existing measures.

키워드 : 데이터슬라이스(data slice), 슬라이스 기반 정보흐름 그래프(slice-based information flow graph), 프로그램 복잡도(program complexity)

1. 서 론

전체 시스템 비용에서 소프트웨어 비용의 증가에 따라서, 소프트웨어의 심리적 복잡도의 측정에 대한 관심이 증가해 왔다. 소프트웨어 개발 단계에서 소프트웨어 측정은 프로그래머에게 중요한 정보를 제공할 수 있고, 이해하기 쉬운 코드의 작성에 도움이 될 수 있다[1].

그러나 측정 분야에서 대부분의 기존의 척도들은 프로그램의 표면적인 특징들만을 고려하고 있다고 말할 수 있다. 프로그램의 이해에 지대한 영향을 미치는 것은 프로그램의 표면적인 특징들이 아니라 프로그램의 정보 흐름(information flow)의 특징이다[1].

프로그램 이해에 대한 대부분의 어려움은 프로그래머가 프로그램의 이해를 위해서 사용하는 문장(statement)들의 묶음(grouping)과 소스 프로그램 리스트와 프로그래밍 환경이 제시하는 문들의 묶음과의 차이 때문이다[1]. Weiser[2, 3]에 의하면 프로그래머들이 프로그램을 이해할 때 순차적 관계 이외의 방법으로 문장(statement)들을 묶는 경향이 있다고 하였다. 일반적으로 묶음을 위한 기준은 정보의 흐름인 데이터와 제어의 흐름에 관계가 있다. 이런 정보가 슬라이스(slice)[2]에 분명하게 표현되므로, 정보 흐름의 복잡도는 슬라이스를 이용하여 가장 쉽고 분명하게 측정될 수 있다[1].

슬라이스는 프로그램의 특정 위치에서 한 변수의 값에 영향을 미치는 문장들의 묶음에 대한 추상화이다[2-4]. 프로그래머가 프로그램을 이해하고 디버깅할 때 슬라이스를 사용한다면[3], 프로그램의 이해는 슬라이스에서의 정보 흐름을 탐색하는 과정이라고 할 수 있다. 따라서 슬라이스 복잡도의 측정은 의미가 있다[1].

프로그램에서 정보 흐름을 표현하려는 기존의 연구들은 주로 제어 흐름 그래프(control flow graph)에 기초하고 있으며, 프로그램의 응집도를 측정하는 슬라이스에 기초한 대부분의 척도[4-8]들은 슬라이스에 나타나는 정보의 흐름을 측정하는 것이 아니다.

그러므로 본 연구에서는 슬라이스에서의 정보 흐름의 복잡도 측정을 통하여 프로그램에서의 정보 흐름의 복잡도를 측정하는 슬라이스 기반 복잡도 척도(Slice-based Complexity Measure : SCM)를 제안한다.

† 준 회 원 : 조선대학교 컴퓨터공학부
†† 정 회 원 : 광주대학교 컴퓨터전자통신공학부
††† 정 회 원 : 조선대학교 컴퓨터공학부

논문접수 : 2001년 4월 16일, 심사완료 : 2001년 5월 29일

슬라이스 기반 복잡도 척도가 의미 있는 측정을 제공하기 위해서는 척도가 엄밀하게 정의되어야하고, 측정을 위한 속성을 정확히 반영해야하고, 이런 속성들을 포착하는 모델에 근거해야한다[9-11].

본 연구의 목적은 적합한 슬라이스 기반 복잡도 측정 모델을 정의하며, 슬라이스 기반 복잡도의 정량화를 위해서 정의된 모델을 사용하는 척도를 개발하고 검증하는 것이다.

본 연구에서는 정보흐름에 따른 미세 변화(elementary change)들을 잘 표현할 수 있도록 슬라이스 상의 데이터 토큰들에서의 정보흐름을 통해 프로그램의 정보흐름을 모델링하기 위해 슬라이스 기반 정보흐름 그래프(Slice based information flow graph : SIFG)를 개발한다. SIFG는 데이터 슬라이스에서 정보의 흐름과정을 분명하게 보여주므로, 관심있는 변수 값의 변화 과정을 쉽게 탐색할 수 있고, 프로그램의 이해를 증진시킬 수 있다.

SCM은 측정 모델에서 정량적인 값으로의 사상을 명시한다. SCM 측정값은 복잡도 척도에 필요한 특성들을 만족해야한다. 따라서 본 연구에서는 SCM이 복잡도 척도에 필요한 특성들을 만족하는가를 평가한다. 또한 기존 복잡도 척도들과의 비교를 통해서 SCM의 특징을 살펴본다.

본 논문은 다음과 같이 구성된다. 제2장에서는 데이터 슬라이스에 대해여 고찰하고, 제3장에서는 데이터 슬라이스 상의 데이터 토큰들에서의 정보흐름을 통해 프로그램의 정보흐름을 모델링하기 위해 슬라이스 기반 정보흐름 그래프와 슬라이스 기반 복잡도 척도를 정의한다. 제4장에서는 제안된 척도가 복잡도 척도에 필요한 특성들을 만족하는가를 평가한다. 제5장에서는 기존 척도들과의 상관관계 분석을 통해서 제안된 척도를 고찰한다. 제6장에서 결론을 제시한다.

2. 데이터 슬라이스

슬라이싱은 Weiser[2]에 의해 소개된 프로그램 축소 방법으로 디버깅 도구와 프로그램 이해를 위한 보조 수단으로 제안되었고, 종래의 흐름 분석의 "사용"관계를 포착한다. 슬라이스는 다음과 같이 정의된다.

슬라이스 기준은 순서쌍 $C = < i, V >$로 표현된다. 여기서, i는 모듈 M에서 특정 문장의 번호이고, V는 M에서의 변수들의 집합이다. 슬라이스 기준 C에서 모듈 M의 모듈내 슬라이스(intramodule slice) S는 다음 특성을 갖는 임의의 실행 가능한 모듈이다 : (1) S는 M에서 임의의 문장들을 제거하여 획득될 수 있다. (2) 임의의 주어진 입력에 대해서, 슬라이스 S에서 모든 점들에서 실행은 문장 i에서 V의 변수들에 관하여 모듈 M에서 관찰된 것과 동일하다[12].

예를 들어서, (그림 1)의 모듈에서 슬라이싱 기준 $C = < 12, \{Sum\} >$와 $C = < 12, \{Prod\} >$로 획득되는 슬라이스는 (그림 2)와 (그림 3)과 같다.

[5]에서, 변수의 "사용"과 "사용됨" 관계를 설명하는 메트릭 슬라이스라 불리는 새로운 슬라이싱 형태가 정의되었다. 메트릭 슬라이스는 VRES(varable-referent executable statement)에 기초하여 모듈의 출력 변수들에 대해서 계산된다.

Bieman[6]은 데이터 토큰을 사용하기 위해서 메트릭 슬라이스를 수정한 데이터 슬라이스(data slices)에 근거한 응집도 척도를 제안했다. 데이터 슬라이스에서 데이터 토큰은 문장에서 정의된 변수와 상수정의를 의미한다. 데이터 슬라이스 역시 각 출력에 대해서 계산된다.

```
1   procedure SumAndProd(N1 : integer ;
2             var Sum1 : integer ;
3             var Prod1 : integer) ;
4   var I1 : integer ;
5   begin
6             Sum2 : = 01 ;
7             Prod2 : = 11 ;
8             for I2 : = 12 to N2 do begin
9                       Sum3 : = Sum4 + I3 ;
10                      Prod3 : = Prod4 + I4 ;
11            end
12  end
```

※ 여기서 아래첨자는 변수의 출현 순서를 의미한다

(그림 1) SumAndProd 프로시저

```
1   procedure SumAndProd(N1 : integer ;
2             var Sum1 : integer ;
4   var I1 : integer ;
5   begin
6             Sum2 : = 01 ;
8             for I2 : = 12 to N2 do begin
9                       Sum3 : = Sum4 + I3 ;
11            end
12  end
```

(그림 2) C=〈12, {Sum}〉으로 획득된 슬라이스

```
1   procedure SumAndProd(N1 : integer ;
3             var Prod1 : integer) ;
4   var I1 : integer ;
5   begin
7             Prod2 : = 11 ;
8             for I2 : = 12 to N2 do begin
10                      Prod3 : = Prod4 + I4 ;
11            end
12  end
```

(그림 3) C=〈12, {Prod}〉으로 획득된 슬라이스

예를 들어서, (그림 1)의 프로시저 SumAndProd에서 출력 Sum과 Prod에 대한 데이터 슬라이스는 (그림 2)와 (그림 3)에 나타나는 일련의 데이터 토큰들의 집합이다. 즉,

DataTokens(SumAndProd) = {N_1, N_2, 0_1, 1_1, 1_2, I_1, I_2, I_3, I_4, Sum_1, Sum_2, Sum_3, Sum_4, $Prod_1$, $Prod_2$, $Prod_3$, $Prod_4$}

$S(Sum) = \{N_1, Sum_1, 1_1, Sum_2, 0_1, I_2, 1_2, N_2, Sum_3, Sum_4, I_3\}$
$S(Prod) = \{N_1, Prod_1, 1_1, Prod_2, I_1, I_2, 1_2, N_2, Prod_3, Prod_4, I_4\}$

여기서 v_i는 프로시저에서 변수 v에 대한 i번째 토큰을 나타낸다.

그러나 위와 같은 데이터 슬라이스는 슬라이스에서의 정보 흐름이 무시된다. 따라서 데이터 슬라이스에서 데이터 토큰의 정보 흐름을 표현할 수 있는 데이터 슬라이스 표현이 필요하므로 본 연구에서는 데이터 슬라이스 상의 데이터 토큰들에서의 정보 흐름을 모델링하기 위한 슬라이스 기반 정보 흐름 그래프(SIFG)를 도입한다.

3. 슬라이스 기반 복잡도 척도의 정의

3.1 슬라이스 기반 정보 흐름 그래프(SIFG)

슬라이스 근거하여 프로그램 복잡도를 조사하기 위하여, 본 연구에서는 프로시저의 출력변수에 대한 슬라이스 내에서의 데이터 토큰들의 정보 흐름을 모델링하기 위한 슬라이스 기반 정보 흐름 그래프를 정의한다.

SIFG(S)로 표현되는 슬라이스 S에 슬라이스 기반 정보 흐름 그래프는 방향성 그래프, $SIFG(S) = <N, E>$이다. 여기서, N은 슬라이스 S에 포함된 데이터 토큰들의 집합이고, E는 데이터 토큰들 간의 정보 흐름을 나타내는 간선들의 집합이다

단일 슬라이스에서의 SIFG는 다음과 같은 단계를 거쳐서 생성된다.

Step 1 : 문장 내에서 사용되는 데이터 토큰들 간의 "정보흐름" 관계에 따라서 슬라이스 내의 모든 문장들을 정보 흐름 간선들의 집합으로 표현한다.

$d_i, d_j \in N$이 하나의 문장에서 나타나는 데이터 토큰이라 하자. 문장 내에서 d_i를 사용하는 연산의 결과가 d_j에 배정되면 ***d_j는 d_i를 직접 사용한다***고 하고, $d_i \rightarrow d_j$로 표현한다. 그렇지 않으면, 즉 비교연산 등이 사용되는 경우는 ***d_j는 d_i를 간접 사용한다***고 하고, $d_i \dashrightarrow d_j$로 표현한다. d_j가 d_i를 직접 또는 간접 사용하면 ***d_i에서 d_j로 직접 또는 간접적으로 정보가 흐른다***라고 한다.

$N(T_k)$는 슬라이스 내의 문장 T_k에 나타나는 모든 데이터 토큰들의 집합이라 하자. 이때, 정보 흐름 간선들의 집합은 다음과 같다.

$$E(T_k) = \{d_i \rightarrow d_j \text{ or } d_i \dashrightarrow d_j \mid d_i, d_j \in N(T_k), i \neq j\}$$

Step 2 : 동일 변수에 속하는 데이터 토큰들간의 "정보 흐름" 관계에 따라서 슬라이스 내의 임의의 변수들에 관한 정보 흐름 간선들의 집합을 구한다.

$N(v)(N(v) \in N)$를 슬라이스 내에서 사용되는 임의의 변수 v에 대한 데이터 토큰들의 집합이라 하자. $d_i, d_j \in N(v)$일 때, $d_i \in N(T_p)$이고 $d_j \in N(T_q)$이고, $p < q$이면 ***d_j는 d_i를 직접 사용한다***(즉, ***d_i에서 d_j로 직접적으로 정보가 흐른다***)라고 한다.

이때, 임의의 변수 v에 대한 정보 흐름 간선들의 집합은 다음과 같다.

$$E(v) = \{d_i \rightarrow d_j \mid d_i, d_j \in N(v), i \neq j\}$$

Step 3 : $E(S_i)$와 $E(v)$를 결합하여 SIFG(S)를 생성한다.

슬라이스 S에 대한 SIFG(S)에서 간선들의 집합 $E = E(S_i) \cup E(v)$이고, $d_i, d_j, d_k \in E$일때, 다음 조건들을 만족하면 SIFG(S)는 d_i에서 d_j로의 정보 흐름 간선(즉, $d_i \rightarrow d_j$ 또는 $d_i \dashrightarrow d_j$)을 포함한다.

① d_i에서 d_j로 정보가 흐른다.

② d_i에서 d_k로 그리고 d_k에서 d_j로 정보가 흐르는 노드 d_k가 존재하지 않는다.

Step 4 : 슬라이스 내에 반복문이 있는 경우, 반복문 내에서 정의되는 동일 변수에 관한 데이터 토큰들간의 정보 흐름을 추가한다.

$SIFG_{loop}(v)$를 반복문 구조 내에서 위치한 문장들에서 사용되는 변수 v에 관한 데이터 토큰들로부터 생성되는 SIFG(S)의 부분그래프(sub-graph)라 할 때, $SIFG_{loop}(v)$에서 종단노드 d_i에서 시작노드 d_j로의 직접사용 간선(즉, $d_i \rightarrow d_j$)을 추가한다.

예를 들어서, (그림 2)의 슬라이스에 대해서, 각 문장들에 대한 노드들의 집합은 다음과 같다.

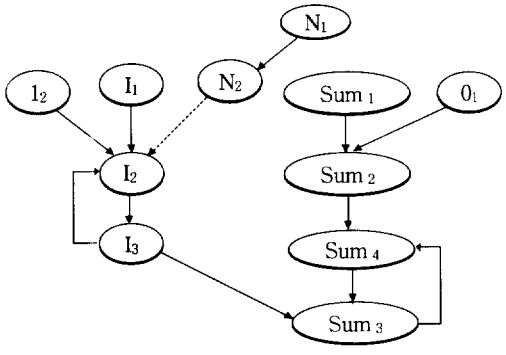
$N(T_1) = \{N_1\}$
$N(T_2) = \{Sum_1\}$
$N(T_4) = \{I_1\}$
$N(T_6) = \{Sum_2, 0_1\}$
$N(T_8) = \{I_2, 1_2, N_2\}$
$N(T_9) = \{Sum_3, Sum_4, I_3\}$

또한, 각 문장들을 위한 간선들의 집합은 다음과 같다.
$E(T_6) = \{0_1 \rightarrow Sum_2\}$
$E(T_8) = \{1_2 \rightarrow I_2, N_2 \dashrightarrow I_2\}$
$E(T_9) = \{Sum_4 \rightarrow Sum_3, I_3 \rightarrow Sum_3\}$

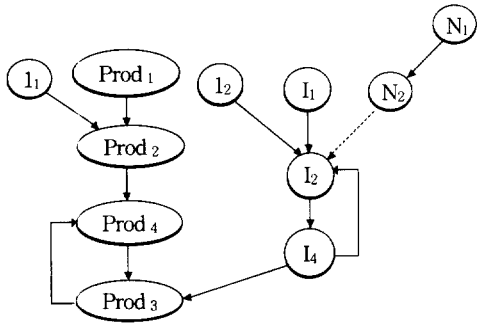
사용되는 변수들에 대한 간선들의 집합은 다음과 같다.
$E(N) = \{N_1 \rightarrow N_2\}$
$E(I) = \{I_1 \rightarrow I_2, I_2 \rightarrow I_3\}$
$E(Sum) = \{Sum_1 \rightarrow Sum_2, Sum_2 \rightarrow Sum_3, Sum_2 \rightarrow Sum_4\}$

위의 문장들에 대한 간선들의 집합과 변수들에 대한 간선들의 집합에서 생성된 SIFG에서, T_8과 T_{10}이 반복문 내

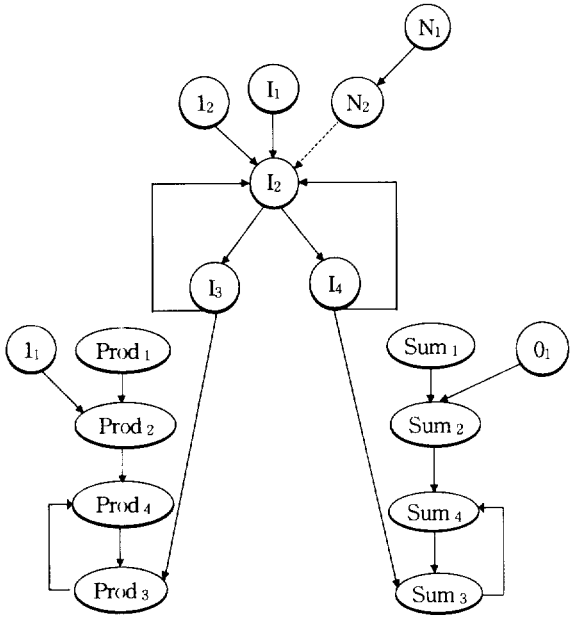
에 있는 문장들이므로 간선 $I_3 \rightarrow I_2$와 $Sum_3 \rightarrow Sum_4$를 추가하면 (그림 2)에 대한 SIFG(Sum)는 (그림 4)와 같다. 또한 (그림 3)에 대한 SIFG(Prod)는 (그림 5)와 같다.



(그림 4) SIFG(Sum)



(그림 5) SIFG(Prod)



(그림 6) SIFG(SumAndProd)

프로시저 P에서 생성되는 모든 슬라이스에 대한 SIFG의 결합을 통해서 프로시저 P에 대한 SIFG(P)는 다음과 같이 정의된다.

정의 1 : $N(S_i)$와 $E(S_i)$가 $SIFG(S_i)$의 노드와 간선들의 집합일 때, 프로시저 P에 대한 SIFG(P)는 다음과 같다.

$$SIFG(P) = < N_{S1} \cup N_{S2} \cup \cdots \cup N_{Sn} \ , \ E_{S1} \cup E_{S2} \cup \cdots \cup E_{Sn} >$$

(그림 4)와 (그림 5)를 결합한 프로시저 SumAndProd에 대한 SIFG(SumAndProd)는 (그림 6)와 같다.

SIFG의 노드로서 데이터 토큰들의 사용은 정보 흐름에 따른 미세 변화들이 SIFG에 반영되고 프로그램 복잡도에 영향을 미치게된다. 이러한 변화들에는 데이터 토큰의 추가와 삭제, 주어진 정황에서 데이터 값 흐름의 변화 등이 있다.

3.2 슬라이스 기반 복잡도 척도

프로그램의 이해와 디버깅에 슬라이스를 사용한다면[3], 프로그램의 이해는 슬라이스에서의 정보 흐름을 탐색하는 과정이라고 할 수 있다. 즉, 슬라이스의 모든 점들에서 실행의 과정을 예측하는 것으로 볼 수 있다. 따라서 특정 시점에서 실행의 예측을 위한 많은 선행 조건들이 요구된다면 그 시점에서의 프로그램의 이해는 어렵고 복잡하게된다.

따라서 본 연구에서는 SIFG의 임의의 노드의 내차수와 간접사용 간선의 개수에 의해 프로그램의 복잡도를 측정하는 슬라이스 기반 복잡도 메트릭(Slice-based Complexity Metric : SCM)을 다음과 같이 정의한다.

정의 2 : 슬라이스 S에 대한 SIFG(S)에서, N과 E는 SIFG (S)의 노드와 간선들의 집합이고, $E_{dirt}(E_{dirt} \subset E)$는 직접 사용 간선들의 집합이고, $E_{ind}(E_{ind} \subset E)$는 간접사용 간선들의 집합이고, $E = E_{dirt} \cup E_{dirt}$이고, d_i ($d_i \in N$)에 대한 $ind(d_i)$는 d_i로 향하는 직접사용 간선과 간접사용 간선들 개수, 즉 내차수(indegree)라 할 때, 슬라이스 기반 복잡도는 다음과 같다.

$$SCM = |E_{ind}| + \sum_{i=1}^{|N|} (ind(d_i), \ \text{if} \ ind(d_i) \geq 1)$$

(그림 4), (그림 5), (그림 6)에 대한 복잡도는 다음과 같다.

SCM(Sum) = 10 + 1 = 11
SCM(Prod) = 10 + 1 = 11
SCM(SumAndProd) = 17 + 1 = 18

(그림 4)에서, 간접 사용 간선은 $N_2 \rightarrow I_2$이므로 $|E_{ind}| = 1$이다. 또한, I_2, Sum_2, Sum_3, Sum_4가 내차수가 2이상인 노드이므로, $Ind(I_2) = 4$, $Ind(Sum_2) = 2$, $Ind(Sum_3) = 2$, $Ind(Sum_4) = 2$이다. 따라서 SCM(Sum) = 1+4+2+2+2 = 11이다.

4. 검 증

Fenton[13]에 의하면 소프트웨어 척도의 검증(validation)을 "척도가 주장된 속성의 적당한 숫자적 특징임을 보증하

는 과정"이다. 연구자들은 소프트웨어 척도의 원하는 다양한 특성(property)들을 제안하였다[14]. 그러한 프레임워크는 새로운 척도의 연구를 위한 지침이 된다. 또한 기존의 척도들은 이 특성들에 근거하여 정당화된다.

Briand[15]에 의해 제시된 특성은 광범위하게 논의되고 받아들여진다. 그것은 광범위한 측정 개념들, 개념 정의의 정형화, 측정 이론과의 일치성 등에서 이전의 연구들에서 나타나지 않은 분명한 장점을 갖는다[16]. 따라서 본 연구에서는 슬라이스 복잡도 척도를 검증하기 위해서 Briand의 복잡도 특성들 토대로 한다.

Briand[Brid 1996]에 의해 주어진 소프트웨어 시스템에 대한 정의들과 복잡도 척도의 가법성 관련된 정의들은 정의 3~정의 6과 같다.

정의 3 : 시스템 S는 순서쌍 S = <E, R>이다. 여기서 E는 시스템 S의 원소들의 집합이고, $R \subseteq E \times E$는 S의 원소들간의 관계를 표현하는 이항관계(binary relation)이다.

위의 정의 3은 다양한 소프트웨어 표현에 사상되기에 충분하다[16]. 시스템 S를 SIFG라 하면, E는 SIFG의 노드들의 집합이고 R은 노드들간의 정보 흐름의 집합이 된다. 그러므로 Briand의 다음 정의들은 SIFG에 직접 적용된다.

$\boldsymbol{P}$를 임의의 프로시저 P에 대한 SIFG라하고, $\boldsymbol{S}$를 $\boldsymbol{P}$에서의 임의의 슬라이스에 대한 SIFG라 하자. 그러면 다음 정의들과 특성들은 $\boldsymbol{P}$에 직접 적용된다.

정의 4 : $\boldsymbol{P}$ = <N, E>와 $\boldsymbol{S}$ = <N_S, E_S>가 있을 때, 다음을 만족하는 $\boldsymbol{S}$는 $\boldsymbol{P}$의 슬라이스이다.

$$\boldsymbol{S} \subseteq \boldsymbol{P} \Leftrightarrow E_S \subseteq E \text{ and } N_S \subseteq N$$

슬라이스 S에 대해서, 입출력 관계는 In($\boldsymbol{S}$)과 Out($\boldsymbol{S}$) 집합에 의해 표현된다.

정의 5 : $\boldsymbol{P}$ = <N, E>와 $\boldsymbol{P}$의 슬라이스 $\boldsymbol{S}$ = <N_S, E_S>가 주어지면,

$$In(\boldsymbol{S}) = \{d_1 \rightarrow d_2 \in E \text{ or } d_1 \dashrightarrow d_2 \in E \mid d_2 \in N_S \text{ and } d_1 \in N\text{-}N_S\}$$

$$Out(\boldsymbol{S}) = \{d_1 \rightarrow d_2 \in E \text{ or } d_1 \dashrightarrow d_2 \in E \mid d_1 \in N_S \text{ and } d_2 \in N\text{-}N_S\}$$

정의 6 : $\boldsymbol{P}$ = <N, E>와 $\boldsymbol{P}$의 슬라이스 $\boldsymbol{S}_i$ = <N_{Si}, E_{Si}>, $\boldsymbol{S}_j$ = <N_{Sj}, E_{Sj}>가 주어지면,

① $\boldsymbol{S}_i$와 $\boldsymbol{S}_j$의 합집합 $\boldsymbol{S}_i \cup \boldsymbol{S}_j$ = <$N_{Si} \cup N_{Sj}$, $E_{Si} \cup E_{Sj}$>
② $\boldsymbol{S}_i$와 $\boldsymbol{S}_j$의 교집합 $\boldsymbol{S}_i \cap \boldsymbol{S}_j$ = <$N_{Si} \cap N_{Sj}$, $E_{Si} \cap E_{Sj}$>
③ $E_{Si} \cap E_{Sj} = \emptyset$이면 $\boldsymbol{S}_i$와 $\boldsymbol{S}_j$는 서로소이다.
④ $E_{Si} \cap E_{Sj} = \emptyset$이고 $In(\boldsymbol{S}_i) \cap Out(\boldsymbol{S}_j) = \emptyset$ 이고 $In(\boldsymbol{S}_j) \cap Out(\boldsymbol{S}_i) = \emptyset$이면 $\boldsymbol{S}_i$와 $\boldsymbol{S}_j$는 연결되지 않는다.

Briand[15]가 제시한 다음 특성들은 슬라이스 기반 복잡도 척도가 만족해야할 충분조건들은 아니지만 필요 조건들이다. 여기서, $\boldsymbol{P}$ = < N, E >와 $\boldsymbol{P}$의 슬라이스 $\boldsymbol{S}$ = < N_S, E_S > 가 주어진다고 하자.

특성 1 : (Nonnegativity)
$SCM(\boldsymbol{P}) \geq 0$

특성 2 : (Null value)
$E = \emptyset \Rightarrow SCM(\boldsymbol{P}) = 0$

특성 3 : (Symmetry)
($\boldsymbol{P}$ = <N, E> and $\boldsymbol{P}^{-1}$ = <N, E^{-1}>)
$\Rightarrow SCM(\boldsymbol{P}) = SCM(\boldsymbol{P}^{-1})$

특성 4 : (Module monotonicity)
($\boldsymbol{P}$ = <N, E> and $\boldsymbol{S}_1$ = <N_{S1}, E_{S1}> and $\boldsymbol{S}_2$ = <N_{S22}, E_{S2}> and $\boldsymbol{S}_1 \cup \boldsymbol{S}_2 \subseteq \boldsymbol{P}$ and $E_{S1} \cap E_{S2} = \emptyset$)
$\Rightarrow SCM(\boldsymbol{P}) \geq SCM(\boldsymbol{S}_1) + SCM(\boldsymbol{S}_2)$

특성 5 : (Disjoint module additivity)
($\boldsymbol{P}$ = < N, E > and $\boldsymbol{P}$ = $S_1 \cup S_2$ and $\boldsymbol{S}_1 \cap \boldsymbol{S}_1 = \emptyset$)
$\Rightarrow SCM(\boldsymbol{P}) = SCM(\boldsymbol{S}_1) + SCM(\boldsymbol{S}_2)$

특성 6 : (Non-decreasing monotonicity)
($\boldsymbol{S}_1$ = < N_{S1}, E_{S1}> and $\boldsymbol{S}_2$ = < N_{S2}, E_{S2}> and $E_{S1} \subseteq E_{S2}$)
$\Rightarrow SCM(\boldsymbol{S}_1) \leq SCM(\boldsymbol{S}_2)$

임의의 프로시저 P에 대한 SIFG인 $\boldsymbol{P}$ = <N, E>에서, 제안된 *SCM*이 위의 특성들을 만족하는가를 살펴보자.

SCM($\boldsymbol{P}$)는 양수값을 가지므로 특성 1을 만족한다. E = ∅이면 *SCM*($\boldsymbol{P}$) = 0이므로 특성 2를 만족한다. **P**상에서 외차 노드의 합과 내차 노드의 합이 같다고 볼 수 없지만, *SCM*은 특정 시점에서 실행의 예측을 위한 많은 선행 조건들에 근거하여 복잡도를 측정하는데, $\boldsymbol{P}$에서의 선행조건들은 임의의 노드의 내차수(indegree)가 되지만 $\boldsymbol{P}^{-1}$에서의 선행조건들은 임의의 노드의 외차수(outdegree)가 되므로 이 특성 2를 만족한다고 볼 수 있다. 두 개의 슬라이스 $\boldsymbol{S}_1$ = <N_{S1}, E_{S1}>과 $\boldsymbol{S}_2$ = < N_{S2}, E_{S2}>가 연결되고 공통 간선이 없는 경우, 두 개의 슬라이스는 하나 이상의 노드를 공유한다. 이때, 공유 노드의 내차수가 1이상이면 연결된 슬라이스의 내차수의 합은 증가한다. 공유 노드의 내차수가 2이상이면 연결된 슬라이스의 내차수의 합은 변하지 않는다. 따라서 SCM은 특성 4를 만족한다. $\boldsymbol{S}_1 \cup \boldsymbol{S}_2 = \boldsymbol{P}$이고 $\boldsymbol{S}_1 \cap \boldsymbol{S}_2 = \emptyset$의 조건을 만족하는 슬라이스들은 연결이 이루어지지 않으므로 슬라이스에서의 내차수의 합과 간접사용 간선의 개수가 변하지 않는다.

따라서 특성 5를 만족한다. SIFG에 임의의 간선의 추가는 하나의 노드의 내차 노드를 증가를 초래하므로 특성 6을 만족한다.

따라서, 제안된 SCM은 복잡도 척도가 만족해야하는 필요조건들을 만족한다고 할 수 있다.

5. 실험 및 결과

프로그램의 복잡도를 측정하는 대표적인 척도들에는 Lines Of Code(LOC), Maccabe의 순환수(Cyclomatic number), Halstead의 소프트웨어 과학(Software Science)등을 들 수 있다[17].

LOC는 프로그램의 크기를 측정하기 위해서 프로그램 코드의 라인 수를 세는 방법으로, 일관성이 없다는 단점이 있지만 간단하여 사용자가 직관적으로 알 수 있다는 장점이 있다[18].

Halstead는 소프트웨어 과학이라 부르는 프로그램에 내재된 논리의 규모를 측정하는 척도들인 Length, Volume, Difficulty, Effort를 제안했다. IBM의 연구자들은 LOC, Length, Volume들 사이의 강한 관련성을 발견했고, 길이와 볼륨은 보편적인 메트릭으로 언어에 종속되지만 LOC는 종속되지 않음을 주장했고[19], 이 메트릭은 적용과 검증결과 정확성이 뛰어나고, 실험적으로도 매우 광범위하게 그 타당성을 인정받고 있으며, 이 메트릭들은 프로그램을 작성하는데 요구되는 시간과 정신적인 노력을 예측하는데 관련이 있다[20].

McCabe가 제안한 순환수는 프로그램의 논리적인 복잡도를 표현하는 제어 구조에 기반을 둔 대표적인 척도이다. 이 척도는 물리적 크기에 의한것이 아닌 프로그램의 제어 흐름을 기초로 하여 복잡도를 측정한다. 이 척도는 많은 유용성에도 불구하고 제어구조에만 의존하여 복잡도를 측정함으로써 다른 요인의 측정에는 문제점을 지니고 있다. 프로그램이 전적으로 순차적인 프로그램으로 이루어졌다면 크기에 관계없이 복잡도는 동일하다는 문제는 있지만, 이 척도는 경험적 연구를 통하여 그 효율성이 증명되었다[21].

본 연구에서는 [5, 6, 8, 22-25]에서 추출한 12개의 프로시저에 대한 SCM 측정값을 구하였다.

<표 1>은 12개의 프로시저에 대한 측정값을 보여주며, <표 2>는 SPSS를 이용한 기술통계량(descriptive statistics)을 보여준다.

일반적으로 통계적 비교 및 추론은 정규모집단이라는 가정하에서 유도되고 분석되는데, 왜도(skewness)와 첨도(kurtosis)는 정규모집단의 가정을 검토하는 하나의 방법으로 이용되고 있다. 왜도는 분포의 치우친 정도를 나타내는 측도이고, 첨도는 대칭인 분포에서 꼬리가 두꺼운 정도를 나타낸다. 왜도와 첨도가 0값과 충분히 다르면 이는 모집단이 정규분포가 아님을 시사한다[26].

〈표 1〉 SCM 측정값

Procedures	SCM	LOC	CYC	VOL	DIF	EFF
lowterm	23	16	3	246	20	5026
convert	34	18	2	326	20	6641
sigma	44	20	4	392	24	9418
lsqe	58	18	2	637	25	15678
rotat	18	13	1	288	13	3740
set_code	28	15	5	303	16	4913
Decode	12	12	2	143	11	1533
Decode 2	14	14	2	199	12	2308
Sum 1	31	13	3	217	18	3868
Sum 2	58	18	5	342	24	8335
FindLargest	41	26	5	401	33	13345
Sum_Max_Avg	28	16	3	255	21	5355

<표 2>에서 각 척도에 대한 왜도와 첨도가 모두 0에 가까우므로 <표 1>의 모집단은 정규분포에 따른다고 할 수 있다.

〈표 2〉 기술통계량(descriptive statistics)

	Min	Max	Mean	Skewness	Kurtosis
SCM	12	58	32.417	0.504	-0.624
LOC	12	26	16.583	1.289	2.312
CYC	1	5	3.083	0.323	-1.177
VOL	143	637	312.417	1.450	3.235
DIF	11	33	19.750	0.485	0.294
EFF	1533	15678	6680.00	1.041	0.404

<표 3>은 LOC와 cyclomatic number와 software science 간의 상관관계를 보여준다. <표 3>에서 LOC와 software science간의 상관관계는 매우 높게 나타났고, 유의 수준 1% 내에서 아주 유의한 것으로 나타났지만, LOC와 cyclomatic number, cyclomatic number와 software science간에는 상관관계가 낮게 나타났고, 5%내에서 유의하지 않았다.

〈표 3〉 기존 척도들간의 상관관계

	LOC	CYC	VOL	DIF	EFF
LOC	1.000	.572	.594*	.929**	.805**
sig.	.	.052	.042	.000	.002
CYC		1.000	.150	.586*	.340
sig.		.	.642	.045	.280
VOL			1.000	.673*	.936**
sig.			.	.016	.000
DIF				1.000	.873**
sig.				.	.000
EFF					1.000
sig.					.

〈표 4〉 SCM과 기존 척도와의 상관관계

	LOC	CYC	VOL	DIF	EFF
SCM	0.625*	0.477	0.818**	0.780**	0.850**
sig.	0.030	0.117	0.001	0.003	0.000

<표 4>는 SCM과 기존 척도들과의 상관관계를 보여준다. <표 4>에서 SCM은 LOC, Volume, Difficulty, Effort와 상관관계가 높게 나타났고 유의수준 1%내에서 유의한 것

으로 나타났고, Effort와의 상관관계가 가장높게 나타났다. 반면에, cyclomatic number와의 상관관계는 낮게 나타났다. 분석결과 SCM은 프로그램에 나타나는 연산자와 피연산자의 개수보다는 연산자와 피연산자의 총 발생 빈도수와 더 높은 상관관계를 보여주었다.

위의 결과에서, SCM과 Software Science간의 상관관계는 높게 나타났다는 것은 SCM이 프로그램에서 사용되어지는 변수들간의 정보의 흐름에 근거하여 복잡도를 측정하지만 프로그램의 물리적 크기를 반영하는 측정이 이루어진다는 것을 보여준다.

따라서, Cyclomatic number가 프로그램에서 제어 정보의 흐름만을 측정하므로 프로그램 크기에 관계없이 순차적으로 이루어진 프로그램에 대한 cyclomatic number 측정값이 동일한데 비하여, SCM은 프로그램 내에서의 제어와 데이터 흐름뿐만 아니라 프로그램의 물리적 크기를 반영하므로 전적으로 순차적으로 이루어진 프로그램도 크기에 따라서 상이한 SCM 측정값을 갖게된다. 또한 프로그램 내에서 복잡한 연산들이 많을수록, 또는 연산의 회수가 많을 수로 높은 SCM값을 갖게된다.

6. 결 론

본 연구에서는 측정 이론으로부터의 원칙들을 이용하여 슬라이스에서의 정보 흐름에 기초하여 프로그램의 복잡도를 측정하는 방법을 제안하였다. 먼저, 슬라이스에서의 정보 흐름을 모델링하기 위해 슬라이스 기반 정보 흐름 그래프(*SIFG : Slice -based Information Flow Graph*)를 개발하였다. 다음으로, SIFG에서의 정보 흐름의 복잡도를 측정하기 위해 슬라이스 기반 복잡도 척도(*SCM : Slice-based Complexity Measure*)를 정의하였다.

본 연구에서는 SCM이 Briand의 복잡도 특성들을 만족함을 보여주었다. 기존 척도들과의 비교를 통해서, SCM은 프로그램 내에서의 제어와 데이터 흐름뿐만 아니라 프로그램의 물리적 크기를 반영하는 측정이 이루어진다는 것을 확인할 수 있었다.

현재까지의 작업은 단일 프로시저에서의 슬라이스(inttra-procedure slice)에 기반한 복잡도 측정에 집중하였지만, 프로시저간 슬라이스(interprocedure slice)뿐만 아니라 객체지향 패러다임에서의 슬라이스에 근거한 복잡도 측정에 대한 연구가 필요하다.

참 고 문 헌

[1] Karl J. Ottensteion, Linda M. Ottensteion, "The program dependence graph in a software development environment," *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment, ACM SIGPLAN Notices*, Vol.19, No.5, pp.177-184, 1984.

[2] M. Weiser, "Program slicing," *In Prodeedings of the 5th International conference on Software Engineering*, pp.439-449, 1981.

[3] M. Weiser, "Programmers use slices when debuggung," *Communication of the ACM*, Vol.25, No.7, pp.446-452, 1982.

[4] M. Weiser, "Program slicing," *IEEE Transaction Software Engineering*, Vol.10, No.4, pp.352-357, 1984.

[5] Linda M. Ott, Jeffrey J. Thuss, "Slice based metrics for estimating cohesion," *Proc. IEEE-CS International Software Metrics Symposium*, Baltimore, pp.71-81, May 1993.

[6] J. M. Bieman, L. M. Otto, "Measuring functional cohesion," *IEEE Transaction Software Engineering*, Vol.20, No.2, pp.111-124, 1994.

[7] James M. Bieman, B. K. Kang, "Cohesion and reuse in an object-oriented system," *Proceeding ACM Symposium on Software Reusability(SSR'95)*, pp.259-262, April, 1995.

[8] J. M. Bieman, B. K. Kang, "Measuring design-level cohesion," *IEEE Transaction Software Engineering*, Vol.24, No.2, pp.111-124, 1998.

[9] B. A. Kitchenham, N. Fenton, S. Lawrence, "Towards a framework for software measurement validation," *IEEE Transaction Software Engineering*, Vol.21, No.12, pp.929-944, December, 1995.

[10] A. L. Baker, J. M. Bieman, N. E. Fenton, A. C. Melton, R. W. Whitty, "A philosophy for software measurement," *Journal of Systems and Software*, Vol.12, No.3 pp. 277-281, July, 1990.

[11] Rachel Harrison, Steve J. Counsell, "An Evaluation of MOOD set of object-oriented software metrics," *IEEE Transaction Software Engineering*, Vol.24, No.6, pp.491- 496, June, 1989.

[12] Lnda M. Otto, "Using Slice Profiles and Metrics during Software Maintenance," *Proc. 10th Annual Software Reliability Symposium*, 1992.

[13] N. Fenton, *Software Metrics-A Rigorous Approach*, Chapman and Hall, London, 1991.

[14] E.Weyuker, "Evaluating software complexity measure," *IEEE Transaction Software Engineering*, Vol.14, No.9, pp.1357-1356, Sept. 1988.

[15] L. C. Briand, S. Morasca, V. R. Basili, "Property-Based Software Engineering Measurement," *IEEE Transaction Software Engineering*, Vol.22, No.1, pp.68-86, 1996.

[16] G. Poels, B. Dedene, "Comments on Property-Based Software Engineering Measurement," *IEEE Transaction Software Engineering*, Vol.23, No.3, pp.190-195, 1997.

[17] Horst Zuse, *Software Complexity-Measures and Methods*, pp.25-37, Walter de Gruyter, New York, 1991.

[18] 이철희 등, *소프트웨어 공학*, 홍릉과학 출판사, 서울, 1989.

[19] Arthur, L. J., *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons, New York, 1985.

[20] Shen,V. Y. Conte, S. D. Dunsmore, H. E,. "Software Science Revisited : A Critical Analysis of the Theory and its Empirical Support," *IEEE Transactions On Software Engineering*, Vol.9, pp.308-320, 1976.
[21] 최완규, "러프논리와 퍼지척도에 기반한 재사용 결정지원 모델", *박사학위 논문*, 조선대학교, 2000.
[22] Linda M. Otto, Jeffrey J. Thuss. "The Relationship between Slices and Module Cohesion," *Proc. 11th ICSE*, 1989.
[23] 송만석, *Structed PASCAL 프로그래밍*, 정익사, 서울, 1994.
[24] Kyle Loudon, *Algorithms with C*, O'REILLY, 1999.
[25] 조순복, 박홍준, *터보 C 실무 프로그래밍 기법*, 기한재, 1996.
[26] 김우철 외, 현대통계학, 영지문화사, 1994.

문 유 미

e-mail : mym60@hananet.net
1983년 조선대학교 전자계산학과 졸업(학사)
1987년 조선대학교 전자계산학과 졸업 (이학석사)
1998년~현재 조선대학교 전자계산학과 박사과정 조선대학교 공과대학 컴퓨터공학부 전자계산학과 출강, 송원대학교 금융정보학과 겸임교수
관심분야 : 소프트웨어공학, 소프트웨어 메트릭스, 프로그램복잡도, 러프및퍼지이론, 전자상거래

최 완 규

e-mail : wkchoi@kwangju.ac.kr
1988년 서울대학교 종교학과 졸업(학사)
1992년~1993년 (주)공성통신 전산실
1993년~1995년 한양시스템 전산실
1997년 조선대학교 전자계산학과(이학석사)
2000년 조선대학교 전자계산학과 (이학박사)
2000년~현재 광주대학교 컴퓨터전자통신공학부 전임강사
관심분야 : 소프트웨어 공학, 프로그래밍 언어, 객체지향 시스템, 러프집합

이 성 주

e-mail : sjlee@mail.chosun.ac.kr
1970년 한남대학교 물리학과(학사)
1992년 광운대학교 전자계산학과 (이학석사)
1998년 대구 가톨릭대학교(이학박사)
1988년~1990년 조선대학교 전자계산소 소장
1995년~1997년 조선대학교 정보과학대학장
1981년~현재 조선대학교 컴퓨터공학부 교수
관심분야 : 소프트웨어 공학, 프로그래밍 언어, 객체지향 시스템, 러프집합