패트리넷트를 이용한 프로그램의 제어구분적 복잡도

이 종 근† 송 유 진††

요 약

본 연구는 패트리네트를 이용한 소프트웨어의 이해용이성에 기인한 효율적인 복잡도방법을 제시한다. 일반적으로 제어의 흐름에 있어서는 순차성과 조건문 그리고 반복문에 의한 구조적인 구분이 가능하므로 본 논문에서는 그러한 구조적인 부문을 패트리네트로 표현하여 각각의 이해용이성에 기인한 복잡도를 산출하여 종합적인 총복잡도를 산출하였다. 또한 기존의 다른 복잡도와의 상관관계를 분석함으로 새로운 복잡도의 신뢰성도 증명하였다.

Software Complexity Measure Based on Program Control Structure Using Petri Nets

Jong Kun Lee† and You Jin Song††

ABSTRACT

In this paper, we present a syntactic software complexity measure based on program control structure using Petri Nets. Since control structure in program may be segregated by three structures such as sequence, condition and iteration structures, we are proposed a structured complexity measure based on program control structure after represented by Petri Nets. Finally, we compare our result with other measures of program complexity.

1. 서 론

복잡도란 데이타 구조의 형태, 중첩된 정도, 조건분기(condition branches)의 수(number), 인터페이스의 수, 그리고 시스템적인 특징(characteristics)등과 같은 요소로서 나타내지는 소프트웨어 시스템이나 시스템적 요소들의 복잡한 정도를 의미한다. 따라서 소프트웨어의 품질을 정의하고 평가하는데 있어서 복잡도의 측정이 매우 유용하게 활용되어 오고 있다. 이러한 소프트웨어의 복잡도를 측정하기 위하여 Magel[2]은 프로그램의 크기, 제어의 흐름, 데이타 흐름(데이타 구조 포함)등을 지적하였고 이러한 요소들을 활용한 많은 복잡도[1, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 22, 23]가 활발하게 연구되어 왔다. 이 중에서 제어의 흐름과 데이타의 흐름에 의한 복잡도는 소프트웨어의 복잡도를 측정하는 요소로서 상호 보완적인 역할을 하며 그래프상에서 쉽게 측정될 수 있고 프로그램의 신뢰성, 생산성, 유지보수성등의 측정과 평가에 효과적으로 적용될 수 있기 때문에 최근의 소프트웨어 개발 분야에서 많이 이용되고 있다. 문제는 프로그램 내부에서의 자료의 흐름과 제어의 흐름과의 관계를 얼마나 잘 표현하더 또한 정량적인 분석을 위한 수식화 하는데 그 중요성이 있다고 말할 수 있다.

본 연구에서는 프로그램을 알기 쉽게 분석하고, 좀 더 크고 복잡한 프로그램의 복잡도를 구하고자 할 때의 용이성을 위하여 시스템의 정보와 제어흐름을 모델링하고 분석하는 강력한 도구로써 사용되어져 온 패트리네트를 이용하고자 한다. 패트리네트는 시스템의 성능분석에 있어서 흐름(transition)을 통한 제어의 흐름과 자료의 흐름을 순차적으로 또한 서로 연관되게 표현할 수 있으며 그 본래의 특성 즉, 정량화하여 분석할 수 있는 특성을 프로그램의 복잡도에 적용하여 특히 프로그램을 모델링함에 있어 순차문, 반복문, 그리고 선택문 등의 3가지 제어구조 형태로

† 정 회 원 : 창원대학교 전자계산학과 교수
†† 정 회 원 : 창원대학교 정보통신시스템연구실 연구원
논문접수 : 1995년 1월 12일, 심사완료 : 1995년 4월 28일

구분하여 각각의 이해용이성적 측정치를 계산하고 복잡도 수치를 계산하여 종합적인 제어구분적 프로그램의 복잡도를 산출한다.

본 논문의 구성을 보면, 2장에서 패트리네트 모델에 관한 정의와, 3장에서는 본 연구의 기초가 되는 구조적 프로그램의 모델화 규칙에 관해 살펴보며, 4장에서는 새로운 복잡도를 제시함과 동시에 예제프로그램을 들어 이해도를 높였다. 그리고 5장에서는 제시된 복잡도와 기존 복잡도간의 비교분석을 통한 척도의 타당성을 검증하였다.

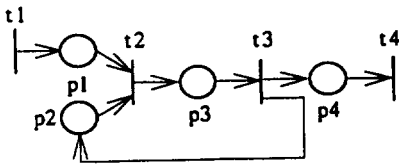
2. 패트리네트 모델

패트리네트이론은 1962년 C.A.Petri[4]에 의하여 제시되어 많은 연구자들에 의해 다양한 목적을 가지고 연구되어 왔으며,그 후 본래의 패트리네트에 폭넓은 수정과 이론적 확장을 통하여 TPN(Time Petri nets), CPN(Coloured Petri nets), HPN(Hierarchical Petri nets)등의 유용한 패트리네트의 이론을 구축하여 왔다. 패트리네트는 또한 일반적인 시스템의 모델링과 분석을 포함하며 통신분야에서는 프로토콜의 모델링등에 널리 이용되어 오고 있다.

패트리네트 PN은 다음과 같이 Place의 유한집합 P, Transition의 유한집합 T, 정방향 영향함수(forward incidence function) F, 역방향 영향함수(backward incidence function) B로 구성된다 : N = {P, T, F, B} 여기서, P = {p1, p2, ……, pm}, $P \neq \emptyset$, T = {t1, t2, ……, tn}, $T \neq \emptyset$, $P \cap T = \emptyset$, $F : P \times T \rightarrow N$, (단, N은 음수를 제외한 정수집합) 그리고 $B : P \times T \rightarrow N$ 이다. 패트리네트는 시스템의 동시적 또는 병행적 프로세스들을 표현하는 모델링 도구로 개발되었는데, 프로세스는 조건(condition), 사건(event) 및 그들간의 관계를 서술하는 규칙(rule)으로 구성된다고 가정할 수 있으며 패트리네트 모델의 도식적 표현인 패트리네트 그래프에서 이들 조건과 사건을 각각 place(원으로 표시)와 transition(선분으로 표시)으로 나타낸다. 조건을 만족하는것은 place에 토큰을 위치시킴으로써 표현하며 place와 transition, transition과 place의 연결은 방향성 화살표로써 표현한다. transition(즉, 사건)은 자신에게로 입력되는 모든 place가 토큰을 보유하고 있어야(즉, 모든 조건을 만족하여야) 점화(fire)될 수 있다. transition이 점화되면 자신의 각 입력 place로부터 토큰을 하나씩 제거하고 각 출력 place에 토큰을 하나씩 첨가한다.

〈예제 1〉 다음은 4개의 Place와 4개의 transition을 갖는 패트리네트를 나타낸 것이다.

PN=(P, T, F, B)
여기서, P={p1, p2, p3,p4},
T={t1, t2, t3, t4},
F(t1)=0, F(t2)={p1, p2}
F(t3)={p3}, F(t4)={p4, p4}
B(t1)={p1}, B(t2)={p3}
B(t3)={p2,p4,p4}, B(t4)=0



(그림 1) 예제 패트리네트
(Fig. 1) Example Petri net

3. 구조적 프로그램의 모델화 규칙

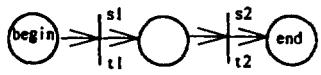
패트리네트는 시스템에서 정보와 제어흐름을 모델링하고 분석하는 그래프형 도구로써 사용되어 왔다. 이것은 또한 컴퓨터 시스템을 묘사하고 분석하는데 중요하게 사용되어져 왔고 최근에는 패트리네트를 이용하여 프로그램의 구조를 분석하는 데에도 활용되어지고 있다[20].

일반적으로 구조적 프로그램은 단일입구와 단일출구의 요소를 가진 프로그램으로서 순차문, 제어문 그리고 반복문〈표 1〉과 같이 세가지 구조로 나누어질 수 있다. 이러한 단일입구-단일출구 요소들은 프로그램을 구성하는데 성공적인 능력을 가지고 있다고 할 수 있다. 따라서 제어의 흐름을 가장 효과적으로 표현할 수 있는 패트리네트에서 Transition은 오퍼레이션의 집합을 표현하고 Place는 조건이나 자료의 집합을 표현함으로 구조적 프로그램에서 기 제시된 세가지 구조를 활용하여 패트리네트로 모델링 할 수 있다.

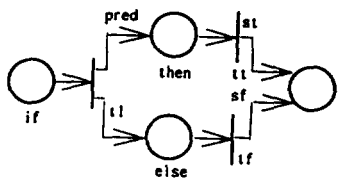
본 연구에서는 (그림 2), (그림 3), (그림 4)로 제어의 흐름을 규격화하여 구조적 프로그램을 모델링화 하는데 필요한 기본 구조를 패트리네트로 설정시켜 복잡도의 산출을 용이하게 하고자 한다 [22, 23].

4. 제어구조별 복잡도 측정

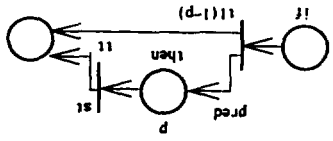
구조적인 프로그램은 앞장에서 설명한 바대로 기본적인 세가지의 구조를 활용하여 구성되므로 획일적인 복잡도의 수치 산정이 무의미하다고 본다. 따라서 본 장에서는 프로그램을 패트리네트에 의하여 구조별로 구성되어지면 각각의 구조적인 특성에 따른 복잡도의 수치를 산출하여 전체적인 프로그램의 복잡도를 제시한다.



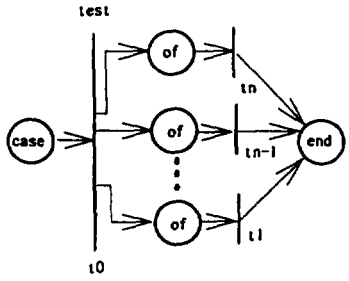
(그림 2) 순차문
(Fig. 2) sequential statement



(a) if-then-else



(b) if-then

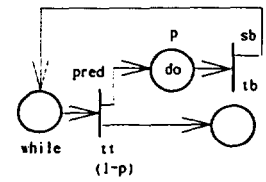


(c) case of

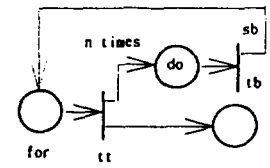
(그림 3) 제어문
(Fig. 3) Choice Statement

〈표 1〉 구조적 프로그램의 구성요소
(Table 1) Structure elements of structured Program

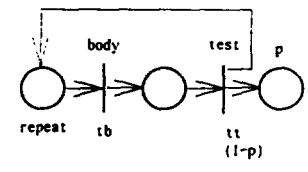
구 분	특 성
Basic actions (순차문)	제어가 하나의 요소에서부터 다른 요소로 순차적으로 흐른다. (Concatenation)
Choice (제어문)	제어가 단일점으로 부터 둘 또는 그 이상의 경로들로 나누어지는 구조로 모든 경로들은 출구의 단일점으로 합병된다. (If-then-else)
Iteration (반복문)	제어가 반복적으로 하나 또는 그 이상의 요소를 통하여 흐르며 단일점의 출구로 나간다. (Looping)



(a) while



(b) for



(c) repeat

(그림 4) 반복문
(Fig. 4) Iteration statement

4.1 순차문 복잡도(B : Basic Actions)

제어구조가 없는 순차문장을 Halstead[21]의 개념에서 복잡도의 값을 계산한다. Halstead는 프로그램의 전체 길이, 알고리즘의 예상되는 최소한의 크기, 실제의 크기, 프로그램의 수준, 언어의 수준 그리고 개발노력과 시간, 소프트웨어의 예상오류수 등과 같은 특성을 계산하기 위한 수식을 개발하는데 기본적인 측정치들을 사용하고 있다. 그러므로 제어구조가 없는 순차문의 경우는 Halstead의 수식을 적용한 복잡도로 계산할 수 있다.

Halstead[21]는 프로그램에 영향을 미치는 요인을 다음 4가지로 보았다.

n1 : 프로그램의 유일한 연산자 수
n2 : 프로그램의 유일한 피연산자 수
N1 : 프로그램의 총 연산자 수
N2 : 프로그램의 총 피연산자 수

이들을 기준으로 구현된 프로그램의 크기(V), 프로그래밍에 필요한 노력(E)을 통하여 다음의 공식[21, 26]들을 제시한다.

Vocabulary : $n = n1 + n2$
Length : $N = N1 + N2$
Volume : $V = N\log2(n)$
Program level : $L = 2/n1 * n2/N2$
순차문의 복잡도 : $B = V/L$
(Program effort)

4.2 분기문 복잡도(C : Choice)

Ramamurthy[7]는 프로그램의 크기와 제어흐름을 동시에 측정하기 위해서 Cabea의 V(a)와 Halstead의 기본 요소의 값을 결합하여 제어구문에 가중치를 부과하였으며 이를 공식화 하면 다음과 같다[3, 7].

$Nw1 = \sum_{X \in T} [1+d(x) * l(x)]$: weightes total of all operator
$Nw2 = \sum_{X \in R} [1+d(x) * l(x)]$: weightes total of all operand
$Nw = Nw1 + Nw2$: weightes length
$Vw = Nw * \log(n1+n2)$: weightes volume
$C = Vw * [(n1 * Nw2)/(2 * n2)]$

여기서 T는 모듈내의 오퍼레이터들의 집합이고 R은 모듈내의 오퍼런드들의 집합이며 d(x)는 제어구조가 있는지 없는지를 나타낸다. 오퍼레이터나 오퍼런드가 제어구조안에 있으면 1 아니면 0으로 계산한다. l(x)는 내포 단계로서 제어구조에 가중치를 부과한다.

〈예제 2〉

```
PROGRAM 1(input, output)
VAR
  a, b, c, d, m : integer ;
BEGIN
  readln(a, b, c, d) ;
  a : = a+a ;
  b : = b+b ;
  c : = c*d ;
  m : = a+b-c ;
  writeln(m)
END.
```

(a) 프로그램 1

```
PROGRAM 2(input, output)
   VAR
     a, b, c, d, m : integer ;
   BEGIN
     readln(a, b, c, d) ;
     IF a > b THEN
            IF b > c THEN
                   m : = a+b
                   ELSE
                   m : = b+c
            ELSE
            m : = b+c*d ;
            writeln(m)
   END.
```

(b) 프로그램 2

(그림 5) 예제 프로그램
(Fig. 5) Example programs

위의 〈예제 2〉에서 (a) 프로그램 1에는 제어구조가 전혀 나타나지 않으므로 가중치가 없다. 그러므로 원래 Halstead's의 노력값 그대로가 적용된다. (b) 프로그램 2의 경우는 d(x)가 1이고 l(x)는 모듈이 중첩된 정도를 나타낸다.

4.3 반복문 복잡도(I : Iteration)

프로그래머가 주어진 프로그램을 이해할 때에

는 문자 단위로 이해하지 않는다는 사실은 다른 많은 연구에서 확인이 되었다. 대신에 프로그램의 공통적인 성질을 갖는 그룹 단위로 이해하는데 이러한 공통된 그룹을 청크(chunk)라고 하며 [24] 프로그래밍은 이러한 청크(chunk)들간의 관계를 맺도록 흐름을 만드는 작업이라고 볼 수 있다. 따라서 프로그래머가 프로그램을 이해하고자 할 때, 특히 친숙하지 않은 프로그램을 이해하고자 할 때는 우선적으로 프로그램내에 존재하는 개별적인 청크(chunk)를 이해하려고 한다는 사실을 쉽게 가정할 수 있다. 즉 명령군을 이해하기 위해서는 이 명령군에 영향을 미치는 다른 청크(chunk)를 참조하면서 여러번 반복적으로 검토를 해야 완전히 이해할 수 있는 것이다.

Woodfield[21]는 모듈을 처음 검토할 때와 두번째 검토할 때 모듈을 이해하는데 드는 노력이 다르다는 것을 증명하였고 이것을 검토상수 R이라고 하며, R은 2/3이라는 값을 실험에 의하여 확인하였다.

따라서 반복문에서의 복잡도 계산은 이러한 검토상수를 이용하여 산출할 수가 있다:

$$I = \sum_{x \in v(i)} [C * R^{Xrev}].$$

여기서, v(i) : 반복을 구성하는 트랜지션의 집합,
Xrev : 트랜지션이 반복된 횟수

4.4 제어별 복잡도(CS : Structured Complexity)

본 연구에서 제시하는 제어별 복잡도는 앞절에서 설명한 세가지의 제어별 복잡도를 종합하여 나타낸다. 즉 제어별 복잡도란 B + C + I를 합한 수를 의미한다. 다시 말하면 모든 프로그램은 위에서 정의된 세가지의 구조가 반복적으로 구성되어 있으므로 각각의 특성에 맞는 가중치적인 복잡도의 값을 부여함으로 종합적인 제어적 복잡도 수치를 제시할 수가 있다고 하겠다.

(정의 1) 구조적 프로그램의 제어적 복잡도 수치 (CS)는 다음과 같이 계산되어 진다:

CS = B + C + I,
여기서, B : 순차문의 복잡도의 합,
C : 분기문의 복잡도의 합,
I : 반복문의 복잡도의 합.

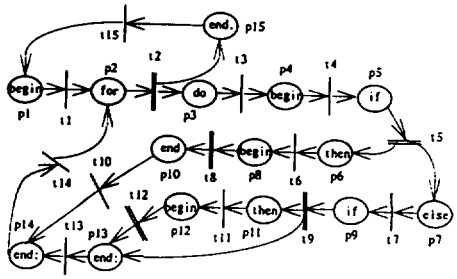
5. 예제와 분석

본 연구에서 척도검증에 이용된 예제 프로그램은 Bieman이 복잡도 측정과 명확도 측정(measurement clarity)에 이용했던 19개의 예제 프로그램 중에서 16개와, 내부 프로시쥬어를 포함한 프로그램의 복잡도 측정을 검증하기 위하여 1개의 프로시쥬어를 포함한 34개의 예제 프로그램을 사용하였다. 따라서 50개의 예제 프로그램을 이용하여 복잡도 측정을 하였고, 이 중 〈예제 프로그램 3〉을 통하여 실제적인 측정을 함으로써 구체적인 실험절차를 예증하도록 한다.

〈예제 3〉

```
 1 program sample2a(input, output) ;
 2 var i,m : integer ;
 3 blnk : char ;
 4 ibn1 : array [30] of integer ;
 5 ibn2 : array [30] of integer ;
 6 bp : array [30] of integer ;
 7 begin
 8 for i : = 1 to m do
 9   begin
10     if (bp[i]+1) < 0 then
11       begin
12         bp[i] : = -1 ;
13         ibn1[i] : = blnk ;
14         ibn2[i] : = blnk ;
15         end
16       else if (bp[i]+1) = 0 then
17         begin
18           ibn1[i] : = blnk ;
19           ibn2[i] : = blnk ;
20           end ;
21   end ;
22 end.
```

(그림 6) 예제 프로그램 3
(Fig. 6) Example program Sample 3



(그림 7) 예제 프로그램 3의 패트리네트
(Fig. 7) Petri net of sample 3

〈표 2〉 예제 프로그램 3의 수행과정과 transition과의 관계
〈Table 2〉 A relation between process flows and transitions in Sample 3

트랜지션	내용
t2	i := 1 to m
t5	(bp[i] + 1) < 0
t8	bp[i] := -1;
	ibn1[i] := blnk;
	ibn2[i] := blnk;
t9	(bp[i] + 1) = 0
t12	ibn1[i] := blnk;
	ibn2[i] := blnk;

〈표 3〉 예제 프로그램 3에서의 Transition 제어구조별 분할
〈Table 3〉 A control structured table of transitions in Sample 3

Basic	Choice	Iteration
t8, t12	t5, t8, t9, t12	t2, t5, t8, t9, t12

(그림 7)에서 우리는 순차문의 경우 트랜지션 T8과 T12이 해당되고 분기문의 경우 트랜지션 t5, t8, t9, t12 그리고 반복문의 경우 트랜지션 t2, t5, t8, t9, t12이 각각 해당됨을 알 수가 있다. 순차문(t8, t12)의 경우 해당 트랜지션에서의 유일한 연산자 n1은 " : = , ; "의 2개이고 n2는 "bp, ibn1, ibn2, i, blank"의 5개가 된다. 총연산자수(N1)는 10개, N2는 14개가 된다. 한편 분기문의 경우, 트랜지션에서 6개, 프레이스(if-then-else, begin-end)에서 2개를 합하여 8개, n2는 순차문과같이 5개를 얻을수 있다. Nw1의 경우 Nested된 분기문이므로 문장10의 경우 d(x) = 3, l(x) = 1을 구할 수가 있다. Nw2의 경우 오퍼랜드의 중첩관계를 나타내므로 문장 10의 경우 d(x) = 2,l(x) = 1. 반복문의 경우는 위의 분기문의 경우와 동일한 방법으로 산출하면 예제 프로그램 3의 복잡도 수치는 다음과 같다:

1) 순차문(Basic Action)
n1 = 2, n2 = 5, N1 = 10, N2 = 14
. B = V / L = 40,

2) 분기문(Choice)
n1 = 8, n2 = 5, Nw1 = 33, Nw2 = 26
C = [Vw * {(n1 * Nw2) / (2 * n2)}] = 4541

3) 반복문(Iteration)
n1 = 9, n2 = 6, Nw1 = 28, Nw2 = 23, 여기서 반복횟수 m은 최대 배열수인 30으로 간주하는데, 특히 여기서는 검토상수의 경우 5항 이하의 경우 거의 0.1의 근접수가 산출되며 특히 6항 이후는 0의 값에 근사하여 6항 이후는 무시할 수가 있겠다.
$I = \sum_{x \in V(i)} [C * R^{Xrev}]$
= 5967

4) 제어적 복잡도 :
CS = B + C + I
= 10548.

6. 비교분석 및 결론

6.1 비교분석 및 평가

본 장에서는 프로그램의 크기 측면에 기반을 둔 Halstead[21]의 effort, 제어흐름 및 제어종속에 기반을 둔 McCabe[17]의 Cyclomatic Complexity, 자료종속 관계에 기반을 둔 Bieman[11]의 DDG Cyclomatic Complexity간의 상관관계를 SPSS/PC+4.0을 이용하여 산출 비교함으로써 본 연구를 통하여 제시한 척도의 타당성을 검증하였다.

〈표 4〉 새로운 복잡도와 기존 복잡도와의 상관관계
〈Table 4〉 Compare with on others complexity measures

	New comp	Halstead Effort	McCabe Cyclo Comp	DDG Cyc Comp
New comp	1	0.8034	0.8219	0.7666
Halstead Effort	0.8034	1	0.8034	0.8751
McCabe Cyclo Comp	0.8219	0.8034	1	0.5815
DDG Cyc Comp	0.7666	0.8751	0.5815	1

본 연구에서 제시한 척도는 Halstead의 척도와 0.8087, McCabe의 Complexity와 0.8219, DDG Cyclo Complexity와 0.7666로 비교적 높은 상관관계가 있음을 알 수 있다.

6.2 결 론

본 연구에서는 소프트웨어 복잡도가 소프트웨

어 프로젝트를 객관적이고 정량적으로 제어할 수 있는 가장 유용한 척도인점에 근거하여 패트리네트를 이용하여 좀 더 쉽고 효율적인 복잡도 방법을 제시하였다.

특히 패트리네트를 이용하여 복잡도를 계산하는 방법이 우선 프로그램을 이해하기 쉽게 모델링화하여 프로그램을 구조적으로 보다 세밀하게 재 표현함으로 이를 순차문, 반복문 그리고 분기문의 구조로 구분하고 이들을 각각의 구조적인 특성을 감안하여 구조적인 가중치를 부여한 수치들을 통하여 보다 명확한 구조적인 계수 산출이 가능한 장점을 가지고 있으나 그 복잡도 수치가 너무 크게 나온다는 단점도 있다.

한편 일반적으로 복잡하고 큰 시스템의 경우 분석되는 패트리네트가 매우 커지거나 복잡하여지는데 이 때에는 패트리네트의 Reduction 방법[25]을 적용시켜 본 연구에서 제시한 제어구조별 복잡도 분석을 활용한다면 보다 효과적인 결과를 얻으리라고 본다.

따라서 앞으로의 연구방향으로는 모듈간의 관계 즉 자료흐름의 관계를 패트리네트로 나타내어 제어흐름과 자료흐름을 모두 분석하는 종합적인 복잡도측정을 위하여 더욱 효율적이고 이해용이한 복잡도 측정방법을 제시하고자 한다.

참 고 문 헌

[1] Joseph P. Cavano, "A framework for the measurement of software quality", The ACM Software Quality Workshop, pp. 133-139, Nov. 1987.

[2] W. Harrison, K. Magel, R. Kluczny and A. Decock, "Applying Software Complexity Metrics to Program Maintenance", IEEE Computer, Vol. 15, No. 9, pp. 65-79 Sep. 1982.

[3] K. B. Lakshmanan, S. Jayaprakash and P. K. Sinha, "Properties of Control-Flow Complexity Measures", IEEE Transactions on Software Engineering, Vol. 17, No. 12, Dec. 1991.

[4] C. A. Petri, "Communication with Automata", New York : Griffiss Air Force Base, Tech. Rep, RADCTR-65-377, Vol. 1, Suppl. 1, 1966.

[5] M. Hack, "Decidability Questions for Petri Nets", Ph. D. dissertation, MIT, Massachusetts, Dec. 1975.

[6] T. G. Lewis, "Software Engineering(analysis and verification)", Reston Publishing Company Inc, 1982.

[7] Bina Ramamurthy and Austin Melton, "A Synthesis of Software Science Measures and the Cyclomatic Number", IEEE Transactions on Software Engineering, Vol. 14, No. 8, Aug. 1983.

[8] Bill Curtis et etl., "Measuring Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", IEEE Trans. on Software Eng. Vol. SE-5, No. 2, Mar. 1979.

[9] Girish Parikh, "The world of software maintenance", Tutorial on Soft-ware Maintenance, pp. 1-10, 1983.

[10] James Martin and Carma McClure, Software Maintenance : The problem and Its Solution, Prentice-Hall, Inc. 1983.

[11] James Michael Bieman, "Measuring Software Data Dependency Complexity", Ph. D. Thesis, Univ. of Louisina, 1984.

[12] James M. Bieman and William R. Edwards, "Experimental Evaluation of The Data Dependency Graph for Use in Measuring Software Clarity", Pro. of the 18th Annual Hawaii International Conferrence on System Science, 1985.

[13] Mark Weiser, "Programmers Use Slices When Debugging", ACM Communica-tion, Vol. 25, pp. 446-452, Jul. 1982.

[14] R. K. Fjeldstad and W. T. Hamlen, "Application Program Maintenance Study Report to Our Respondents", Tutorial on Software Maintenance, pp. 11-27, 1983.

[15] Sallie Henry and Dennis Kafura, "Software Structure Metrics Based on Information Flow", IEEE Trans. on Software Eng. Vol.

SE-7, No. 5, Sep. 1981.

[16] S. S. Yau et etl., "Ripple Effect Analysys of Software Maintenance", IEEE, 1987.

[17] Thomas J. McCabe, "A Complexity Measure", IEEE Trans. on Sortware Eng. Vol. SE-2, No. 4, Dec. 1976.

[18] Warren Harrison et etl., "Applying Software Complexity Metrics to Program Maintenance", Computer Journal, Vol. 15, pp. 65-79. Sep. 1982.

[19] M. H. Halstead, "Elements of Software Science", NY, Elsevie North-Holland, 1977.

[20] G. S. Hura, "Petri net Approach to the Analysis of a Structured Program", Microelectronic Review, Vol. 22, No. 3, pp. 429-431, 1991.

[21] S. N. Woodfield, "Enhanced Effort, Estimation by Extending Basic Programming Models to Include Modularity Factors", Ph. D diseertation, Dep. of Computer Sci., Purdue Univ., Dec. 1980.

[22] 송유진, 이종근, "Petri Nets를 이용한 분할 제어 구조에 따른 소프트웨어 복잡도 연구", 정보과학회 춘계 학술 발표회, pp. 645-648, 1994.

[23] 이종근, 송유진, 전인효, "Petri Net에의한 프로그램 제어별 복잡도연구", 창원대학교 논문집, 제16권, pp. 537-557, 1994.

[24] 김태공, 우치수, "프로그램 경로에 기반을 둔 복잡도의 척도", 정보과학회 논문지, Vol. 20, No. 1, pp. 34-42, 1993.

[25] 이종근, "세미죠인을 기반으로 하는 패트리 넬의 형식적 정의", 정보처리 논문지, Vol. 1, No. 2, pp. 202-214, 1994.

[26] 유철중, 김용성, 장옥배, "객체지향 프로그램의 복잡도 측정을 위한 척도", 정보과학회 논문지, Vol. 21, No. 11, pp. 2039-2049, 1994.

송 유 진

1992년 창원대학교 전산과 졸업 (이학사)
1995년 창원대학교 대학원 전산학 전공(이학석사)
1995년~현재 창원대학교 강사, 창원대학교 정보통신시스템 연구실 연구원
관 심 분 야 : 소 프 트 웨 어 공 학, Petri Nets 이론.

이 종 근

1974년 숭실대학교 전산과 졸업 (이학사)
1978년 고려대학교 경영대학원 경영학 전공(경영학석사)
1986년 숭실대학교 대학원 전산학 전공(공학석사)
1992년 USTL(프) 전산과 박사과정 수료
1987~92년 LSI/USTL 연구원
1993~95년 창원대학교 전산소장
1983년~현재 창원대학교 전산과 교수
관심분야 : Petri Nets 이론, 성능분석 및 검증소프트웨어 복잡도 분석 및 검증.