논문 01-26-5A-6 한국통신학회논문지 '01-5 Vol.26 No.5A

프로그램 동작특성과 실행경로 기반의 테스트 경로 생성과 복잡성 척도

정회원 고 일 석*

A Program Test Path Generation and Complexity Metrics Based on Execution Path and Program Activity Characteristic

Il Seok Ko* *Regular Member*

요 약

소프트웨어의 유지보수 과정에서 효율적인 복잡성 척도와 테스트 경로의 생성은 중요한 문제이다. 대부분의 경우 테스트 경로의 생성과 복잡성 척도의 측정은 독립적인 기법이 필요하다. 본 논문에서는 테스트 경로의 생성과 복잡성 척도를 통합적으로 생성하고 있다. 제안한 기법은 PUT(Program Under Test)를 확장된 페트리네트 그래프(EPG)를 이용하여 모델링하고 이것의 통합적인 분석을 통하여 테스트 경로를 생성하며, 이 과정에서 생성된 실행경로의 제어구조별 평균 발생 빈도수를 이용하여 복잡성 척도 EV(G)를 구하였다. EV(G)는 실제 프로그램의 실행경로에 기반을 두었기 때문에 프로그램의 제어구조별 차이점 외에도 프로그램의 동작 특성을 복잡도에 잘 반영할 수 있다. 본 논문에서 제안한 통합 기법에 의한 테스트 경로 생성 기법과 복잡성 척도를 소프트웨어의 유지보수에 활용한다면 노력과 비용의 절감 및 소프트웨어의 질적 향상을 가져올 것이다.

ABSTRACT

Efficient software complexity metrics and test path generation are important problems. Mostly we uses independent method of test path generation and complexity metrics. In this paper we propose a combined method to generate test path and complexity metrics based on petri nets. First, we generates test path using dynamic combined analysis of EPG which on extended petri nets graph after modeling PUT using EPG. And we propose EV(G) using mean-frequencies of each control structure based on execution path. EV(G) reflects different program control structure of PUT and execution characteristics of PUT, and EV(G) based on really execution path of a program. If we use EV(G), we can reduce the cost and effort and improve the software quality.

Ⅰ. 서 론

소프트웨어 라이프사이클에서 유지보수가 차지하는 상대적인 비용이 점차 높아짐에 따라 소프트웨어의 개발 및 유지보수 비용의 감소와 품질 보증에 큰 관심을 갖게 되었다. 특히 소프트웨어의 품질을 정량적으로 측정하려는 연구, 즉 소프트웨어 척도에 대한 관심이 날로 증대되고 있다. 이러한 정량적인 측정을 소프트웨어 라이프사이클에 적절히 활용한다면 소프트웨어의 품질 향상과 비용 감소에 상당한 진전을 가져다 줄 것으로 기대된다[1,5,11,12,13].

복잡도란 데이터 구조의 형태, 중첩(nesting)된 정도, 조건 분기의 수, 인터페이스의 수와 그리고 다른 시스템 특징 같은 요소로서 결정되는 시스템이나 시스템 요소의 복잡한 정도를 말한다[1,2,4,6,17]. 따라서 소프트웨어의 품질을 정의하고 평가하는데 있어서 복잡도의 측정은 매우 유용하다. 또한 프로그램의 구조를 고려하여 테스트를 실행하는 방법[14,16]으로

* 대덕대학 정보통신과
논문번호: K01051-0206, 접수일자 : 2001년 2월 6일

첫째, 프로그램의 크기에 따른 측정의 경우에 고려할 수 있는 방법으로 실행 가능한 모든 경로를 실행하는 방법이 있으며 이것은 프로그램이 루프를 포함할 경우에는 경로의 수가 무한개가 되는 문제점과 크기가 큰 프로그램이 크기가 작은 프로그램보다 반드시 복잡하다고 할 수 없으므로 프로그램의 크기와 복잡도가 비례하지 않는다는 문제점이 있다. 둘째로 프로그램의 제어 구조에 따른 복잡도를 측정할 경우에 이용할 수 있는 방법으로 프로그램 내부의 각 문장을 적어도 한번은 실행하는 방법이며 이것은 여러 조건의 조합에 의해 발생하는 오류들을 체크할 수 없다는 문제점이 있다[16]. 프로그램의 테스트 경로의 생성과 복잡도는 유지보수과정에서 중요한 문제이며 통합적인 방법으로 실행이 가능해야 한다. 본 논문에서는 PUT(Program Under Test)를 EPG(Extended Petri-net Graph)를 이용하여 정형화하고 이것의 동적인 분석법을 통하여 프로그램의 실행 경로에 기반을 둔 복잡도와 테스트 경로를 생성한다. 이는 소프트웨어의 유지보수 문제에서 중요한 부분을 차지하는 두 가지 문제 즉, 프로그램의 테스트 경로 생성과 복잡성 척도의 측정을 연계시켜 통합기법에 의해 생성 가능하다. PUT의 정형화는 전형적인 패트리네트[3,9]를 확장한 그래프인 EPG를 사용하였으며 이것의 도달성 트리[10]를 이용한 수행경로 분석을 통하여 프로그램의 테스트 경로를 생성하였다. 또한 이때 수행 경로에서 나타난 제어구조별 평균빈도수를 가중치로 이용한 복잡성 척도 EV(G)를 제안하였다. EV(G)는 실제 프로그램의 실행 경로에 기반을 두었기 때문에 프로그램의 제어구조별 차이점 외에도 프로그램의 동적인 동작 특성을 복잡도에 잘 반영할 수 있다. 그림 1은 통합기법에 의한 테스트 경로와 복잡도의 생성 절차를 나타낸 것이다. 먼저 입력된

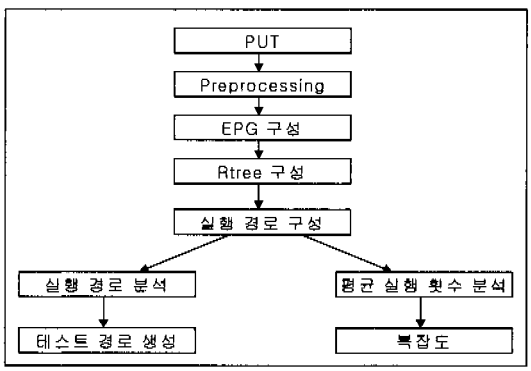


그림 1. 통합기법을 이용한 테스트 경로와 복잡도의 생성 절차

PUT는 전처리(preprocessing)단계를 거쳐 플레이스가 삽입된 확장된 PUT가 된다. 확장된 PUT는 전처리 단계에서는 프로그램의 수행단위별로 플레이스를 삽입한다. 확장된 PUT는 EPn(Extended Petri-net)으로 모델링한 후 EPG로 표현된다. 또한 이것을 프로그램의 동적인 동작 특성을 분석하기 위해 Rtree (Reachability tree)로 구성한 다음 Rtree의 분석에 의해 PUT의 테스트 경로와 실행 경로에서 각 제어구조별 평균 발생 빈도수를 가중치로 가지는 복잡도를 생성하게 된다.

Ⅱ. PUT의 정형화와 테스트 경로의 생성

2.1 PUT의 정형화

PUT를 정형화하기 위해서 PNt의 트랜지션을 프로그램의 제어구조별로 구분 확장한 EPn(Extended Petri-net)을 사용한다. EPn은 정의 1과 같이 6가지 튜플로 구성되며 이것을 이용하여 정의 2의 도식적 표현인 EPG(Extended Petri-net Graph)를 구성한다.

[정의1] EPn = <P, Ts, Tc, Ti, F, B, t>
P = {ps, pf, p1, p2, ···, pm}, P ≠ ∅
Ts = {ts, tf, ts1, ts2, ···, tsn}, Ts ≠ ∅
Tc = {tc1, tc2, ···, tcn},
Ti = {ti1, ti2, ···, tin},
P ∩ Ts ∩ Tc ∩ Ti = ∅,
F : P × T → N, B : T × P → N

정의 1에서 EPn은 플레이스의 유한집합 P, 순차 트랜지션의 유한 집합 Ts, 선택 트랜지션의 유한 집합 Tc, 반복 트랜지션의 유한 집합 Ti, 정방향 영향함수(Forward incidence function) F, 역방향 영향함수(Backward incidence function) B로 구성된다. 이때 Ts는 순차트랜지션(sequential transition), Tc는 분기트랜지션(choice transition), Ti는 반복 트랜지션(iteration transition)이다. 또한 N은 음수가 아닌 정수이며 Ts, Tc, Ti중 하나이고 t는 토큰의 분포이다.

정의 1에는 주어진 PUT에 시작 플레이스 ps, 종료 플레이스 pf, 시작 트랜지션 ts, 종료 트랜지션 tf가 추가되어 있는데 이것은 EPn의 도식적 표현인 EPG가 초기상태로 회복할 수 있게 한다.

정의 2는 EPn을 그래프로 나타낸 EPG를 나타내며 각 트랜지션은 그 트랜지션의 입력에 토큰이 존재할 때 점화 가능하고 하나의 트랜지션의 토큰이

하나 이상 플레이스의 입력이 될 수 없다.

[정의2] 확장된 페트리네트 그래프 EPG=(V, E)

1. V는 노드들의 집합으로 V={Vp, Vtr, Vto}, Vp∩Vtr=∅이다. 이때 Vp는 플레이스 노드이며 구성 요소는 {ps, p1, …, pn, pf}이며 Vtr은 트랜지션 노드이며 구성요소는 {ts, t1, …, tn, tf}이다. 또한 Vto는 토큰을 가진 플레이스 노드이다. 병렬성이 없는 프로그램에서 토큰은 어느 시점에서 하나의 플레이스 노드에 존재하므로 n(Vp∩Vto)=1이다.

2. E는 간선들의 집합이며 플레이스의 출력에서 트랜지션의 입력이 되는 간선 P→T와 트랜지션의 출력에서 플레이스의 입력이 되는 간선 T→P이다.

[정리 1] EPn과 EPG는 논리적으로 동일하다.

(증명) EPG는 EPn의 그래프 표현이므로 EPG에서 EPn을 유도할 수 있으면 된다. 먼저 EPG=(V, E)에서 V={Vp, Vtr, Vto}이고 Vp의 구성 요소는 {ps, p1, …, pn, pf}이므로 ∀pi∈Vp에 대해 EPn에서 pi∈P인 관계가 성립한다. 따라서 EPn의 P와 EPG의 Vp는 동치이다. 마찬가지로 Vtr의 구성 요소는 {ts, t1, …,tn, tf}이므로 ∀ti∈Vtr에 대해 EPn에서 ti∈<Ts, Tc, Ti>인 관계가 성립한다. 따라서 EPn의 트랜지션의 집합인 <Ts, Tc, Ti>와 EPG의 Vtr은 동치이다.

EPG에서 Vto는 한 시점에서 토큰을 가진 플레이스 노드이며 토큰을 가진 플레이스는 그것을 입력으로 갖는 트랜지션의 실행 가능함을 나타내므로, 병행성이 없는 프로그램에 대해서 pi에 토큰이 존재한다면 ti가 실행 가능함을 나타낸다. 따라서 이는 EPn의 토큰의 위치를 나타내는 t와 동일하다. 그리고 E={(T→P),(P→T)}에서 (T→P)와 (P→T)는 EPn의 P× T→N, T×P→N이므로 정방향 영향함수 F와 역방향 영향 함수 B가 된다. 따라서 EPG에서 EPn을 유도할 수 있으므로 EPn과 EPG는 논리적으로 동일하다. ■

2.2 EPG를 이용한 제어구조별 모델링

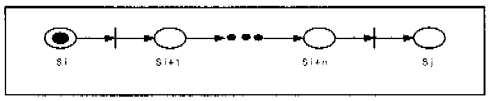
EPG를 이용한 프로그램의 제어 구조별 모델링은 순차문, 조건문, 분기문으로 나누어 생각할 수 있다. 이때 초기 토큰의 상태는 최초의 플레이스에 위치한다고 가정한다.

(1) 순차문

그림 2의 (a)와 같은 순차문은 그림 2의 (b)와 같이 모델링할 수 있다.

```
Si : statement i;
Si+1       : statement i+1;
……
Si+n       : statement i+n;
Sj : statement j;
```

(a) 순차문의 예



(b) 순차문의 EPG

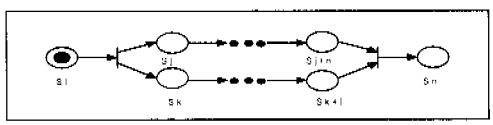
그림 2. 순차문

(2) 분기문

분기문은 다음과 같이 제어 구조에 따라 세 가지로 나누어 모델링 된다.

```
Si  : if(condition){
          Sj  : statement j; ……
          Sj+n : statement j+n;
      };
      else{
          Sk  : statement k; ……
          Sk+1 : statement k+1;
      };
Sn  : statement n;
```

(a) 분기문 1

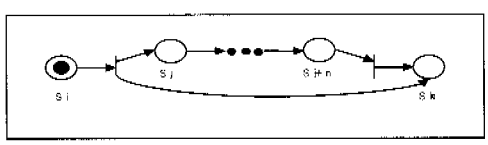


(b) 분기문 1의 EPG

그림 3. 분기문 1

```
Si  : if(condition){
                Sj : statement j;
                ……
                Sj+n : statement j+n;
      };
Sk  : statement k;
```

(a) 분기문 2

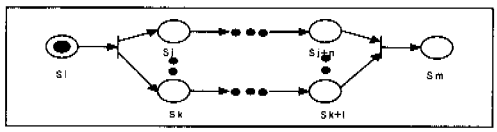


(b) 분기문 2의 EPG

그림 4. 분기문 2

```
Si  : switch(conditon){
        case() : Sj  : statement j; ·····
                 Sj+n : statement j+n;break;
        case() : Sk  : statement k; ·····
                 Sk+i : statement k+i;break;
        }
Sm  : statement m;
```

(a) 분기문 3



(b) 분기문 3의 EPG

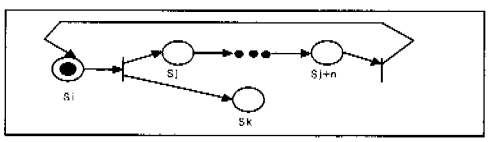
그림 5. 분기문 3

(3) 반복문

분기문과 마찬가지로 제어 구조에 따라 세 가지로 구분하여 모델링 된다.

```
Si  : while(condition){
        Sj      : statement j;
                  ·····
        Sj+n : statement j+n;
      }
Sk  : statement k;
```

(a) 반복문 1

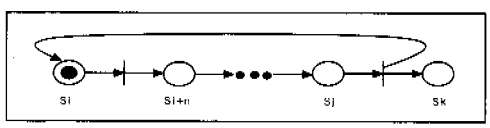


(b) 반복문 1의 EPG

그림 6. 반복문 1

```
      do{
        Si  : statement i;
              ·····
        Si+n : statement i+n;
Sj  : }while(condition)
Sk  : statement k;
```

(a) 반복문 2

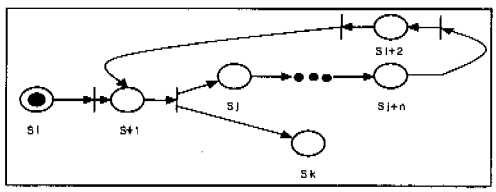


(b) 반복문 2의 EPG

그림 7. 반복문 2

```
Si  : for(초기값; Si+1:종료조건; Si+2:증감치){
        Sj  : statement j;
              ·····
        Sj+n : statement j+n;
    }
Sk  : statement k;
```

(a) 반복문 3



(b) 반복문 3의 EPG

그림 8. 반복문 3

그림 9는 입력된 PUT를 그림 1의 전처리 단계를 거쳐 플레이스가 삽입된 상태이고 그림 10은 그림 9의 예제에 대한 EPG를 나타낸다. 그림 10을 분석하면 P = { ps, pf, p1, p2, ···, p7, p8 }이고 Ts = { ts, tf, t3, t4 t7, t8 }, Ti = { t1, t5 }, Tc = { t2, t6 } 이다. 표 1은 그림 10의 EPG에 대한 각 트랜지션과 수행과정과의 관계를 나타낸 것이다.

```
p1 : while(x){
  p2 : if(c1){
         p3 : then A1
         p4 : else A2
       }
         p5 : while(y){
            p6 : if(c2){
                   p7 : then A3
                   p8 : else A4
                 }
          }
        }
```

그림 9. 전처리된 예제 프로그램

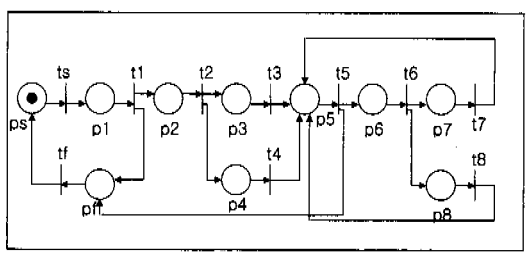


그림 10. 예제에 대한 EPG

표 1. 예제 프로그램에 대한 transition

transition	transition의 종류
ts, tf, t3, t4, t7, t8	순차 transition
t1, t5	반복 transition
t2, t6	분기 transition

표 2. 예제 프로그램의 분석

구 분		갯 수
place의 수		10
transition의 수	sequential	6
	choice	2
	iteration	2

2.3 도달성 트리의 생성

EPG의 실행 경로를 분석하기 위하여 페트리네트의 동적분석에 사용되는 도달성 트리를 이용한다. 알고리즘 1은 EPG를 입력으로 토큰의 이동에 따른 트랜지션의 점화 가능성을 이용하여 도달성 트리를 생성하는 알고리즘이다. 정의 3은 EPG의 구성에 대해 도달성 트리(Rtree)를 생성하는 과정에서 현재 경로의 종료 조건을 위한 완전트랜지션의 정의이다. 그림 10에서 완전트랜지션은 {t1, t5}이며, 그림 11과 같이 Rtree를 구성할 때 현재 생성중인 경로에 대한 종료조건은 이들 트랜지션에 토큰이 존재하거나, 종료 트랜지션 또는 정의 9, 10과 같은 순환관계가 존재할 때이다. 정의 4는 EPG의 제어구조에서 분기문이나 반복문에서 발생하는 형제 트랜지션의 정의이다.

[정의 3] 완전트랜지션(complete transition)

어떤 트랜지션에서 종료트랜지션으로 전이가 일어날 수 있을 때, 즉 출력함수 (O(ti), Pf)가 존재할 때 이 트랜지션을 완전트랜지션이라 한다.

[정의 4] 형제트랜지션(civiling transition)

EPG에서 시작트랜지션에서 전이가 일어날 때 그 트랜지션까지의 전이의 깊이(레벨)가 같은 트랜지션을 형제트랜지션이라 한다.

[정의 5] 입력함수(input function)

입력함수 (Pi, I(tj))는 플레이스 Pi의 출력이 트랜지션 tj의 입력이 되는 경우를 말한다.

[정의 6] 출력함수(output function)

출력함수 (O(ti), Pj)는 트랜지션 ti의 출력이 플레이스 Pj의 입력이 되는 경우를 말한다.

[정의 7] Rtree

Rtree는 EPG의 수행 경로 분석에서 나타나는 트랜지션에서 토큰의 점화 가능성을 나타내는 특성 벡터를 노드로 하고 수행경로를 간선으로 하는 트리이다.

Rtree의 생성은 PUT의 정적인 분석을 통해 이루어지며, PUT의 동적인 특성은 알고리즘2의 실행경로를 구하는 과정에서 분석된다. 이때 병행성이 없는 경우의 토큰의 분포는 한 노드에서 하나의 트랜지션에만 존재한다. 알고리즘 1의 Rtree의 생성에서는 EPG의 각 플레이스와 트랜지션을 모두 노드로 취급하며 이때 노드의 입력은 ps→ts→p1→t1→p2→t2→…의 순으로 들어온다고 가정한다. 따라서 각 플레이스에서 트랜지션으로의 입력노드의 변화에서 그 트랜지션이 점화되는 것으로 간주한다.

[알고리즘 1] Rtree의 생성

```
input=EPG
output=Rtree
node()={a set places, a set of transitions}
algorithm make_Rtree()
{
  if(node(i)=place)
  discard(node(i));
  else if(node(i)=transition)
  {
    if(node(i)=ts){
          Rtree(root_node)=start transition of EPG;
    }
    if(input_function of transition is child node of
          Rtree(root_node)){
            node(i) is child node of root_node;
       s3:flag=1;
       while(flag){
       if(node(i) is not complete transition){
          if(input function of node(i) is
            output function of parents node){
              node(i) is child node of parents
node;
                  } /* end of if */
```

```
        else if(node(i)=final node){
              complete make_node of this path;
                  flag=0;
                 } /* end of else if */
        else if(there is not_complete transition)
                  flag=0;
        } /* end of while */
        if((generate_node is complete transition)||
            (there is civiling transition(i){
                        exit();
            } /* end of if */
        else goto s3;
    } /* end of else if */
} /* end of algorithm */
```

알고리즘의 수행을 살펴보면 먼저 EPG에서 시작 플레이스의 시작트랜지션을 도달성 트리의 루터 노드로 한다. 그리고 다음으로 입력되는 노드가 플레이스이면 Rtree는 트랜지션에서 토큰의 분포만으로 구성하므로 그 플레이스에 해당하는 트랜지션에 대해 수행을 계속한다. 시작플레이스의 트랜지션의 출력함수의 플레이스를 입력함수로 갖는 트랜지션은 루터 노드의 자식 노드이며 이때 생성되는 자식노드들은 루트 노드에서 같은 레벨에 있으므로 형제 노드가 된다. 또한 완전 트랜지션이 아닌 노드에 대해 그 트랜지션의 출력함수의 플레이스를 입력함수로 갖는 트랜지션은 그 노드의 자식 노드가 된다. 그리고 이전 과정에서 생성된 노드가 종료 트랜지션인 경우는 그 노드의 자식 노드의 생성은 완료되었으며, 생성한 노드 중 완전 트랜지션이 아닌 노드가 존재할 경우에는 이전 과정에서 생성된 노드 중 완전 트랜지션이 아닌 노드가 존재할 경우 그 노드의 상위 노드들 중에서 그 노드와 같은 노드가 있으면 그 경로의 수행을 종료한다. 마지막으로 생성된 노드들이 모두 완전트랜지션이거나 그 상위 노드들 중에서 같은 노드가 있는 경우에는 수행을 종료하고 아니면 알고리즘을 계속해서 수행하게 된다.

그림 11은 그림 10의 EPG를 위의 알고리즘에 따라 도달성 트리로 구성한 것이다. 각 노드는 EPG에서 트랜지션(ts,t1,t2,t3,t4,t5,t6,t7,t8,tf)의 점화 가능성을 나타내는 토큰 분포의 특성벡터를 나타내며 1의 값을 가진 트랜지션은 현재 토큰을 가지고 있으며 점화가 가능함을 나타낸다. 예를 들면 (0000010000)을 트랜지션 t5의 위치에 토큰이 존재하므로 트랜지션 t5가 점화 가능함을 나타낸다.

2.4 제어 구조에 대한 테스트경로의 수

PUT의 블록에 대한 테스트 경로의 수는 서로 다른 제어 구조를 고려하여 얻을 수 있다. 먼저 순차문에서는 블록 내에 서로 다른 n개의 실행 경로가 존재한다면 테스트경로의 수는 n개이다.

분기문에서는 IF THEN ELSE 구조에 대해서 IF THEN의 경로의 수가 n개이고, ELSE의 경로수가 m개인 경우 테스트 경로의 수는 n+m이다. IF THEN의 구조에 대해서는 ELSE의 m개의 경로는 고려할 필요가 없다. 그러나 조건을 만족시키지 않는 하나의 경로가 존재하게 되므로 테스트 경로의 수는 n+1 개가된다.

반복문에서는 루프내의 실행 경로가 경우에 따라서는 무한개가 될 수 있으므로 전체 실행 경로의 생성 및 테스트는 불가능하다. 따라서 프로그램의 동적인 실행 경로 중에서 다른 실행 경로를 적어도 한번은 실행하고 PUT내부의 문장을 적어도 한번은 수행할 수 있는 최소 경로의 집합을 구하는 것은 중요한 문제이다.

2.5 테스트 경로의 생성

알고리즘 1에 의해 생성된 도달성트리는 PUT의 정적인 특성을 나타낸 것이다. PUT의 동적인 특성은 생성된 Rtree 의 분석에서 얻어진다. 알고리즘 2는 알고리즘 1에서 생성한 Rtree를 분석하여 동적인 실행경로를 생성하는 알고리즘이다.

[알고리즘 2] 도달성 트리에서 실행 경로의 생성

```
input : Rtree
output : a set of exe_path
algorithm exe_path_generation()
{
  node=(node of Rtree);
  flag=1;
  clear(path); /* path is a set of node */
  while(flag){
    do{
    append_path()=getnode();
    } while(not final node of current path);
    if(final_node=tf){
          exe_path=root_node to final_node;
          append(exe_path);
          flag=0;
       }; /* end of if */
```

```
    else if{final_node≠tf){
        if(there is a same node of current node in
            a child node){
            exe_path=root_node to final_node;
            append(exe_path);
            flag=0;
        }; /* end of if */
    }; /* end of else if */
s2:if(there is a different node of current node in child
    node but it is civiling node of current node)
  {switch(path){
    case(there is direct loop):
        insert loop to make new path between loop:
        flag=0;
    case(there is indirect loop):
        insert loop from ti to tj to make new path;
        flag=0;
    case(there is no loop):
        make new path from ti→tj→tk;
        flag=0;
    }; /* end of switch */
  }; /* end of if */
  if(the final node of extended_exe_path is tf )
        exit();
   else goto s2;
 } /* end of while */
} /* end of algorithm */
```

Rtree는 그림 11에서와 같이 각 트랜지션에서 토큰의 분포만으로 구성하며 이것은 토큰이 위치한 해당 트랜지션이 점화 가능함을 나타낸다. 또한 프로그램의 정적인 실행 경로는 제어구조에 따라 순환관계를 가질 수 있으며 따라서 정의 9와 정의 10과 같은 트랜지션만을 고려한 두 가지 순환관계를 정의할 수 있다.

[정의 9] 직접 순환 경로(direct loop)

도달성 트리의 서로 다른 두 트랜지션 노드 ti, tj에서 ti→ … →tj의 경로가 존재하며 tj→ … →ti의 경로가 존재할 경우 직접 순환 경로가 존재한다.

[정의 10] 간접 순환 경로(indirect loop)

도달성 트리의 서로 다른 트랜지션 노드 ti, tj, tk에서 ti→ … →tk의 경로와 tk→ … →ti의 경로가 있을 경우 결국에는 ti→ … →tj의 순환 관계가 있으며 이때 간접순환 경로가 존재한다.

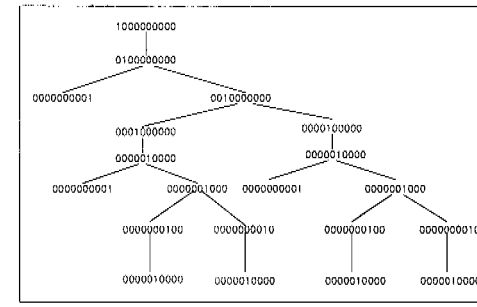


그림 11. 도달성 트리

알고리즘의 실행을 살펴보면 Rtree에서 하나씩 노드를 읽어서 종료 노드가 종료트랜지션인 경우는 루트 노드에서 그 종료 노드까지가 하나의 실행 경로이므로 실행경로에 추가한다. 또한 종료 트랜지션 이외의 트랜지션이 종료 노드가 되는 경우는 첫째, 자신의 노드와 자신의 하위 노드들 중에 같은 노드가 존재하면(ti → … → ti)그 사이에 존재하는 노드를 자신의 노드가 다른 실행 경로를 구성할 때 삽입하여 새로운 실행 경로를 만든다. 둘째, 자신의 하위 노드들 중에 자신의 노드와는 다르지만 자신의 노드와 형제 트랜지션일 때는 다음과 같이 세 가지 경우로 나누어 생각할 수 있다.

1) 직접 순환관계가 있을 경우 : 순환관계를 이루는 두 노드들 사이의 노드들을 자신의 노드가 다른 실행 경로를 생성할 때 삽입하여 새로운 실행 경로를 만든다.

2) 간접 순환 관계가 있을 경우 : 서로 다른 두 노드 ti에서 tj로 간접 순환관계를 이룰 경우에는 ti 밑의 모든 테스트 경로에 tj노드의 자식인 순환되는 노드를 삽입하여 자신의 노드인 ti 노드까지 실행을 생성한다. 3) 순환 관계가 없을 경우 : 서로 다른 노드 ti, tj, tk에서 ti→ … →tk의 경로와 tk→ … →tj의 경로가 존재할 경우에는 순환 관계가 존재하지 않으며 이때의 실행 경로는 ti→…→tk→…→tj가 된다. 셋째, 확장된 실행 경로의 모든 종료노드가 종료트랜지션이면 실행 경로의 생성은 종료되며 종료노드 중에서 종료트랜지션이 아닌 노드가 존재할 경우는 step2를 진행한다. 표 3은 그림 11의 Rtree에서 구성된 정적인 실행 경로이며, 표 4는 알고리즘 2에 의해 수행된 동작특성을 고려한 실행 경로이며 이것은 프로그램의 동작특성을 나타낸다는 의미에서 동적인 실행경로 (dynamic execution path)라 할 수 있다.

표 3. 정적인 실행 경로

경로번호	실행경로
1	ts t1 tf
2	ts t1 t2 t3 t5 tf
3	ts t1 t2 t3 t5 t6 t7 t5
4	ts t1 t2 t3 t5 t6 t8 t5
5	ts t1 t2 t4 t5 tf
6	ts t1 t2 t4 t5 t6 t7 t5
7	ts t1 t2 t4 t5 t6 t8 t5

알고리즘 3은 알고리즘 2에서 생성된 실행 경로에 대해 테스트 경로를 생성하는 알고리즘이다. 테스트 경로 생성은 알고리즘 3과 같이 알고리즘 2에 의해 생성된 확장된 실행 경로들 간의 포함관계를 이용한다. 한 경로가 그 경로를 포함하는 다른 경로에 포함된다면, 가장 많은 경로를 포함한 경로의 테스트가 그 경로가 포함한 다른 모든 경로상의 프로그램의 부분을 최소한 한번은 테스트하게 된다. 따라서 가장 많은 경로를 포함한 경로를 선택하여 테스트 경로를 생성한다. 표 4에서 경로번호 3.3은 경로 1, 2, 3.1, 3.2를 포함한 테스트 경로가 되며 경로 4.2는 경로 1, 2, 4, 4.1, 4.3을 포함하는 테스트 경로가 된다.

표 4. 동작 특성을 고려한 실행 경로

경로	실 행 경 로
1	ts t1 tf
2	ts t1 t2 t3 t5 tf
3	ts t1 t2 t3 t5 t6 t7 t5
3.1	ts t1 t2 t3 t5 t6 t7 t5 tf
3.2	ts t1 t2 t3 t5 t6 t7 t5 t6 t7 t5 tf
3.3	ts t1 t2 t3 t5 t6 t7 t5 t6 t8 t5 tf
4	ts t1 t2 t3 t5 t6 t8 t5
4.1	ts t1 t2 t3 t5 t6 t8 t5 tf
4.2	ts t1 t2 t3 t5 t6 t8 t5 t6 t7 t5 tf
4.3	ts t1 t2 t3 t5 t6 t8 t5 t6 t8 t5 tf
5	ts t1 t2 t4 t5 tf
6	ts t1 t2 t4 t5 t6 t7 t5
6.1	ts t1 t2 t4 t5 t6 t7 t5 tf
6.2	ts t1 t2 t4 t5 t6 t7 t5 t6 t7 t5 tf
6.3	ts t1 t2 t4 t5 t6 t7 t5 t6 t8 t5 tf
7	ts t1 t2 t4 t5 t6 t8 t5
7.1	ts t1 t2 t4 t5 t6 t8 t5 tf
7.2	ts t1 t2 t4 t5 t6 t8 t5 t6 t7 t5 tf
7.3	ts t1 t2 t4 t5 t6 t8 t5 t6 t8 t5 tf

[알고리즘 3] 테스트 경로의 생성

input : dynamic execution path

output : a set of test path tp(k),

k=1, ···, number of dynamic execution path

```
{
k=1;
for i=1 to end of dynamic execution path
   {
   if (path(i) is included by other path)
     {
     select execution path which include
     maximal other execution path;
     } /* end of if */
     else if(path(i) is independent of other path)
        select path(i);
     tp(k)=path(i); k=k+1;
     } /* end of for */
} /* end of algorithm */
```

또한 경로 6.3은 경로 1, 2, 5, 6.1, 6.2를 포함하므로 테스트 경로가 되며 경로 7.2는 경로 1, 2, 5, 7.1, 7.3을 포함하는 테스트 경로가 된다. 따라서 표 4에서 진하게 표시된 경로 3.3과 4.2, 6.3, 7.2를 테스트함으로서 테스트의 효율을 높일 수 있다.

PUT는 전처리 단계를 거쳐서 플레이스를 삽입하게 된다. 이때 그림 9와 같이 전처리 단계를 거쳐 삽입된 플레이스는 PUT 내부의 실행문장에 해당한다. 또한 EPG에서 ··· → P_n → t_n → P_{n+1} → t_{n+1} → ···의 관계가 성립하므로 각 플레이스의 실행은 EPG에서 트랜지션의 실행을 의미한다.

표 4와 같이 동적인 실행경로는 트랜지션의 집합에서 서로 다른 트랜지션을 모두 포함한다. 또한 알고리즘 3에 의해 생성된 테스트 경로는 실행경로상에서 가장 많은 실행경로를 포함한 경로의 집합이므로 EPG의 트랜지션의 집합을 모두 포함한다.

그러므로 알고리즘 3에 의해 생성된 테스트 경로는 PUT 내부의 각 문장을 적어도 한번은 실행할 수 있다. 또한 알고리즘 2에 의해 생성된 실행경로는 PUT의 실행에서 생성될 수 있는 모든 실행경로이다. 알고리즘 3에 의해 생성된 테스트 경로는 이들 경로 중에서 가장 많은 실행 경로를 포함한 경로와 포함관계를 가지지 않는 독립된 경로를 모두 원소로 가지는 집합이다. 따라서 알고리즘 3에 의해 생성된 테스트 경로는 실행 가능한 모든 경로 중 서로 다른 실행 경로를 적어도 한번은 포함하며 생성된 테스트 경로는 PUT의 실행 가능한 경로 중 서로 다른 실행 경로를 적어도 한번은 실행할 수 있다.

Ⅲ. 복잡도의 측정 및 비교 분석

주어진 프로그램에 대한 실행 경로 분석에 대한 정보는 4의 테스트 경로의 생성에서만이 아니라 복잡도의 측정에도 적용할 수 있다.

본 논문에서는 이들 정보를 이용한 복잡성 척도인 EV(G)(Extended V(G))를 제안한다. EV(G)는 McCabe[15]의 V(G)를 확장한 것이며 또한 이는 프로그램의 제어구조와 실제 프로그램의 동작 특성을 고려한 실행경로에 기반을 두고 있다. EV(G)는 프로그램의 제어구조에 기반을 두고 있으며 EV(G)는 EPG의 도달성트리를 이용한 분석을 이용하여 프로그램의 실제 동작 특성을 고려한 실행경로를 기반으로 하기 때문에 프로그램의 실제 실행 상에서 특성을 잘 반영하고 있으며 복잡도의 측정뿐만이 아니라 테스트 경로의 생성 또한 가능하다. EV(G)에서 프로그램의 복잡도에 영향을 미치는 요인은 다음과 같다.

NTs : the number of sequential transition
NTc : the number of choice transition
NTi : the number of iteration transition
NP : the number of Places
ΣNi : 테스트 경로에서 반복트랜지션의 발생횟수의 합
ΣFi : 테스트 경로에서 반복문이 발생한 경로수의 합
ΣNc : 테스트 경로에서 분기트랜지션의 발생횟수의 합
ΣFc : 테스트 경로에서 분기문이 발생한 경로수의 합
m : $\lfloor \Sigma Ni / \Sigma Fi \rfloor$
(실행 경로상에서의 반복문의 평균 발생 빈도수)
t : $\lfloor \Sigma Nc / \Sigma Fc \rfloor$
(실행 경로상에서의 분기문의 평균 발생 빈도수)
Co : 연결요소의 수

여기에서 NTs는 PUT의 EPG에서 나타난 순차 트랜지션의 개수를 나타내며 NTc는 분기 트랜지션의 수를, NTi는 반복 트랜지션의 수를 나타낸다. m은 전체 테스트 경로에서 반복문의 평균 발생 빈도수이고 t는 전체테스트 경로에서 분기문의 평균 발생 빈도수이다. 이때 Co는 연결요소의 수이며 단일 모듈에 대한 값은 1이다.

위의 요소들을 이용하여 EV(G)척도를 다음과 같이 정의한다.

$$EV(G) = NTs + t \times NTc + m \times NTi - NP + 2Co$$

EV(G)는 V(G)의 제어흐름 그래프에서 에지의 수에 해당하는 e를 프로그램의 동적인 동작 특성과 제어구조별 차이점을 반영하기 위해, 실제 실행 경로의 분석에서 각 제어구조별로 발생한 평균 발생 빈도수를 가중치로 이용하여 확장한 것이다. 예제에 대한 EV(G)의 값을 계산하면 NP=10, NTs=6, NTc=2, NTi=2이고, t=2, m=3이므로 EV(G)=6+2*2+3*2-10+2=8이 된다. <표 5>는 V(G)와 EV(G)를 비교한 것이다.

표 5. V(G)와 EV(G)의 비교

구분	V(G)	EV(G)
고려 사항	제어 흐름	1. 제어 흐름 2. 제어구조별 실행경로의 평균발생빈도수를 가중치로 이용 3. 실행경로 기반이므로 프로그램의 동적인 동작특성 반영
그래프 이용	제어흐름 그래프	페트리네트

표 6. 예제 프로그램의 제어구조별 특성

PUT	제어구조별 특성
PUT01	두 개의 분기문이 순차적으로 연결
PUT02	두 개의 분기문이 중첩구조를 가짐
PUT03	하나의 반복구조내에 하나의 분기문을 가짐
PUT05	하나의 반복 구조내에 두 개의 분기문이 중첩구조를 가짐
PUT08	두 개의 분기문이 중첩 구조를 가짐
PUT13	하나의 반복 구조내에 하나의 분기문
PUT21	두 개의 반복문이 중첩 구조를 가지면서 두번째 반복문내에 분기문을 가짐
PUT23	두 개의 반복문이 중첩구조를 가지며 분기문을 포함

<표 5>에서와 같이 V(G)가 제어의 흐름에만 기반을 두고 있지만 EV(G)는 제어의 흐름뿐만 아니라 각각 서로 다른 제어구조를 실행 경로상의 평균 발생 빈도수를 이용하여 구별할 수 있고 실제 프로그램의 실행 경로를 기반으로 하기 때문에 프로그램의 동작 특성을 잘 반영하고 있다. 또한 EV(G)는 실행 경로의 분석을 통하여 얻어지므로 테스트 경로의 생성이 용이하다.

표 7은 비교분석에 사용한 프로그램의 복잡도의

측정결과를 나타낸 것이다. 측정에는 30개의 PUT를 사용하였으며 Halstead의 effort와 V(G)를 대상으로 하였다. 표 6에 나타낸 것은 측정에 사용된 30개의 PUT 중에서 제어구조별 차이점을 구별하기 위해 제어구조가 다른 것만을 나타낸 것이다. 또한 표 6은 표 7의 실험 프로그램에 대한 제어구조별 특성을 나타낸 것이다.

표 7. 예제 프로그램의 복잡도

PUT	Halstead effort	V(G)	EV(G)
PUT01	285	3	2
PUT02	928	4	2
PUT03	1470	3	3
PUT05	4280	4	4
PUT08	978	4	4
PUT13	3574	3	5
PUT21	4080	4	11
PUT23	3729	4	11

표 7에서 Halstead의 척도를 함께 나타낸 것은 프로그램의 크기를 나타내기 위한 것이다. 표 7과 표 6에서와 같이 V(G)와 EV(G)는 반복문이 적은 구조에서는 비슷한 값을 가지지만, 중첩된 반복 구조와 같은 복잡한 제어구조에 대해서는 EV(G)의 값이 높음을 알 수 있다. 이는 V(G)보다 EV(G)가 서로 다른 제어 구조별 차이점을 잘 반영하기 때문이다. 따라서 이 두 가지 척도의 상관관계는 제어구조가 복잡할수록 낮아짐을 알 수 있다.

표 8. 상관관계

구분	V(G)	EV(G)
V(G)	1	0.7835
EV(G)	0.7835	1

Ⅳ. 결론 및 향후 연구 방향

테스트 경로의 생성과 복잡도는 소프트웨어 유지보수 과정에서 중요한 문제이기에 지금까지 많은 연구가 진행되었으며 이 두 가지를 효율적으로 통합할 수 있다면 소프트웨어의 생산성 향상에 도움을 줄 것이다. 본 논문에서는 소프트웨어의 유지보수 문제에서 중요한 부분을 차지하는 두 가지 문제 즉, 프로그램의 테스트 경로생성과 복잡성 척도의 측정을 페트리네트 기반으로 연계시킨 통합기법에 의해 생성 가능함을 보였다. 제안한 기법은 PUT (Program Under Test)를 확장된 페트리네트 그래프를 이용하여 정형화하고 이것의 동작 특성을 고려한 분석을 통하여 프로그램의 실행 경로와 제어의 흐름에 기반을 두었기 때문에 프로그램의 제어구조별 차이점 외에도 프로그램의 동적인 동작 특성을 복잡도에 잘 반영할 수 있다. 하지만 본 논문에서 제안한 기법의 효용성을 높이기 위해서는 다음과 같은 점에 대해 향후 연구가 계속 되어야 할 것이다. 첫째, 본 논문에서 제안한 기법을 좀더 규모가 큰 프로그램에 대해 적용하기 위해서는 EPG로 모델링하는 과정과 분석과정의 복잡성을 줄일 수 있어야 한다. 이를 위해 프로그램의 동작 특성을 고려한 페트리네트의 축약에 관한 연구가 필요하다. 둘째, 데이터의 종속성 및 제어의 종속성을 표현할 수 있도록 EPG를 확장한다면 병행 수행이 가능한 테스트 경로를 생성할 수 있고 그에 따라 수행 시간의 단축이 가능할 것이다. 또한 본 논문에서 테스트 경로의 생성과 제어구조별 복잡도의 분석에 사용된 페트리네트는 동시 시험 가능한 수행경로의 분석에 활용한다면 페트리네트의 장점을 좀 더 살릴 수 있을 것이다.

따라서 위와 같은 문제들에 대해 계속적인 연구가 이루어져야 할 것이다.

참 고 문 헌

[1] M. Shepperd, "Software Engineering metrics : Measures and Validations", McGRAW-HILL BOOK COMPANY, pp.18-51, 1993.

[2] 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.

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

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

[5] 유철종, 김용성, 장옥배, "객체지향 프로그램의

복잡도 측정을 위한 척도", 정보과학회 논문지, Vol.21, No.11, pp.2039-2049, 1994

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

[7] Hwang CH, Lee DI "A Concurrency Characteristic in Petri net Unfolding", In: Proceeding SMC'97, 1997.

[8] Richard P., Xie X., "Scheduling sequential and Flexible machines Using Timed Petri nets," In: Proceeding ICATPN'97, 1997, pp.151-165

[9] T. Murata. Petri Nets : Properties, analysis and applications. Proceedings of the IEEE, 77(4):541-580, April 1989.

[10] M. Notomi and T. Murata, "Hierarchical Reachability Graph of Bounded Petri Nets for Concurrent Software Analysis", IEEE Transaction on Software Engineering Vol.20, No.5, pp.325-336, 1994.

[11] Bindu Mehra, "A Critique of Cohesion Measures in the Object-Oriented Paradigm", Masters Thesis, Department of Computer Science, Michigan Technological university, 1997.

[12] Sunghee Park et al., "Metrics Measuring Cohesion and Coupling in Object-Oriented Programs", Journal of Korean Information Science Society, vol.25, no.12, pp.1779-1787, 1998.

[13] Lionel C. Briand, S. Morasca, V. R. Basili, "Defining and Validating High-Level Design Metrics", IEEE Transactions on Software Engineering. vol.25, no.5, pp.722-743, 1999

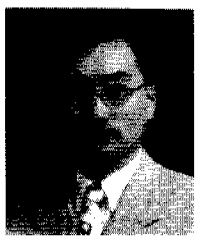
[14] Program Testing Strategy," IEEE Transactions on Software Engineering, Vol.SE-9. No.3, pp.347-354, MAY 1983.

[15] T. McCabe, "A Compexity Measure," IEEE Transactions on Software Engineering, Vol.SE-2, No.4, pp.308-320, DEC. 1976.

[16] 이용근, 최형진, 양해술, "프로그램의 구조를 고려한 테스트 경로 생성과 시간 복잡도," 정보과학회 논문지, Vol.21, No.10, pp.1883-1889, 1994.

[17] HeungSeok Chae, YoungRae Kwon, "A Cohesion Measure for classes in Object-Oriented Systems", Proceedings of the 5th International Symposium on software Metrics, 1998.

고 일 석(IL Seok Ko) 정회원



현) 연세대학교 대학원 컴퓨터과학산업시스템공학과 박사과정
경북대학교 대학원 컴퓨터공학과 졸업
경북대학교 전자계산기공학 전공 졸업
USIU MBA 과정 수료
성균관대학교 경영대학원 전략경영 과정 수료
Ansoff Associates 전략경영 컨설턴트과정 수료
현) 대덕대학 정보통신과 전임
현) 사)한국컨텐츠학회 사업이사
현) 기술신용보증기금 기술경영 자문단 자문위원
전) 컴퓨터아카데미 대표
전) 문경대학 전자계산기과 전임
<주관심 분야> 컨텐츠공학, KM, 에이전트 기반 시스템, e-비지니스 시스템 및 평가 모델
e-mail : isko@mail.ddc.ac.kr