교환 소프트웨어 복잡도 연구

The Switching Software Metrics and Their Fault Analysis

이재기(J.K. Lee) ACE시스템팀 선임연구원
신상권(S.K. Shin) ACE시스템팀 연구원
이수종(S.J. Lee) 음성DB기술팀 책임연구원
남상식(S.S. Nam) ACE시스템팀 책임연구원, 팀장

소프트웨어 관리 모델은 크게 소프트웨어 프로젝트 견적 모델과 소프트웨어 설계평가 모델, 소프트웨어 복잡성 모델, 소프트웨어 신뢰도 성장 모델, 소프트웨어 프로세스 개선 모델 등으로 나누어진다. 그 중에서도 개발된 소프트웨어를 정량적으로 분석하여 평가하는 모델이 소프트웨어 복잡도 모델이다. 본 논문은 이런 관점에서 대표적인 소프트웨어 복잡성 모델에 대한 적용법에 대해 기술하고 개발중인 교환시스템의 소프트웨어에 대해 volume metrics와 process complexity metrics 방법에 대한 분석 결과와 기타 시스템 개발을 수행하는 과정에서 발생되고 있는 문제점들에 대해 다각도로 분석을 하여 이를 연구개발 및 프로젝트 관리에 활용하고자 한다.

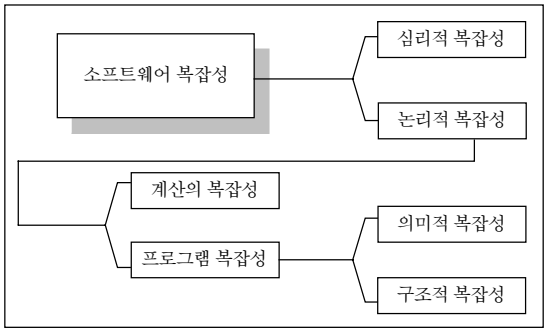
I. 서 론

일반적으로 소프트웨어 복잡성은 소프트웨어 작성(coding)이나 이해가 난해한 부분이 나타나는 특성이 있을 때 크게 2가지 관점인 이론적 복잡성(theoretical complexity)과 심리적 복잡성(psychological complexity)으로 구분된다.

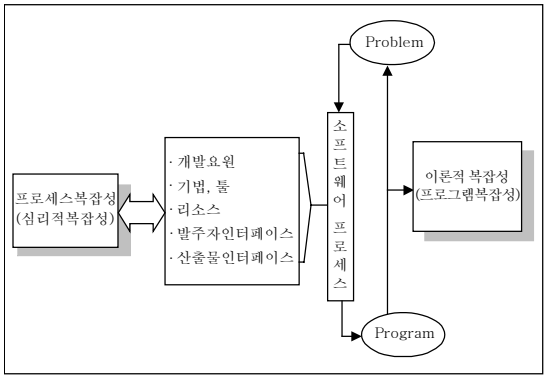
이론적 복잡성은 소프트웨어 고유의 성질을 반영, 소프트웨어 구현(실현)의 문제나 알고리즘의 복잡성에 대한 것이다. 이것은 소프트웨어의 본질적인 성질에 착안한 연산(혹은 계산)의 복잡성(computational complexity)과 현상적인 성질에 근거한 프로그램의 복잡성으로 구분된다.

전자는 프로그램 실현시 수반되는 문제로써 요구사항을 반영하는 데 나타나는 현상으로 이런 것들을 이론적 복잡성이라 칭한다. 다시 말해서 문제의 해결을 구하는 알고리즘을 실행 머신 위에서 실행하는 데에는 필요한 메모리 용량이나 수행시간(execution time)이 필요하다. 소프트웨어 science 분야의 제반 계산상의 복잡성이 여기에 속한다. 후자는 설계상의 프로그램 결과가 구조적인 복잡성에 대해서 프로그램을 구현하는 것이다. 여기에는 의미적 구조와 형식적 구조에 착안한 2개의 복잡성이 존재하게 된다. 의미적 구조의 복잡성에 대한 연구 사례는 거의 없으나 형식적 복잡성에 대해서는 여러 측면에서 다수의 연구 결과가 발표되고 있다. 이것에 대한 연구 결과는 주로 프로그램에 포함된 정보량이나 조건분기수로 표시하는 복잡성 연구가 대표적이다. 즉, 주요 소스코드에 대해 소스 비용을 계측 대상으로 삼아 예측하는데, 설계단계의 프로그램 복잡성(설계 복잡성)의 적용도 연구되고 있다. 이러한 연구는 소프트웨어 설계 평가 모델에서 다루어지고 있다[1].

심리적 복잡성은 프로그램의 작성이나 이해가 난해한 경우 사람에 의해 주관적인 경험을 통해서 얻는



(그림 1) 소프트웨어 복잡성의 분류



(그림 2) 소프트웨어 복잡성 정의

복잡성이다. 이런 소프트웨어 프로세스(process)의 성질을 반영하는 데 있어서 프로세스 특성 요인의 관계를 정의하는데, 이 특성 요인으로는 개발요원, 개발기법, 사양변경에 의한 소프트웨어 개발을 특징지을 수 있는 요인이 존재한다. 즉, 심리적 복잡성은 여러 소프트웨어의 구현에 관계되는 시간이나 비용으로 표현되는 복잡성이다. 이것을 정리하면 (그림 1)과 같다.

◇ 복잡성의 정의

S/W 복잡성의 정의에 대해 Curtis는 이론적 복잡성과 심리적 복잡성의 이원화된 복잡성을 통일시켜 소프트웨어 복잡성에 대해 계측된 정의를 제안하고 있는데, 여기서 소프트웨어 복잡은 "소프트웨어 자신뿐만 아니라 다른 시스템과의 상호 작용의 관계"로 정의하고 있다[2].

Curtis 정의에 따르면 2개의 복잡성을 포함하는 소프트웨어의 복잡성은 소프트웨어 자신의 성질을 계수로 나타내는 이론적 복잡성과 최종 가능한 형태의 소스 비용을 가미한 프로그램 복잡성을 취급시 소프트웨어와 다른 시스템과의 상호 작용을 함수표로 정리한 심리적 복잡성으로 구성된다.

이상의 결과를 종합하면 (그림 2)와 같이 정리할 수 있다.

여기서 정의한 프로그램 복잡성은 소스 비용으로부터 계측된 형식적 구조의 복잡성에 대해 특성을 정리한 것이다.

본 논문의 구성은 I장에서 소프트웨어 복잡성에 대해 살펴보고 II장에서 VI장까지는 프로그램 복잡성, 제어구조, 데이터구조, 복합 메트릭스, 프로세스 복잡도에 의한 여러 가지 메트릭스 분석기법에 대해 알아본다. 그리고 마지막에 프로그램 복잡도에 근거한 방법을 채택하여 개발중인 ATM 교환기의 프로젝트에 적용한 결과를 분석하고 향후 고려할 사항에 대해 언급하고 맺는다.

II. Program Complexity Metrics

프로그램의 복잡성을 수치로 표현하는 경우 프로그램의 신뢰성이나 보수의 용이성을 표시하는 지표와 이용을 목적으로 한다. 프로그램 복잡성의 측정치를 프로그램의 신뢰성이나 유지보수성의 지표로 이용하는 데 있어서 프로그램 복잡성을 계측할 수 있는 메트릭스를 제공할 때 프로그램의 신뢰성이나 보수성과의 관계를 정량적으로 명확히 밝힐 수 있어야 한다. 이런 입장으로부터 프로그램의 복잡성 메트릭스에 대하여 McCabe의 척도나 Halsted의 척도 등과 같이 많은 연구가 수행되고 있다.

대표적인 예로는 Fitzsimmons & Love[3]에 의해 Halsted의 소프트웨어 공학이론에 입각한 실험적 평가, Curtis *et al.*[4]에 의한 McCabe 척도에 대한 통계적 수법을 이용한 평가, 春原[5]에 의한 실용시스템의 데이터에 기초한 분석 예가 있다. 여기서 프로그램의 복잡성 메트릭스에 대해서는 여러

가지 시도가 있었는데, 현재 소프트웨어 공학의 관점에서 중요한 일 중의 하나로 인식되고 있으며, 해결된 문제가 많고 표준 메트릭스로 확립해 나가고 있다. 대표적인 프로그램 복잡성 메트릭스에 대해 설명하면 다음과 같다.

◇ Volume metrics

Volume metrics는 프로그램의 크기에 착안한 척도로 이 경우의 메트릭스는 프로그램 소스코드의 행수(스탭 수, LOC, 스탭 멘트 수 등), Halsted의 소프트웨어 science 이론, 기능 수(function point 수)가 대표적이다. 이런 메트릭스의 정의는 소프트웨어 견적 모델로 정의되는데, 소프트웨어 관리 관점에서의 적용 상의 문제점을 중심으로 기술하면 LOC (Line of Code)는 소프트웨어 science 이론에 의거한 기능 수와 프로그램의 주요 설계 메트릭스, 소프트웨어 관리상의 중요 지표의 기본이 되는 메트릭스가 있다. 즉, 생산성(단위 cost 당 생산성)이나 신뢰성(단위 생산량 당 에러 수)의 평가에 대한 생산량의 메트릭스 등이 있다. LOC는 가장 간단하게 사용되는 메트릭스나 한편으로는 프로그램의 크기로 복잡성을 표시하는 경우 프로그램의 질적인 특성을 반영하기가 어렵다. 프로그램 크기 자체만을 설계 메트릭스에 적용시 필요한 메트릭스로는 불충분하며, 다음과 같은 문제점이 있다.

1) 측정법의 음미가 필요

행, 스텝에 대한 선언문에 대한 용어가 소스 비용 레벨로 사용되는 데에 대한 의견의 일치가 필요한데 소스 비용에 대한 측정 대상에 대해 정하기가 용이치 않다. 소스 비용을 차지하는 구성 요소로는 데이터 선언행, 실행 명령행, 매크로 명령행, 주역행(comment line) 등이 있는데, 동일 프로그램에 대한 주역행을 부정하는 경우 LOC의 측정치가 증가 또는 감소할 수 있다. 이러한 요소를 대상으로 평가하는 경우는 표준 해석이 없다. 프로젝트를 수행하는 조직에서는 이러한 요소들을 포함하는 것을 대체로 수용하고 있고, 일반적으로 실행 명령과 데이터 선언에 대한 측정을 많이 하며 이러한 평가는 크게 문제가 없다. 행, 여러 기계어에 대해서는 물리적으로 행이 긴 경우 1행에 대한 측정치에 대해 문제가 없으나 선언문에 대한 해석이 다른 경우가 발생되고 있다.

2) 언어 수준에 의존하는 경우

프로그래밍 언어 사용시 동일 기능의 프로그램을 작성하는 경우 수준이 높은 언어를 사용할 때는 낮은 언어를 사용하는 경우에 비해 기술되는 양이 적어진다. 이러한 경우 소프트웨어 생산성에 대한 문제가 대두되는데, 다시 말해서 생산성을 단위 비용당 생산량으로 정의시 수준이 높은 언어로의 LOC는 값이 적어지고 생산성이 낮아지는 현상이 발생된다. 이러한 문제는 생산성을 정의시 작업시간에 대한 고려로 해결할 수 있다.

3) 코딩 스타일에 의존하는 경우

언어의 수준이 동일한 경우 코딩의 방법에 따라 LOC의 값이 달라지는데 매크로나 서브루틴의 코딩 기법을 이용하는 경우에 코딩량이 적어지게 된다. 즉 언어수준에 의존하는 문제와 동일한 문제를 포함하고 있으며, 소프트웨어 science 이론에 의해 코드 레벨의 결과를 비교하는 방법으로 프로그램의 논리도나 기능사양의 레벨에 대한 사용법이 필요하며, operator와 operand의 경계를 정하는 어려움이 존재한다.

프로그램 논리도에 소프트웨어 science 이론을 적용한 예는 高橋[6]의 프로그램 논리도의 처리기호를 operator에 대응시켜 총수의 프로그램 크기를 표현했는데 이 이론은 소프트웨어 science 이론의 N_1(프로그램 중간의 operator 총 출현 수) 메트릭스에 해당된다. 기능 수는 소프트웨어 개발 초기에 단계적으로 프로그램의 기능사양 레벨에 소프트웨어 science 이론을 응용한 것인데 특정의 환경에 대해 operator와 operand가 검출되는 비율을 토대로 경험적 사실에 비추어 평가하는 N_2(프로그램 중간에 총 출현한 operand 수) 메트릭스를 사용하여 프로

그램의 크기를 나타내는 기법이다[7].

사무처리 분야의 프로그램은 기능사양에 의존하는 데이터 프로그램의 외부로부터 발견되는 입, 출력 항목이나 입, 출력 포트 수가 존재한다. 프로그램의 기술언어나 코딩기법에 대한 개발 프로세스의 영향을 받는 것은 어렵지만 기능 수는 LOC에 내재하는 문제점을 해결하는 하나의 방법이다. 그러나 프로그램의 질적 특성을 평가하는 문제는 조정계수의 보정을 계측자가 수행하는 데 주관적일 수밖에 없으며, 또 다른 문제를 야기시킨다. 프로그램 데이터의 처리에 데이터의 수가 직접 관여하는 방법에 대한 연구를 다수 수행하고 있으며, 개선책이 제안되어 계속 발표되고 있다[8],[9]. 예를 들면 宮崎[10]는 데이터를 정적/동적 수를 적용하여 프로그램의 크기를 정확히 표현하고자 하는 노력을 하고 있다.

III. 제어구조 메트릭스

제어구조 메트릭스(control structure metrics)는 프로그램의 제어 구조에 착안하여 가장 활발히 제안되고 있는 메트릭스로 대표적인 것은 McCabe cyclomatic number[4], chen의 최대 교차 수(Maximal Intersect Number: MIN)[11], woodward의 knot count[12], zolnowski[13]의 depth of nesting 등이 있다.

1. Cyclomatic Number

프로그램 제어의 순서에 따라 표시되는 그래프로 프로그램의 복잡성을 표시하는 방법으로 프로그램에 대한 하나의 입력과 출력에 대해 순방향 그래프에 대응시킨다. 이때 노드(node)는 제어가 수행되는 방향의 연속으로서 또 그래프의 분기(edge)는 제어의 분기를 표시한다. 일반적으로 유한방향의 강 결합(strongly connected) 즉, 절점으로부터 타 임의의 절점에 도달이 가능한 것으로 판단한다. 이 그래프를 포함한 1차 독립 폐회로(linearly independent circuits)의 수 V(G)는

$V(G) = e - n + 1$, (e: edge, n: node)

로 표시된다. 또 2차 입력에 대한 복수개의 출력이 있는 경우는

$V(G) = V(G_0) + E$

E: 2차 입력 수,

$V(G_0)$: 복수 출력이 없는 프로그램의 cyclomatic 수

로 표현된다.

2. 복수 출력이 있는 프로그램

복수개의 출력이 있는 프로그램은 각각의 출력에 1개의 절점에 대응시 END 문에 대한 출력의 분기로 가정하는데 이때 cyclomatic 수 V(G)는

$V(G) = V(G_0)$ 로 정의된다.

3. GO TO 문이 있는 프로그램

GO TO 문은 절점에 분기를 변경하는 경우 새로운 분기 추가가 안되며, GO TO 문을 삭제하는 경우에 고려되어야 할 프로그램 G_0의 cyclomatic 수 $V(G_0)$는 같아진다. 즉, $V(G) = V(G_0)$로 정의된다.

4. 내부 연결 프로그램

여러 개의 프로그램이 내부로 연결되어 있고, 각각은 외부와 독립적일 때 cyclomatic 수는 (1)로 표현된다.

$$V(G) = V(G_0 + G_1 + ... + G_m)$$

$$= \sum_{i=1}^{m+1}(\pi_{i-1} + 1) = \sum_{i=1}^{m+1}\pi_{i-1} + m + 1 \quad (1)$$

이것은 다시 (2)와 같이 정리된다.

$$V(G) = \pi + P = V(G_0) + P - 1 \quad (2)$$

P: Procedure 문 수,

$V(G_0)$: 내부 연결 존재시 프로그램 G_0 cyclomatics 수

가. 최대 교차 수

최대 교차 수도 cyclomatic 수와 동일한 프로그램의 흐름도(flow graph)를 이용한 메트릭스의 하나이다. 프로그램 그래프의 표현에서 그래프에 대한 영역을 1회 통과하는 연속선을 추출할 때 그 영역을 형성하는 분기에 대해 교차하는 수를 말한다.

나. 노드 수

GO TO 문의 사용은 제어의 연속성을 저해하고 프로그램을 복잡하게 한다는 연구결과가 있으며[14], GO TO 문이 많은 프로그램이 반드시 복잡하지만은 않다는 보고도 있다. 또 if-then-else 구조 실현시 프로그램을 구조화하는 데 있어서의 GO TO 문은 복잡하지 않다는 보고가 있다[15]. 여기서는 단순한 프로그램 실현시 GO TO 문의 수를 측정할 때 복잡 메트릭스를 충분히 고려하지 못한 것을 의미한다. 노드 수[12]는 이러한 방법을 고려하여 비 구조적인 방법을 사용하는 경우의 GO TO 문에 착안한 메트릭스이다. 노드 수 K는 (3)으로 정리된다.

$$K = \frac{\sum_{i=1}^{n} e_i}{2}, (n : GOTOStatement) \qquad (3)$$

다. Depth of Nesting

분기나 루프의 net는 매우 필요한 프로그래밍이다. 분기나 루프의 조건식을 극단적으로 복잡하게 하는 것을 피하는 경우 프로그램의 이해성은 향상된다. net 레벨을 극단적으로 단순화 시키는 경우에 특정 state에 도달하고자 하는 조건을 이해하지 못하면 프로그램의 이해성을 저해한다는 보고가 있다[16].

프로그램의 단일조건에 대한 분기나 루프의 경우 직관적으로 직렬인 경우보다 병렬인 경우가 더 복잡해진다. 즉, cyclomatic 수나 분기 수에 기초한 메트릭스나 노드 수에 대한 비 구조적 프로그램을 대상으로 한 메트릭스는 적절한 평가가 어렵다. 이러한 문제를 고려하여 제안한 메트릭스로 depth of nesting은 프로그램 구현시 net 군에 속한 nesting의 deaph에 최대치나 평균치를 측정하는 방법이다. 일반적으로 실행문의 정규화에 대해 평균 net 레벨(Average Nesting Level: ANL)을 사용하고 있다. ANL은 아래와 같은 절차에 의해 구해진다.

① 최초 실행문의 net 레벨에 대해 1개씩 분리한다.
② net 레벨 n의 실행문 a가 있을 때 실행문 b가 실행문 a의 뒤를 이어 실행되는 경우 실행문 b의 net 레벨은 실행문 a와 같은 조건이다.
③ net 레벨 n의 실행문 a가 있을 때 실행문 b가 실행문 a에 종속된 루프에 분기가 포함되어야 하며, 실행문 b의 net 레벨은 n+ 1이 되어야 한다.
④ 평균 net 레벨 ANL은 (4)로 구해진다.

$$ANL = \frac{\sum_{i=1}^{S} NL_i}{S} \qquad (4)$$

(NL_i: 실행문 I의 net 레벨, S: 실행문 수)

IV. 데이터 구조 메트릭스

프로그램 내의 데이터량은 프로그램의 복잡성을 나타내는 주요 요인이다. 프로그램에서 취급하는 데이터량이 많음은 프로그램의 복잡성을 증가시킨다는 것을 경험적으로 알 수 있다. 데이터량이나 참조 특성을 계측하는 메트릭스를 데이터 구조 메트릭스(data structure metrics)라고 부른다.

1. 데이터량

프로그램 구현시 데이터량을 계측하는 방법으로 변수의 종류 수, operand 종류 수, operand의 총 출현 수 등 3가지로 나눈다.

2. 변수의 활성 구간

프로그램의 변수 참조의 흐름에 착안하여 변수의 상태를 인식하고 각 구간에 대한 인식을 행할 때 그

구간이 길어지면 프로그램의 작성이나 이해가 곤란해진다. 변수의 활성 구간에 대한 정의가 가능하고 최초에 참조한 문번호와 마지막으로 참조한 문번호의 정적인 위치의 차를 정의하면 간단해진다. 변수의 활성구간을 포함한 기초이론을 바탕으로 변수의 활성거리와 활성 변수의 수 등 2개의 메트릭스로 정의한다.

3. 변수의 참조간격

동일한 변수에 대해 임의로 2개의 참조 간격을 포함하는 경우의 실행문의 수를 정의한다. 즉, 변수의 참조 간격이 단축되면 변수의 사용빈도가 높아지고 있음을 의미한다.

V. 복합 메트릭스

프로그램의 제어구조나 데이터 구조의 복잡성 메트릭스는 프로그램 일부에 대한 복잡성을 평가하는데 반해 복합 메트릭스는 프로그램의 복잡성을 전체적으로 평가하는 데 이용되는 메트릭스이다. 다시 말해서 프로그램의 복잡성을 총체적으로 계측하기 위해서는 특정한 단일 메트릭스로 측정하기는 곤란하다. 즉 단순 접근 방식으로는 간단한 메트릭스를 사용해도 프로그램마다 다르게 측정되므로 이러한 문제를 총체적으로 다룬 것이 복합 메트릭스이다.

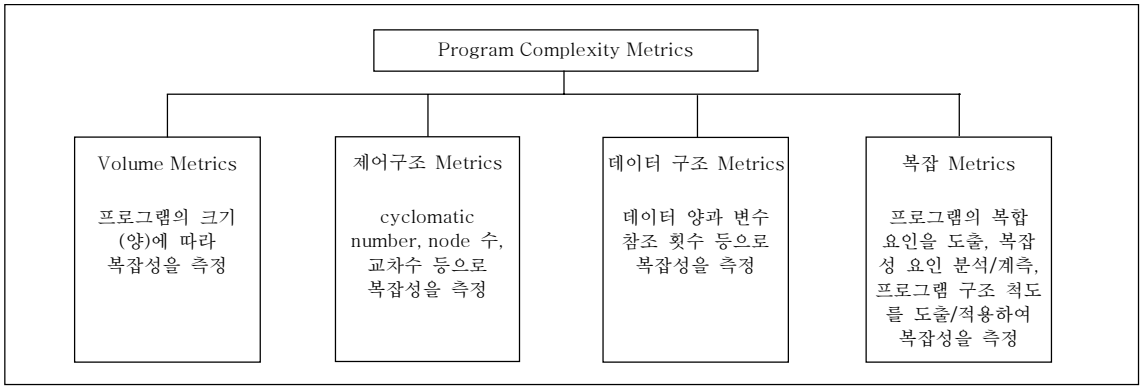
Baker와 zweben[17]은 소프트웨어 science 이론의 E-metrics와 cyclomatics 수에 대한 복합 메트릭스를 제안하여 소프트웨어 복잡성을 평가하였다. 또 프로그램의 복잡성 요인을 여러 가지 측면으로 고려한 방법으로 花田[18],[19]의 프로그램 구조척도 방법이 있다.

복합 메트릭스에 의한 방법은 프로그램의 복잡성 요인의 도출, 프로그램의 복잡성 요인의 계측, 프로그램의 복잡성 요인의 분석, 프로그램 구조척도의 도출, 프로그램 구조척도의 적용 순서로 다양하게 평가할 수 있다.

복잡성 요인은 프로그램 구조의 데이터 구조, 데이터 참조, 처리, 제어구조, 인터페이스 등 5가지로 구분하여 도출하고 복잡성 요인에 대한 계측은 언어요소의 출현 수를 계측, 언어요소의 출현 유/무 계측, 언어요소의 출현 상태를 수치화하는 경우로 구분, 평가할 수 있다.

복잡성 요인의 분석은 실행문수의 정규화, 소규모 프로그램의 제거, 복잡성 요인의 출현 특성(분포)을 통계화, 분산분석에 의한 복잡성 요인의 검정 수행을 수행한다.

프로그램의 구조척도의 도출은 V(G)와 실행문수, E와 실행문수와의 관계, V(G)와 에러율의 관계, E와 에러율과의 관계 등을 도출하고 그 적용 방법은 복잡성 메트릭스(C_i)와 에러율과의 관계나 C_i의 벡터 표현 등을 적용해 소프트웨어의 복잡성을 평가하는 방법이다. 위에서 언급한 복잡성 메트릭스 기법을 분류, 정리해 보면 (그림 3)과 같다.



(그림 3) 프로그램 복잡성 Metrics 분류

VI. Process Complexity Metrics

프로세스 복잡성 메트릭스 기법은 프로세스 복잡성 요인과 프로세스 복잡성 요인 계측법으로 구분되며, 이것은 각 요인에 대한 5가지 특성에 따라 분류되어 계측된다.

1. 프로세스 복잡성 요인

개발요원(개발자), 리소스, 개발기법, 툴, 발주자와의 인터페이스, 산출물에 대한 인터페이스로 구분된다. 이것들은 5가지 특성에 따라 분류되고 각각의 특성에 따라 세분화되어 계측된다.

1) 개발요원 특성

개발자의 경험과 팀 구성, 개발요원의 변동(리소스 할당 문제와 연계) 등으로 분류되며, 숙련도 및 팀 구성의 숙련자 비율, 개발요원 변동률을 파악, 측정한다. 즉, (5)와 같이

$$S = \frac{\sum_{i=1}^{n} p_i}{developer} \tag{5}$$

S: 숙련도, P_i: 개발요원 프로그램 경력(년)
T(숙련자율) = 숙련자 수/개발자 수
P(개발요원 변동률) = (최대가동 인원수 - 최소가동 인원수)/최대가동 인원수로 표현된다.

2) 리소스 특성

개발 공수 배분 및 개발기간 제약성에 대해 평가한다.

3) 개발기법, 툴 특성

설계기법 및 도큐먼트 특성, 코딩 기법(프로그래밍 언어), 컴퓨터 사용 환경에 대한 특성으로 구분, 측정한다.

4) 발주자 인터페이스 특성

발주자 인터페이스는 주로 사양 변경률에 좌우되며, 기능설계 변경률과 개발규모(KLOC)로 측정한다.

$$S_c = \frac{D_c}{K} \tag{6}$$

S_c: 사양 변경률, D_c: 기능설계서 변경량
K: 개발규모(KLOC)

5) 산출물 인터페이스(product interface)

도큐먼트 품질, 프로그램 개조 및 재 사용 비율로 측정되며, 도큐먼트 비율은 도큐먼트 작성량과 개발규모(KLOC)로 측정된다.

$$D = \frac{D_p}{K} \tag{7}$$

D: Document rate, D_p: Document 작성량,
K: 개발규모(KLOC)

2. 프로그램 개조 및 프로그램 재 이용

프로그램 개조는 프로그램 전체 에러율과 밀접한 관계를 갖으며, 개조부문의 규모에 대한 신규 작성부문으로 측정되고 프로그램의 재 이용은 두 가지 측면으로 분류되어 측정할 수 있다.

첫번째는 재 이용된 모듈로부터 재 이용 비율을 측정해 보는 시각과 재 이용한 프로그램으로 보는 시각이다. 이에 대한 연구결과는 1991년에 Galdier, G and Basli, V. R이 모듈의 재 이용 횟수로 프로그램의 재 이용에 대한 연구결과를 발표한 바 있다[20].

○ 프로그램 개조율(P_m)

$$P_m = \frac{K_m}{K_n + K_m} \tag{8}$$

K_m: 개조부분(KLOC), K_n: 신규작성(KLOC)

○ 프로그램 전체 에러율(r)

$$r = \frac{K_n \gamma_n + K_m \alpha \gamma_n}{K_n + K_m}, (\alpha > 1) \tag{9}$$

a: 개조모듈 에러율/신규작성 에러율

r_n: K_n의 에러율

$$r = r_n P_m(\alpha - 1) + r_n$$

여기서 a 의 값은 매우 적은 값을 의미한다. 또, 유용부문이 있는 프로그램 개조율($P_{m'}$)은 (10)과 같이 정리된다.

$$P_{m'} = \frac{K_n + K_m}{K_n + K_m + K_0} \quad (10)$$

K_0: 유용부분 규모(KLOC)

○ 프로그램 재 이용률(P_r)

$$P_r = \frac{K_r}{K_n + K_m + K_r} \quad (11)$$

K_r: 재 이용부분 규모(KLOC)

이와 같이 표시되며, 이러한 분석 방법은 개발하고 있는 시스템의 분석에 매우 중요하나 너무 복잡하고 실제 프로젝트 분석 적용에는 많은 어려움이 따른다. 본 논문에서는 프로젝트에 간단하게 적용해 볼 수 있는 프로그램 양에 대한 분석과 프로그램 개조 및 재 이용에 대한 분석 결과, 그밖에 이를 수행하는 과정에서 발생되고 있는 초기 문제점을 분석해 보는 방법을 택한다.

VII. 교환 소프트웨어 분석 결과

1. ACE2000 시스템의 프로그램 분석

전체 시스템의 규모는 6개의 서브시스템(ASS, CCS, MGS, AMS, MSS, ATS)과 11개의 모듈로 구성되어 있으며, 링크 당 정합속도는 2.5G로 최대 처리 용량이 160G인 대형 ATM 교환 시스템이다. 시스템을 제어하는 OS 및 각종 기능을 담당하는 응용프로그램에 대한 전체 상황은 <표 1>과 같다.

VI장의 프로세스 복잡성 요인 중 개발요원 특성에 대한 분석 결과는 각 기능별로 프로그램 숙련도가 <표 2>와 같으며, 개발요원 변동률은 대략 공동개발업체 인력을 포함하여 최고 33%, 최저 15%(최대 가동인원 60[공동개발기업체 인원 제외시 47]명, 최소 가동인원 40명)로 밝혀졌다. 그 외에 프로그램 숙련자율은 53%로 파악되었다(숙련자는 교환기 프로그램 경력 10년 이상자를 기준으로 함).

<표 1> ACE2000 시스템 규모

구분	블록 수	기능 수	규모(KLOC)
H/W	28	-	-
S/W	121	147	1053

* 총 소프트웨어 개발참여 인원은 40명

시스템 설계 기법은 UML(Unified Modeling Language) 설계툴을 이용한 객체지향설계(object oriented design) 기법을 도입하고 다양한 운영환경을 통합하고 일관된 동작 규격으로 통신하기 위한 소프트웨어인 미들웨어(middleware)를 이용한 IDL (Interface Definition Language) 코딩기법을 활용하였으며, 사용 언어는 C, C++, Java 언어를 이용하고 UNIX 환경과 웹 기반의 개발 환경을 이용하였다. 프로그램 재 사용 관련 프로그램 개조율과 재 이용률은 분석에 많은 어려움이 수반되나 그 결과는 <표 3>과 같다.

<표 2> 개발요원 특성 분석 결과 (단위: 년)

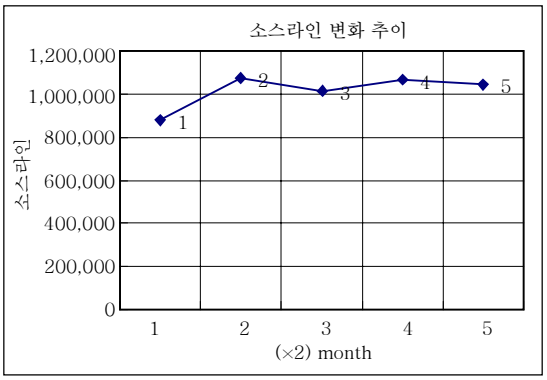
기능별	경험치(AVG.)	담당블록 수
호 제어(CP)	11.3	5.6
운용보전(O&M)	7.3	6.2
운영체제	7.3	1.0
데이터베이스 시스템(DBMS)	5.0	1.25
망관리(TMN)	2.5	1.0
펌웨어(DC)	8.2	1.0

* DC: Device Controller(장치 제어계)

<표 3> 프로그램 개조 및 재 이용 현황

구분	비율(%)
개조율	17.2
재 이용률	16.3

(그림 4)는 교환 소프트웨어에 대한 프로그램 재 사용 및 개조율을 구하기 위한 프로그램 소스 변화이다. 그림에서 기준이 되는 시간(x축 값)은 2001년 1월을 기준으로 매 2개월씩 변화된 것을 나타낸 것이다. 4월 기업체 배포 이후 2001년 8월 말 현재 약 105만 라인정도로 개발되고 있다. 변동률은 5~18% 정도로 증가되는 반면 줄어드는 경우도 있는데 이는 소프트웨어의 성능을 높이고 중복사용을 줄이는 한편 시스템에서 공통으로 사용할 수 있도록 컴포넌트들을 라이브러리화(공통 라이브러리)한 덕분에 소스라인 규모가 일부 줄어들거나 재 사용 및 개조에 의한 현상이다.

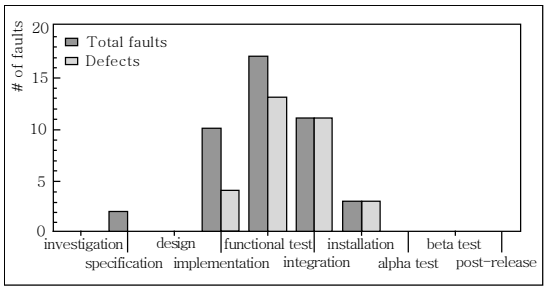


(그림 4) 소프트웨어 소스라인 수 변화

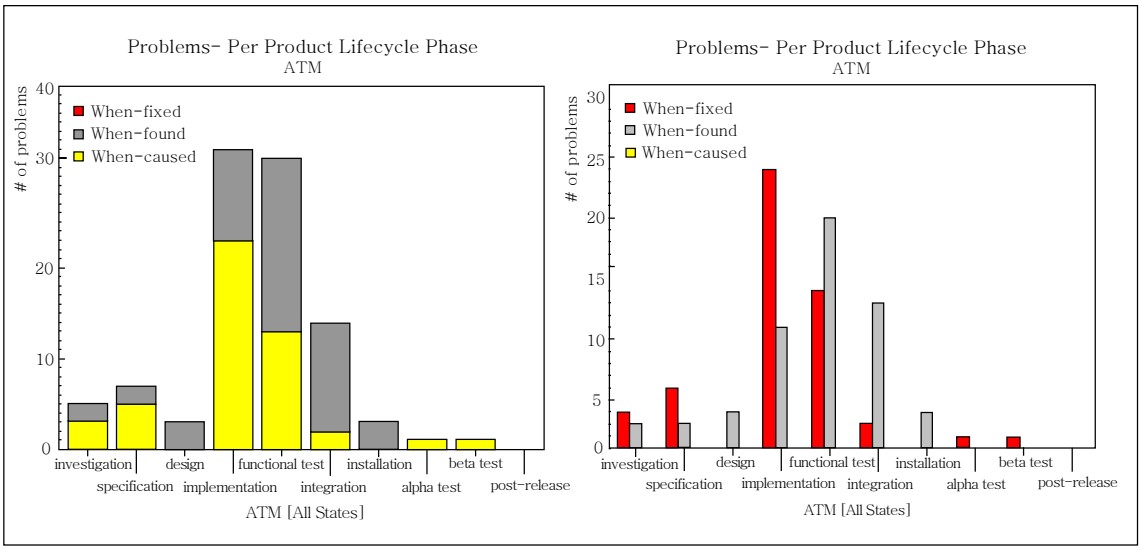
2. 소스 프로그램 문제점 분석 결과

(그림 5)는 시스템의 각 기능 범주별로 분석된 결함 데이터이다. 이 데이터는 개발초기단계를 거쳐 기능의 구현을 마무리하는 인테그레이션 단계로 대다수의 결함들이 기능구현 및 기능시험, 통합시험 시 발생하는 결함이 80% 이상 차지하고 있음을 알 수 있다.

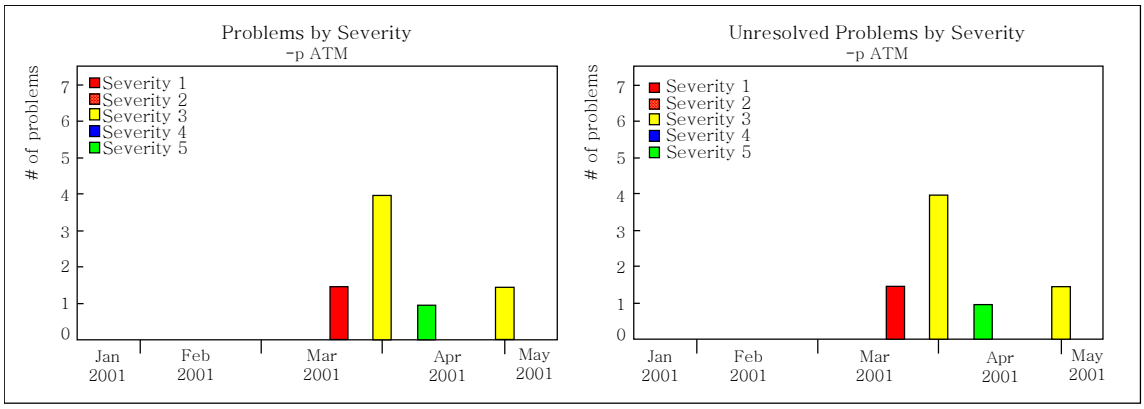
프로젝트 생명 주기에 따른 발견/원인/조치별 데이터로 개발 중간 단계인 구현 및 기능시험, 통합 단계의 데이터가 가장 많이 차지하고 있다. 이 데이터를 참조하면 프로젝트의 진행 상황을 파악할 수 있는 유용한 자료로 활용될 수 있다. 제시된 데이터에 의하면 기능시험을 마무리하고 통합단계로 접어들고 있음을 알 수 있다((그림 6)의 오른편 그림 해결 추세 참조).



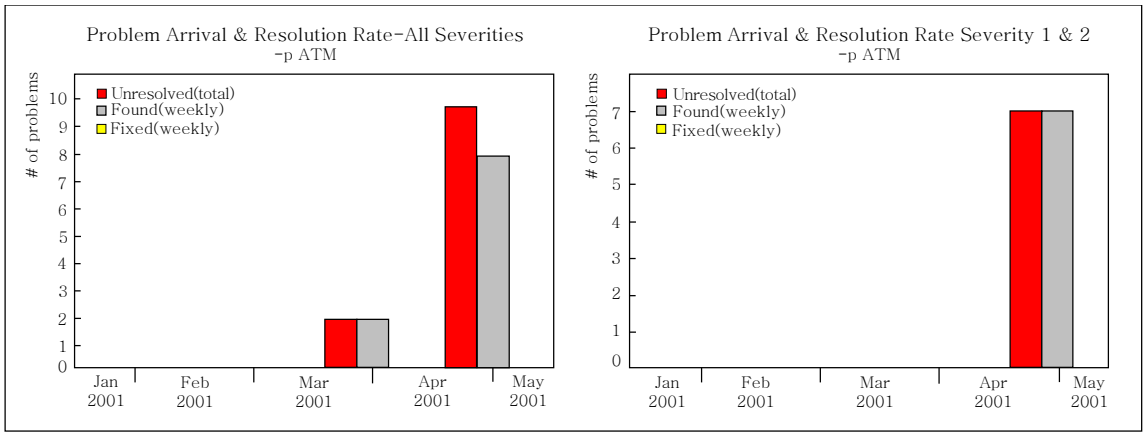
(그림 5) 프로젝트 생명 주기별 고장 현황



(그림 6) 프로젝트 생명 주기별 고장발견/원인/조치별 현황



(그림 7) 심각성에 의한 월별 문제점 제기 및 미해결 상황

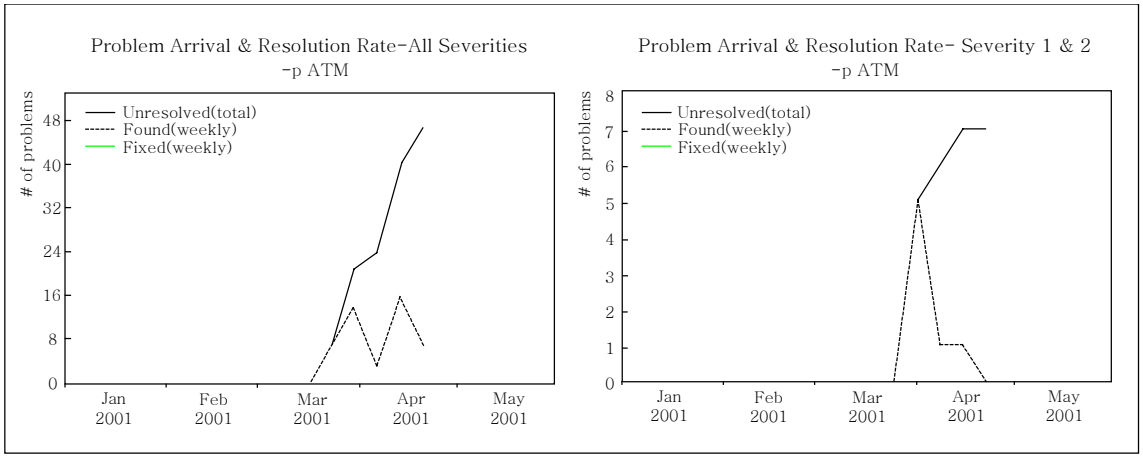


(그림 8) 주별 문제점 처리 현황

문제 발생 원인은 구현단계에서 가장 많이 발생하고 있는데, 그 원인은 기존에 시스템 개발에 사용되던 CHILL 언어에서 C/C++/JAVA로 언어의 변화에 의한 개발환경의 변화와 미들웨어를 이용한 새로운 개념의 기능 구현방법을 시도하여 이에 대한 충분한 경험과 각 툴에 대한 검증이 미흡함으로 인한 원인으로 추정된다.

(그림 7)은 발견된 문제점과 미해결 문제점에 대한 월별 분석 결과로 발생된 문제점에 대해 심각성을 1~5단계(severity 1~5)로 구분하여 단계가 높을수록 시스템에 미치는 영향이 심각함을 나타낸다. 즉 운용상에 별 지장이 없는 minor한 문제부터 5단계의 critical한 문제로 중간 레벨의 문제가 대다수이며, 의외로 심각한 문제도 상당수 차지하고 있다. 이 원인은 시스템간 통신을 위한 IPC 상태가 불안하여 발생한 문제로 이를 지원하는 하드웨어와 제어용 운영체제의 안정성, 미들웨어 통신에 많은 문제가 발생한 것으로 파악되었다.

(그림 8)의 문제점 제기와 해결 내역에 대한 분석 결과는 대체로 사소한 문제점이 많이 발견, 해결되는 추세이며, 전체 문제점의 심각성에 대한 해결과 미해결 문제점에 대한 분석 결과는 대등한 상태로 발견과 즉시 해결되는 경향을 띤다. 이러한 경향은 개발이 한참 진행되고 있음을 보여주고 있으며, 이러한 경향은 개발이 완료되어 사용자에게 release 되는 시기에 도달하면 제기되는 문제점이 적고 심각성이 높아지는 경향을 보인다. 즉, 배포 단계(beta test & post-release)로 갈수록 유지보수 비용의



(그림 9) 주별 문제점 발견 추이와 전체 미해결 현황

증가와 발견된 결함 데이터의 수정(bug-fix)이 어려워지기 때문이다.

(그림 9)는 발생된 전체 문제점에 대한 주별 발견 추이와 미해결에 대한 전체 상태를 보여주고 있다. 오른편 그림은 운용상에 크게 지장이 없는 문제점에 대한 발견과 미처리 현황이다. 특정 기간에 사소한 문제점들이 많이 발생되고 있는 현황은 주별 시험용 패키지 제작에 따른 시험자원의 집중 투입에 따른 현상과 개발 계획에 따른 신규 기능추가를 의미한다. 즉, 발견된 문제점에 대한 처리 속도가 느리고 새로운 기능의 추가가 한창 일어나고 있는 시기임을 그래프를 통해 알 수 있다. 다시 말해서 프로젝트의 전 생명주기 중 설계단계(design)를 지나서 구현단계(implementation)에 이르고 있음을 의미한다.

VIII. 결 론

교환 소프트웨어 개발관리에 주로 사용되어 왔던 방법은 주로 개발된 프로그램을 동적인 환경에서 실행시켜 얻어진 결과를 토대로 소프트웨어 개발프로젝트를 평가하는 소프트웨어 신뢰도성장모델을 사용하여 왔다. 이 방법은 실제 프로젝트 적용시 간단하게 적용이 가능하여 널리 이용되어 왔다. 그러나 기존에 사용된 방법은 개발된 프로그램에 대한 분석이 용이치 않아 프로젝트 관리방법에는 잘 활용되지 못했다.

본 논문은 이러한 단점을 보완하고 개발된 프로그램에 대한 정적 분석과 동적 분석을 병행하여 소프트웨어 품질을 향상시키고 소프트웨어에 대한 신뢰도를 향상시킬 수 있는 방법의 일환으로 교환시스템에 대한 소스 프로그램을 복잡도 모델의 기초 이론에 입각하여 프로그램의 규모를 분석함으로써, 교환 소프트웨어의 품질 척도는 물론 개발 진행중에 발생되고 있는 문제점들을 다각도로 분석하여 소프트웨어 품질 척도와 신뢰도를 향상시킬 수 있는 틀을 만들었다. 이러한 분석 정보들은 프로젝트 관리 즉, 전체 개발 진행 현황 파악을 위한 유용한 정보로 활용이 가능하다.

향후 연구방향으로는 다양한 품질 척도 방법과 프로젝트 견적 모델인 개발비용을 기초로 한 연구생산성 평가 모델과 연계하여 최소의 비용으로 최적의 소프트웨어를 생산해 낼 수 있는 모델 개발과 이에 대한 적용방법의 연구가 필요하다.

약 어 정 리

ASS : ATM Switching Subsystem
CSS : Central Server Subsystem
MGS : Media Gateway Subsystem
AMS : Access Multiplex Subsystem
ATS : AAL-2 Trunking(Switching) Subsystem
MSS : MPLS Service Subsystem

참 고 문 헌

[1] 山田 戌, 高喬 宗雄, ソフトウェアマネジメントモデル入門 – ソフトウェア品質の可視化と評價法, 共立出版社, 1993, pp. 25 – 46.

[2] B. Curtis, "Measurement and Experimentation in Software Engineering," *Proc. IEEE*, Vol. 68, No. 9, 1980, pp. 1144 – 1147.

[3] A. Fitzsimmons and T. Love, "A Review and Evolution of Software Science," *ACM Computing surveys*, Vol. 10, No. 1, 1978, pp. 3 – 18.

[4] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halsted and McCabe Metrics," *IEEE Trans. Software Engineering*, Vol. SE-5, No. 2, 1979, pp. 96 – 104.

[5] T. Sunohara, A. Takano, K. Uehara, and T. Ohkawa, "Program Complexity Measure for Software Development Management," *Proc. 5th Int. Conf. Software Engineering*, 1981, pp. 100 – 106.

[6] M. Takahashi, T. Miyake, and S. Hanata, "Statistically-based Program Size Estimation," *Proc. COMPSAC 89*, 1989, pp. 574 – 579.

[7] A.J. Albrecht, "Measuring Application Development Productivity," *Proc. joint SHARE/GUIDE symp.*, 1979, pp. 83 – 92.

[8] C.R. Symons, "Function Point Analysis: Difficulties and Improvements," *IEEE Trans. Software Engineering*, Vol. 14, No. 1, 1988, pp. 2 – 10.

[9] J.M. Verner, G. Tate, B. Jackson, and R.G. Hayward, "Technology Dependence in Function Point Analysis: A Case Study and Critical Review," *Proc. 11th Int. Conf. Software Engineering*, 1989, pp. 375 – 382.

[10] 宮崎幸生, 山田松治, 倉捻, "段階的規模見積りモテルの概念とその作成方法," 情報處理學會論文誌, Vol. 32, No. 2, 1991, pp. 140 – 148.

[11] E.T. Chen, "Program Complexity and Programmer Productivity," *IEEE Trans. Software Engineering*, Vol. SE-4, No. 3, 1978, pp. 187 – 194.

[12] M.R. Woodword, M.A. Hennell, and D. Hedly, "A Measure of Control Flow Complexity in Program Text," *IEEE Trans. Software Engineering*, Vol. SE-5, No. 1, 1979, pp. 45 – 50.

[13] J.C. Zolnowski, and D.B Simmons, "Taking the Measure of Program Complexity," *Proc. National Computer Conf.*, 1981, pp. 329 – 336.

[14] E.W. Dijkstra, "GO TO Statement Considered Harmful," *Communication of the ACM*, Vol. 11, No. 3, 1968, pp. 147 – 148.

[15] 花田收悅, 高橋宗雄, 永瀨淳夫, 黑田幸明, "プログラム構造と信賴性に關する分析," 情報處理學會論文誌, Vol. 23, No. 1, 1985, pp. 9 – 15.

[16] H.E. Dunsmore and J.D. Gannon, "Data Referencing: An Empirical Investigation," *IEEE Computer*, Vol. 12, No. 12, 1979, pp. 50 – 59.

[17] A.L. Baker and S.H. Zweben, "A Comparison of Measures of Control Flow Complexity," *IEEE Trans. Software Engineering*, Vol. SE-6, No. 6, 1980, pp. 506 – 512.

[18] 花田收悅, 高橋宗雄, 永瀨淳夫, 黑田幸明, "プログラム構造の複雜さ尺度の評價と導出法の提案," 情報處理學會論文誌, Vol. 23, No. 6, 1988, pp. 701 – 706.

[19] 花田收悅, 高橋宗雄, 永瀨淳夫, 黑田幸明, "プログラム構造の複雜さ要因分析に基づく尺度の導出法," 研實報, Vol. 32, No. 2, 1983, pp. 405 – 416.

[20] G. Galdier and V.R. Basili, "Identifying and Qualifying Reuseable Software Components," *IEEE Computer*, Vol. 24, No. 2, 1991, pp. 61 – 70.