정적 테스팅을 위한 소프트웨어 풀의 설계에 관한 연구

— COBOL을 중심으로 —

A Study on the Design of the Software Tool and COBOL Static Testing

이 종 철 *
신 양 호 **

Abstract

It is very important to assure the software quality. The static and dynamic testing are to be performed on the program to find some error in it, and it's purposes are to assure the software quality with cost-effectiveness. For it we use the automated tool.

In this paper, it suggest the design of the software tool for COBOL static testing.

I. 서 론

소프트웨어를 테스트한다는 것, 즉 검증(verification)과 확인(validation) 활동의 목표는 소프트웨어를 개발하고 변경하는 동안에 만들어진 생산제품의 질을 평가하고 향상시키는데 있다. 품질의 속성에는 정확성(correctness), 완전성(completeness),

I. 서 론

소프트웨어를 테스트 한다는 것, 즉 검증(verification)과 확인(validation) 활동의 목표는 소프트웨어를 개발하고 변경하는 동안에 만들어진 생산제품의 질을 평가하고 향상시키는데 있다. 품질의 속성에는 정확성(correctness), 완전성(completeness), 일관성(consistency), 유용성(usability), 효율성(efficiency), 표준화 준수 그리고 전체적인 비용-효과성(cost-effectiveness) 등이 있다.

검증에는 두 가지 형태 즉, 주기 검증(life-cycle verification)과 형식 검증(formal verification) 이 있으며 확인은 소프트웨어 개발과정의 종료시 소프트웨어가 요구사항에 맞게 제작되었는가를 결정하기 위하여 이를 평가하는 과정이다.[1]

보헴은 검증과 확인을 다음과 같이 정의하였다.[2]

- 검증(verification) : 제품을 올바르게 만들고 있는가?
- 확인(validation) : 올바른 제품을 만들고 있는가?

소프트웨어를 테스트하는 기법에는 여러가지의 기법이 있는데 크게 두 가지로 구분할 수 있다. 첫번째 정적테스팅으로 소프트웨어의 실제적인 실행을 동반하지 않는 방법으로 소프트웨어의 부분적 정당성(partical correctness)을 확인해가는 것이다. 두번째 동적테스팅은 설정은 되었으나 표본 데이터를 가지고 실지 프로그램을 실행시켜 프로그램의 행위(behavior)를 관찰함으로써 소오스코트의 실행통계, 제어흐름, 데이터의 흐름 및 데이터의 민감도등을 분석하는 기법이다.

본 논문에서는 정적테스팅 기법을 사용하여 DP(Data Processing) 분야에서 일반적으로 사용되는 COBOL 프로그램에 대해 DATA DIVISION 에서 정의된 식별자(identifier)가 PROCEDURE DI-

* 국제대학 전산통계학과 부교수
** 명지실업전문대학 전자계산과 조교수

1) IEEE Standard Glossary of Software Engineeing Terminology, IEEE Standard 729-1983.
2) Boehm, B.: The Hardware/Software cost Ratio : Is It a Myth? IEEE COMPUTER, Vol. 16, No. 3, March 1983.

VISION에서 사용된 경우와 사용되지 않는 경우를 검증하기 위한 툴(tool)의 구축 방법에 대해서 연구하였다.

Ⅱ. 본 론

2.1 정적 테스팅의 정의

정적 테스팅은 원시 코드의 구조적 특징, 설계 사양 혹은 잘 정의된 구문 법칙과 일치하는 임의의 표기법을 이용한 표현을 평가하는 기법이다. 즉, 요구(requirement)와 설계과정 및 관련문서류 그리고 소오스 프로그램의 논리적 일치성(consistancy), 완전성(completeness), 그리고 구조적 유효성 등의 정적인 상황에 대해 해석한다. 그 주된 목적은 1) 다양한 의미론적 그리고 변칙(anomalies)의 검출 검출

2) 프로그램으로부터 단순한 에러제거 및 더욱 상세한 해석을 위한 상황의 설정

3) 동적 테스팅의 표적이 될 수 있는 의문스런 성질들의 식별등이다.

정적 테스팅은 검토회나 검열 기법을 사용하여 수작업으로 실시할 수 있으나 정적 테스팅 또는 정적 분석이란 용어는 자동화 도구에 의한 프로그램의 구조 조사를 나타내는데 가장 많이 사용된다(RAM 75, MIL 75, FOS 76, SOF 80, GRC 83).[3] 정적 테스팅기는 전체 프로그램에 대한 호출 그래프뿐만 아니라 기호표(symbol table)과 각 서브프로그램에 대한 제어흐름 그래프를 만든다. 기호표는 각 변수에 관한 정보, 즉 이의 형, 속성, 선언한 문장, 새로운 값을 설정하는 문장, 그리고 값을 제공하기 위해서 사용되는 문장등에 관한 정보를 갖고 있다.

정적 테스팅에는 실제적인 한계와 이론적인 한계가 있는데, 실제적인 주요한계에는 실행시에 기억장소의 참조를 동적으로 평가하는 것이다. 고급 프로그래밍(High-level language)에서 배열첨수와 포인터 변수는 프로그램에 의해서 실행되기 이전의 계산을 기초로 한 동적인 기억·장소·참조·기능을 제공한다. 정적 분석기는 첨수나 포인터 값을 계산할 수 없기 때문에, 정적 분석 기법을 사용하여 리스트의 배열 원소 혹은 멤버 사이에서의 차이점을 구별하는 것은 불가능하다. 동적 테스팅 사례는 정적 분석 기법에 의해서 구해지기 어렵거나 구하기 불가능한 정보를 얻기 위해서 사용된다.

정적 분석의 주요 이론적인 한계는 가결정 이론(decidability theory)에 의해서 부가되는데, 가결정 결과는 여러가지 방법으로 문장화될 수 있으며, 매우 심오한 암시를 지니고 있다. 범용 프로그래밍 언어로 작성된 임의의 프로그램에 대해서 연산 방식으로 프로그램을 시험하는 것은 불가능하며, 프로그램내에 있는 임의의 문장이 그 프로그램이 임의로 선택된 입력 자료에 따라 작동될 때 실행할 것인가를 결정하는 것은 불가능하다.

따라서 가판단 이론은 임의의 프로그램과 임의의 입력 자료를 조사하며, 그 프로그램이 특정한 정지문(halt statement)을 실행하거나 특정 서브루틴의 호출, 특정 입·출력문의 호출, 혹은 주어진 레이블로의 분기 등을 결정하는 정적 분석기를 제작하는 것이 불가능하다는 것을 나타낸다. 그러므로 임의의 프로그램에 대해서 정적 분석을 실시하기란 불가능하며, 특정한 제어 경로를 따라 프로그램을 유도시키는 입력자료의 집합을 자동적으로 만드는 것은 불가능하다. 예를 들면 분기문이 없는 간단한 프로그램은 모든 실행(rerflow 등은 제외)시에 그 프로그램 내에 있는 모든 문장을 실행하는 것을 보장하며 정적 분석은 프로그램이 분기문을 지니고 있는 지를 결정하는데 사용할 수 있다.

정적 분석기에 의해서 제공되는 대표적인 항목들은 다음과 같다.

1) 제어 흐름 그래프(control flowgraph)
2) 변수에 대한 기호표(symbol table)
3) 호출 그래프(call graph)
4) 각 루틴에 전달되는 독립 변수
5) 초기화되지 않은 변수
6) 설정되었으나 사용되지 않는 변수
7) 분리된 코드 세그먼트
8) 코딩 표준의 위반 사항
9) COMMON 블럭에서의 정렬 에러(FORTRAN)
10) 서브프로그램 매개 변수의 오용
11) 라인의 총 수
12) 주석 라인의 총 수와 위치
13) 빈 라인의 총 수와 위치
14) 레이블과 GOTO 문의 수와 위치
15) 시스템 호출의 수와 위치
16) 문자 상수의 수와 위치

위와 같은 항목들의 목적을 달성하기 위해서는 거듭해서 소오스 정보(source information)를 탐색할 필요가 있다. 분석의 효율성을 높이기 위해서 대부분의 테스팅 시스템은 내부적인 데이터베이스로 대상 프로그램을 표현한다. 결과적으로 정적분위기는

3) Ramamoorthy, C., and S.HO: "Testing Large Software with Automated Software Evaluation system", IEEE Trans. Software Eng., Vol. SE-1, No. 1, January 1975.

통상 데이터베이스를 생성하는 입력 분석기(imput analyzer)와 구조적분석과 변칙성분 검출을 수행하는 라이브러리 루틴들로 구성되며[4] 그림-1과 같다.

그런데 요구라던가 설계에 관한 대부분 해석은 그다지 형식적이지 않기 때문에 자동화하기 어렵다. 이들을 형식화 혹은 자동화하는 것은 어느 정도의 명세 기술언어(specification description Language)와 분석기술이 개발될 수 있는가에 달려있다. ISDOS 프로젝트의 PSL/PSA[5]는 한 예의 시스템이다.

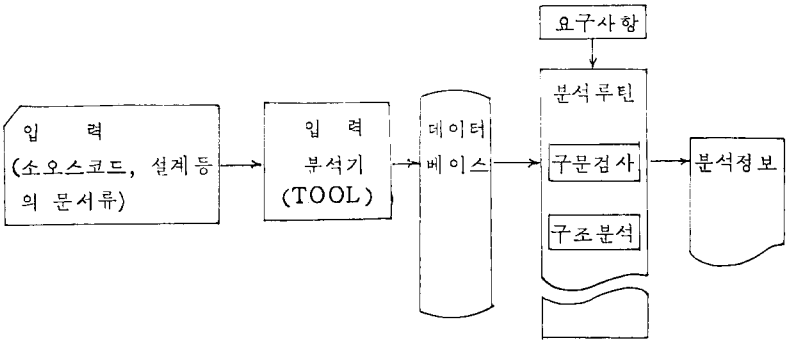

그림-정적 분석기(static Amalyzer)

2.2 정적 테스팅의 툴의 설계

정적 테스팅은 (혹은 분석) 일반적으로 프로그램의 원시 코드의 구조적 성질을 조사하는데 사용되며 다시 말해서 프로그램의 약단정(allegation)을 조사한다. 약단정이란 바람직한 프로그램 속성(갖추어야 할 프로그램의 자태)의 기술이며, 표명(assertion)보다는 약한 규칙이다. 약단정의 예로는 다음과 같은 규칙이다. 약단정의 예로는 다음과 같은 것들이 있다.

1) 배열변수의 정수형 첨자는 미리 정의된 배열의 차원을 벗어나지 않을 것

2) 각 변수는 참조 전에 값이 주어질 것(set-before-use) 규칙

3) 모든 변수는 명시적으로(explicitily)형 선언(type decralation)될 것

4) 식표현의 평가에서 암시적인(implicit)한 형변환을 행하지 않을것("="을 통한 형변환은 허용).

일반적으로 약단정이란 것은 프로그래밍 언어로는 기술 가능하지만 신뢰성 좋은 프로그래밍이란 관점에서는 허용되지 않는 그런 제 특성에 관해서 설정된다. 경우에 따라서는 프로그램의 의미론(semantics)에 관한 것도 있다.

그림-2는 정적 테스팅 툴(tool)의 처리절차를 나타내고 있다.

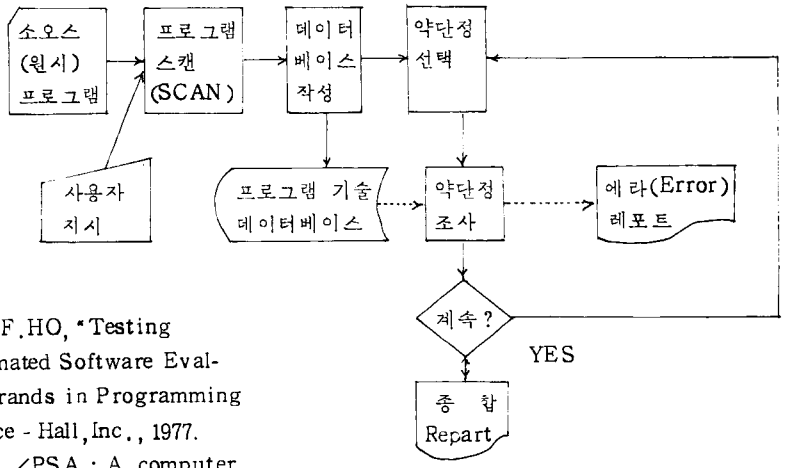

그림-2 정적 분석 툴의 처리절차

4) C.V.Ramamoorthy and S.F.HO, "Testing Large Software with Automated Software Evaluation Systems", Current Trands in Programming Methodology, Vol. II, Prentice-Hall, Inc., 1977.

5) D.Teichroew et al., "PSL/PSA : A computer-aided Technique for Structured Documentation and Analysis of Information Processing System", Proc. 2nd Inf. Conf. on SE., OCT. 1976.

프로그래머가 하나의 단위 테스팅을 실시할 수 있는 것은 다음과 같이 4가지 범주로 분류한다.

1) 기능 테스트(functional tests)
2) 강도 테스트(stress tests)
3) 성능 테스트(performance tests)
4) 구조 테스트(structure tests)

위와 같은 테스팅(또는 테스트)를 지원하기 위한 여러가지 툴이 이미 개발되었는데 이에는 SOFTOOL 패키지, RXV 980, NBS FORTRAN 77 그리고 TOOLPACK 등이 있으며[6] 이러한 툴들이 제공하는 정적 분석 분야의 기능은 다음과 같다.

1) 서브루틴 호출에서의 부정확한 매개 변수의 수
2) 실행되지 않는 코드 세그먼트
3) 루프 내에서 재 정의된 DO 루프변수
4) 서브프로그램 내에 있는 모의 매개 변수의 변경
5) 조건부로 실행된 식에서의 함수 참조
6) 참조되었으나 설정되지 않은 식별자(identifier)
7) 설정되었으나 참조되지 않은 식별자(identifier)
8) 참조되고 그 이후에 설정된 식별자
9) 시스템이 선언해준 식별자
10) 각 루틴을 호출하는 루틴리스트
11) 각 프로그램 단위에 대한 식별자(identifier) 색인
12) 전 프로그램에 대한 식별자(identifier) 색인

등이다.

2.2.1 COBOL의 구성과 특징

COBOL 프로그램의 식별자(identifier)에 대한 토큰 생성 방법이전에 COBOL 프로그램의 구성과 특징을 먼저 기술하기로 한다.

COBOL 프로그램의 구성은 다음과 같다.

DIVISION
↑
SECTION
↑
paragraph
↑
sentence
↑
statement
↑
word
↑
character

COBOL 프로그램의 특징은 다음과 같다.

1) 자료의 구조와 특성을 정의하는 부분과 자료의 처리방법 및 순서를 정의하는 부분이 완전히 독립되어 있다.
2) (space)는 단어와 단어를 구분한다.
3) sentence 이상이 끝날 때는 반드시 (period)를 사용한다.
4) 한 줄에 여러 개의 statements가 쓰여질 수 있고, 하나의 state-ment가 여러 줄에 쓰여질 수 있다.
5) A area (8 - 11컬럼)과 B area (12 - 72 컬럼)에서 시작되는 것들로 구분된다.

2.2.2 DATA DIVISION의 Symbol Table 작성 방법

DATA DIVISION에 다음과 같은 Symbol Table을 작성하여 식별자(identifier)의 사용여부 및 횟수를 검사할 수 있도록 작성하였다.

표 - 1,

식별자(identifier)	구 분	사용여부	사용횟수
	G		
	E		
	⋮		
	⋮		

표 - 1에서 구분은 식별자(identifier)의 G (group), E (element) 항목(item)을 구분하기 위한 것이다.

2.2.3 PROCEDURE DIVISION의 식별자(identifier)에 대한 토큰(token) 생성방법

다음은 PROCEDURE DIVISION의 토큰 생성 알고리즘을 제시한다.

6) SOFTOOL 80 : A Computer Example, Report No. FO 05 - 11 - 80. 1, Softool Corp., Goleta, CA FORTRAN 77 Analyzer User's Manual NTIS Report No. PB 83 - 117101, July 1982.

〈토큰 생성 알고리즘〉

```
1.  data line read
2.  begin
3.     initialize
4.     while(BLANK-FLAG=YES) do
5.     begin
6.        IO-INDEX = IO-INDEX + 1;
7.        if IO-INDEX > 65 then
8.           begin
9.             SPACE -> BLANK-FLAG;
10.            YES   -> SPACE-FLAG;
11.          end;
12.       else
13.          begin
14.            if COL(IO-INDEX) = SPACE then
15.               next sentence;
16.            else
17.               begin
18.                 SPACE -> BLANK-FLAG;
19.                 IO-INDEX -> TOKEN-START;
20.               end;
21.          end;
22.       while(SPACE-FLAG=SPACE) do
23.       if COL(IO-INDEX  = QUOTE and
             NOTE-FLAG = SPACE then
24.          begin
               SPACE -> QUOTE-FLAG;
25.            while(QUOTE-FLAG=SPACE) do
26.            begin
27.              LITERAL-PROCESS;
28.            end
29.            YES -> LITERAL-FLAG;
30.         end;
31.      else
32.         if COL(IO-INDEX) = PERIOD then
33.            begin
34.              NUMERIC-LITERAL-PROCESS;
35.            end;
36.      else
37.         if COL(IO-INDEX) = SPACE
38.            begin
39.              SPACE -> PERIOD-FLAG;
40.              YES -> SPACE-FLAG;
41.            end;
42.      else
43.         begin
```

```
44.              TOKEN-INDEX = TOKEN-INDEX + 1;
45.              COL(IO-INDEX) ->
                    TOKEN-COL(TOKEN-INDEX) :
46.              IO-INDEX -> TOKEN-END :
47.              IO-INDEX = IO-INDEX + 1 :
48.              if IO-INDEX > 65 then
49.                 YES -> SPACE-FLAG :
50.           end :
51. end :
```

다음은 위의 알고리즘에 대한 설명이 있다.

단계 1. 입력 file로부터 한 줄을 읽어들인다.

단계 2. 알고리즘에 사용되는 identifier 들에 대해 초기값을 설정한다.

/단계 4 - 21. 토큰의 첫번째 문자를 찾는다./

단계 6. IO-INDEX에 1을 더한다.

단계 7. 만약 IO-INDEX의 값이 65보다 크면 다음 데이터를 읽기 위해 BLANK-FLAG와 SPACE-FLAG의 값을 바꾸어 준다.

단계 20. 그러나, IO-INDEX의 값이 65이하이고 입력 데이터의 첨자번째가 공백이 아니면 BLANK-FLAG의 값을 바꾸어 주고, 첨자의 값을 TOKEN-START에 저장한다.

/단계 22 - 50. 토큰을 생성한다./

단계 23. 만약 입력 데이터의 첨자번째가 "(qoutation mark)이면 non-numeric literal 이므로 다음"가 나올 때까지 입력

단계 30. 데이터의 첨자번째 값을 TOKEN에 저장시킨다.

단계 31. 만약 입력 데이터의 첨자번째가 (period) 이면 소숫점을 가진 numeric literal 인지, sentence의 끝을 나타내는지를 알아 본다.

단계 35. 만약 입력 데이터의 첨자번째 값이 SPACE 이면 sentence의 끝을 나타내며, 그렇지 않으면 소숫점을 가진 numeric literal 로써 계속되는 토큰이므로 TOKEN에 저장시킨다.

단계 36.

만약 입력 데이터의 첨자번째가 공백이면 하나의 토큰이 찾아진다.

단계 41.

단계 42. 위의 경우가 모두 아니면 연속되는 토큰

단계 50. string이므로 TOKEN에 값을 저장시킨다.

2.2.4 툴의 구축

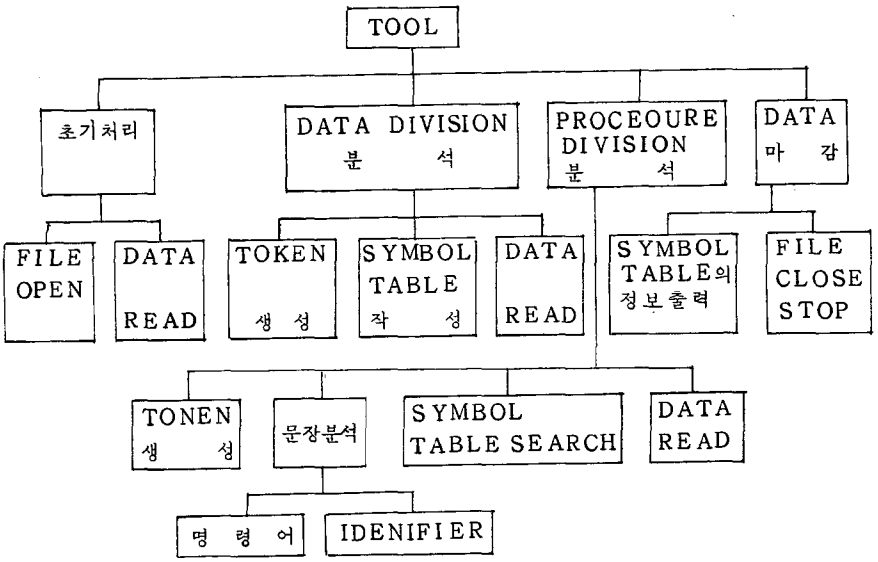

그림 - 3. 툴의 구축 구성도

3. 결 론

본 논문에서는 정적 테스팅만을 위한 툴을 구축하는 것만 연구하였으나 앞으로 동적 테스트를 위한 기능의 추가와 자동테스트, 데이터의 추출, 가정 검증을 위한 기능등을 추가하여 완전한 테스팅 툴이 되도록 계속적인 연구가 요구된다.

즉, 보다 더 높은 효율성을 위해서는 단독적인 사용보다는 정적 테스팅과 동적 테스팅을 교번적으로 혹은 함께 사용하는 방법이 바람직함으로 소프트웨어 개발, 보수 전단계에서 업무를 자동화할 수 있게 하기 위한 노력과 더불어 보다 효과적인 테스트 기법 및 정당성 증명법에 대한 연구를 병행해서 지속적으로 추진되어야 한다.

그러나 무엇보다도 어떤 항목이나 제품이 설정된 기술적인 요구 사항과 일치하는가를 확인하는데 필요한 체계적이고도 계획적인 유형의 활동 즉, 품질보증(QA : Quality Assurance)과 전 소프트웨어 사이클에 걸쳐서 여러 생산 제품의 품질을 평가하여 개선시키는데 사용되는 검토회(work through) 또는 검열(inspection) 등에 의하여 최대한의 에러(error)를 줄이는 것이 중요하다.

위의 논문을 이용하여 실행(implementation) 할 경우 Symbol Table의 정보를 파일(file)로 저장하여 수시로 정보를 검사하여 출력하는 방법과 즉시 출력하여 결과를 제공받을 수 있는 방법이 있다.

참 고 문 헌

1. Boehm, B. : "Software Engineering," IEEE Transaction on Compaters, Vol. C-25, no. 12, December 1976.
2. "New in Perspective," Datamation September, p. 124, 1980.
3. IEEE Standard Glossary of Software Engineering Terminology, IEEE Standard, p. 729, 1983.
4. Naur, P., et al (eds) : Software Engineering : Petrocell/Charter, New York, 1976.
5. Belady, L., and M. Lehman : "The characteristics of Large Systems, "in Research Directions in Software Technology, P. Wegner, ed., MIT Press, Cambridge, Mass., 1979.
6. Curtis, B., et al. : "Third Time Charm : Stronger Prediction of Programmer Performance by Software Complexity Metrics, "Proceedings of the Fourth ICSE, Munich, September, 1979.
7. Feldman, J. : "Make-A Program for Maintaining Computer Programs, "Software Practice and Experience, April, 1979.
8. Glass, R., and R. Noiseux : Software Maintenance Guidebook, Prentice-Hall, Englewood Cliffs, N. J., 1981.
9. Grier, S. : "A Tool that Detects Plagarism in Pascal Programs, "ACM SIGCSE Bulletin, Vol. 13, No. 1, February, 1981.
10. Halstead, M. : Elements of Software Science, Elsevier, New York, 1977.
11. Josephs, W. : "A Mini-Computer Based Library Control System, "Proceedings of the Software Quality and Assurance Worksshop, ACM Software Engineering Notes, Vol. 3, No. 5, November, 1978.
12. Lehman, M. : On Understanding Laws, Evoltion, and Conservation in the Lavge-Program Life Cycle, "The Jounal of Systems and Software, Vol. 1, No. 3, 1980.
13. Lientz, B., and E. Swanson : Software Maintenance Management : A study of the Maintenance of Computer Application Software in 487 Data Processing Organizations, Addison-Wesley, Reading, Mass., 1980.
14. McCabe, T. : "A Complexity Measure, "IEEE Trans. on Software Eng., Vol. SE-2, No. 4, December, 1976.
15. Shigo, O., et al. : Configuration Control for Evolutional Software Products, "Proceeding's of the Sixth ICSE, Tokyo, Japan, 1982.
16. Tichy, W. : Design, Implementation, and Evaluation of a Revision Control System, "Proceeding's of the Sixth ICSE, Tokyo, Japan, 1982.