프로그램 슬라이스 기법과 백트랙 기법을 조합한 오류 위치의 결정 방법

양 해 술† · 이 하 용††

요 약

소프트웨어 개발비용의 대부분이 생명주기 중에서도 특히 테스트 공정에 소비된다. 일반적으로 디버깅을 위해 자주 사용되는 기법으로서 백트랙(backtrack)기법과 프로그램 슬라이스 기법이 있으나 디버거에 대한 풍부한 경험이 요구되거나 대규모 프로그램에 대해서는 적용하기 곤란하다. 본 논문에서는 표준적으로 사용되고 있는 설계문서의 정보를 이용하고 백트랙기법과 프로그램 슬라이스 기법을 조합시킨 새로운 오류 위치의 결정방법을 제안하였다. 디버깅 기법을 제안하기 위해 몇가지 기본개념과 대상으로 하는 오류, 표준적인 설계문서 및 제안하는 오류 위치의 결정방법에 대해서 기술하였다. 그리고 오류가 포함된 프로그램의 예를 이용하여 제안한 방법의 절차를 설명하였으며 끝으로 제안방법과 일반 슬라이스 기법을 비교·분석하였다.

A Decision Method of Error Positions Compounding Program Slicing Method and Backtracking Method

Hae-Sool Yang† · Ha-Yong Lee††

ABSTRACT

Almost all the software development cost is especially spended in the test phase of lifecycle. Backtracking method and program slicing method are often used for debugging. But these have need of abundant experience on debuggers or can't apply for large scale programs. In this paper, I used informations of design documents which is generally used, and proposed a new determination method of error positions combining backtracking method and program slicing method. I described several fundamental concepts, error classes, standard design documents and determination method of error positions to propose a debugging method, and I explained the process of proposed method using an example program with errors. Finally, I compared and analysed the proposed method with usual slicing methods.

1. 서 론

소프트웨어 개발에서의 관건은 가능한한 최소한의 비용과 시간으로 고품질의 소프트웨어를 개발하는 것이다. 개발 기간을 최소화하고 효율적인 개발이 될 수 있도록 우수한 프로젝트 및 품질관리방법론과 정보시스템 개발방법론 등이 연구되고 있으며 그 결과 실제 업무에 활용되고 있다[1, 3, 6].

소프트웨어의 개발 기간은 공학의 다른 분야와는 달리 명확한 사전 계획에 따라 진행되지 않는 경우가 대

† 종신회원:한국소프트웨어품질연구소(INSQ) 소장
†† 종신회원:한국소프트웨어품질연구소(INSQ) 선임연구원
논문접수:1996년 12월 12일, 심사완료:1997년 3월 12일

부분이며 대규모 프로젝트일수록 이러한 문제점 때문에 예정된 기일을 지키지 못하는 경우가 많이 발생한다. 특히, 소프트웨어 생명주기 초기단계가 얼마나 충실하게 진행되었는가에 따라 개발된 소프트웨어에 대한 테스트에 소요되는 기간과 비용이 결정되므로 생명주기 단계별로 충실한 개발 과정을 거치는 것과 함께 테스트 단계에서 얼마나 효율적인 테스트 기법을 적용하고 있는가도 또한 중요한 문제라고 할 것이다[2, 8].

소프트웨어 개발비용의 대부분이 테스트 공정에서 소비되고 있으며 총 개발비의 40% 이상이 테스트 공정에서 소비된다. 테스트 공정은 고장을 검출하는 테스트와 고장의 원인인 오류를 수정하는 디버깅의 2가지 작업으로 분류된다. 일반적으로 오류 위치의 결정이 디버깅에서 가장 시간이 걸리는 작업이라고 말할 수 있으며[1], 오류의 위치를 효율적으로 결정하는 방법의 개발이 중요해지고 있다.

통상 자주 사용되는 디버깅 기법으로서 백트랙(backtrack)기법은 고장의 징후를 최초로 본 장소로부터 프로그램의 제어흐름을 역으로 추적하여 그 징후가 소멸되는 위치를 볼 수 있으므로 결국 오류가 존재하는 범위를 정할 수가 있다. 그러나 프로그램 실행시에 몇 개의 장소에서 변수의 올바른 값을 알 필요가 있지만 구체적인 기법은 확립되어 있지 않다. 또, 백트랙기법은 디버거에 대한 지식이나 경험에 상당히 의존한다는 문제점이 지적되고 있다[8]. 한편, 프로그램 슬라이스 기법(Program Slicing)[4, 9, 10]을 이용한 오류 위치의 결정기법도 제안되어 있다[2, 15]. 프로그램 슬라이스 기법은 Mark Weiser에 의해 제안된 것으로 자료 및 제어 흐름을 분석하여 프로그램 행위를 관심있는 특정 부분으로 나누는 방법이다[13]. 이 기법에서는 값에 오류가 있는 변수에 대해서 모든 프로그램에 걸쳐 그 변수의 값에 영향을 줄수 있는 문의 집합(슬라이스)를 찾고 그 슬라이스 중에서 오류가 되고 있는 문을 추출한다. 따라서 전술한 백트랙기법과는 다르고 디버거의 지식이나 경험에 의지하지 않고 해결된다는 이점이 있다. 그러나 최악의 경우 슬라이스로서 프로그램의 모든 문이 추출되는 것이 있고 오류 위치 결정에 노력이 많이 든다. 즉, 대규모적인 프로그램에 대한 디버깅에 적용하는 것은 일반적으로 곤란하다는 단점이 있다[12].

본 연구에서는 표준적으로 사용되고 있는 설계문서의 정보를 이용하고 백트랙기법과 프로그램 슬라이스 기법을 조합시킨 새로운 오류 위치의 결정방법을 제안한다. 제2장에서는 디버깅 기법을 제안하기 위한 준비로서 몇가지 기본개념에 대해 기술하였으며 3장에서는 대상으로 하는 오류, 표준적인 설계문서 및 제안하는 오류 위치의 결정방법에 대해서 기술한다. 4장에서는 오류가 들어있는 프로그램의 예를 이용하여 제안한 방법의 절차를 설명하고 5장에서는 제안한 방법과 일반 슬라이스 기법을 이용한 기법을 비교하였으며 마지막으로 결언 및 향후 연구과제에 대해 기술하였다.

2. 관련 연구

2.1 소프트웨어 개발 프로세스

소프트웨어의 개발은 일반적으로 (그림 1)과 같은 흐름에 따라 이루어진다. 요구분석 단계에서 요구명세서가, 설계단계에서 설계문서가, 구현 단계에서 프로그램이 각각 작성된다.

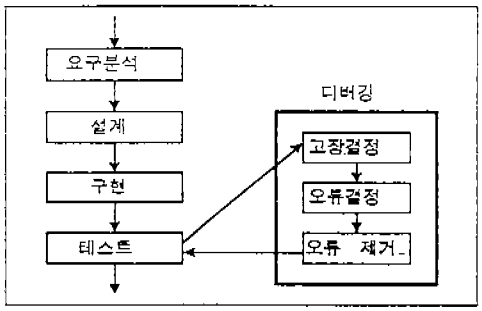
2.2 테스트와 디버깅

소프트웨어 테스트는 노출되지 않은 숨어 있는 결함을 찾기 위해 수동 또는 자동화된 조작에 의해 시스템 또는 시스템 구성요소를 가실행하고 조작하여 지정된 요구명세를 만족하고 있는가를 검증하거나 예상과 실제의 동작 차이를 확인하는 과정이다. 테스트는 일반적으로 다음과 같은 단계로 실시한다.

(단계 1) 테스트 데이타를 사용하여 프로그램을 실행한다. 요구명세를 이용하여 구하는 예상결과와 출력결과를 비교하여 프로그램이 바르게 동작하고 있는지를 확인한다.

(단계 2) 프로그램이 바르게 동작하지 않을 때는 그 원인 즉 오류를 찾아 제거한다. 이 작업을 (그림 1)과 같이 디버깅이라고 한다. 프로그램을 수정한 후 다시 테스트를 실시하여 프로그램이 바르게 동작하는지를 확인한다.

즉, 디버깅은 소프트웨어에서 발생된 오류의 원인을 찾아 교정하는 작업이다. 테스트를 성공적으로 수행하였을 경우 잠재하고 있던 오류를 발견할 수 있으며 그 다음 단계로 디버깅을 수행하게 되고 오류가



(그림 1) 소프트웨어 개발 프로세스
(Fig. 1) Software development process

야기하는 징후를 분석하여 개발자가 그 원인을 찾는 논리적 과정과 원인을 발견한 후에는 수정을 가하는 작업이 필요하다. 본 논문에서는 주로 디버깅에 착안하여 새로운 오류위치의 결정방법을 제안한다.

2.3 종래의 디버깅 기법

일반적으로 디버깅을 위한 접근방법으로 다음과 같은 세가지 범주를 들 수 있다[1].

1) Brute force

소프트웨어의 에러를 수정하는 가장 일반적이며 효율이 떨어지는 방법으로서 이 방법이 이용되는 이유는 생각을 적게 할 수 있으며 정신노동력을 감소시킬 수 있기 때문이다. 다음과 같은 세가지 기법이 주로 이용된다.

① 메모리 덤프: 문제의 원인을 메모리의 정보에서 분석하는 것으로 메모리와 원시코드와의 관계가 확실하지 않으며, 불필요한 부분까지 살펴보아야 하고, 정적인 데이타를 분석하는 것에 불과하므로 효용성이 부족하다.

② 출력문 삽입: 소프트웨어의 도처에 출력문을 임의로 삽입하여 수행시켜서 에러를 찾아내는 기법으로 문제의 원인을 놓치거나 불필요한 데이타를 분석하게 되며 대형 소프트웨어인 경우 많은 노력이 소요된다.

③ 디버깅 도구의 이용: 디버깅 도구를 이용하여 수행과정을 추적하고 필요한 지점에서 명령의 수행을 중단하여 검사할 수 있으나 문제의 근원을 놓칠 수 있다.

2) 백트랙킹

백트랙킹은 오류가 발견된 위치에서 역으로 거슬러 올라가 오류를 발생시킨 원인을 찾는 기법으로 소형 프로그램에서는 성공적으로 사용될 수 있는 디버깅 기법이다. 소스 코드의 라인수가 증가 함에 따라 잠재적인(potential) 후행경로(packward path)가 통제할 수 없을만큼 많아진다.

3) 원인 제거(Cause elimination)

오류 발생에 관련된 데이타는 잠재적인 원인을 찾기 위해 조직화되며, 하나의 원인 가설(cause hypothesis)이 도출되고 위의 데이타가 가설을 입증하거나 기각하기 위해 사용된다. 모든 가능한 원인의 목록이 만들어지고 각각을 제거하기 위해 수행된다. 테스트가 특별한 원인 가설이 가망이 있음을 보이면 그 데이타는 오류를 제거하기 위한 시도로 정련된다.

이상과 같은 디버깅 기법으로 많이 사용되고 있는 백트랙기법에서는 우선 어디에서 고장의 징후가 최초로 발견되었는가를 조사한다. 다음에 프로그램의 제어흐름을 역으로 추적하여 그 징후가 처음으로 제거되는 위치를 발견하여 오류가 존재하는 범위를 지정한다. 마지막으로 오류가 존재하는 범위를 면밀히 검토하여 오류의 위치를 결정한다.

한편, 문헌[2] 등에서는 프로그램 슬라이스 기법을 이용한 오류 위치의 결정기법이 제안되어 있다. 프로그램 슬라이스 기법이란 프로그램 내의 어떤 문의 실행에 영향을 줄 수 있는 모든 문을 추출하는 기술이고, 추출된 문의 집합을 슬라이스라 부른다[9, 10]. 프로그램 슬라이스 기법을 이용하는 디버깅 기법에서는 값에 오류가 있는 변수에 대해서 모든 프로그램에 걸쳐 슬라이스를 찾고 그 슬라이스 중에서 오류가 되고 있는 문을 발견할 수 있다. 문제점으로서 백트랙 기법에서는 프로그램 실행시에 몇몇 위치(프로그램 내의 문)에서 바른 상태를 알 필요가 있으며 디버깅에 대한 지식이나 경험에 상당히 의존한다. 슬라이스 기법을 이용하는 디버깅 기법은 대규모 프로그램의 디버깅에는 적용이 곤란하며 또한 일반 프로그램 슬라이스 기법을 이용한 오류 위치의 결정기법에서는 문기술이 누락된 오류를 검출할 수 없다. 문헌[5, 7]에서는 문기술이 누락된 오류를 검출하기 위해서 C 슬라이스기법(Critical Slicing)을 제안하고 있다. 그러나 C 슬라이스기법도 최악의 경우 슬라이스로서 프로그램의 모든 문이 추출되는 경우가 있고 오류 위치의 결정에 시간이 걸리므로 대규모 프로그램에 대한 오

류의 결정에는 적용하기 어렵다.

3. 제안 방법

3.1 대상으로 하는 오류

프로그램이 실행될 때 명세에 포함되지 않은 결과가 나오는 것을 고장이라 하고 그 고장을 일으킨 원인을 오류라고 정의하였으며 제안방법에서의 프로그램의 오류는 기본적으로 다음과 같이 4가지로 구분할 수 있다.

(1)문기술 누락 오류

프로그램 실행 결과가 올바르게 산출되기 위해 반드시 필요한 문의 기술이 누락되어 있어서 실행 결과에 오류를 발생시킨 경우, 문기술 누락 오류가 발생했다고 한다.

(2)문기술 과다 오류

불필요한 문이 기술되어 있어 실행 결과에 결과에 영향을 주는 변수 등의 내용을 변경시키는 등의 오류를 발생시킨 경우, 문기술 과다 오류가 발생한 것이다.

(3)문기술 오류

필요한 문이 누락되었거나 불필요한 여분의 문이 존재하지는 않지만 문의 기술 자체에 오류가 있어서 실행 결과에 오류를 유발하는 경우, 문기술 오류가 발생한 것으로 오퍼레이터(operator)에 발생한 오류를 말한다.

(4)명칭 기술 오류

다음에 기술한 ①~④까지의 오류가 발생한 경우, 명칭 기술 오류가 되며 오퍼랜드(operand)에 발생한 오류를 말한다.

①대입문에서 좌측의 변수명이 잘못된 것

②함수의 이름이 잘못되어 있는 것

③함수 호출문에서 출력 파라메타에 대응하는 인수명이 잘못되어 있는 것

④입력문에서 입력변수명이 잘못되어 있는 것

예를 들어 대입문 A:=B;에 대해 바른 문이 C:=B;인 경우에는 명칭 기술 오류가 발생한 것이고 대입문 A:=B-D;에 대해 바른 문이 A:=B+C;인 경우에는 문기술 오류가 발생한 것이다.

3.2 설계문서

설계문서는 설계단계에서 수행한 작업결과를 기록한 내용으로 프로그래머, 시험자, 유지보수자에게 유용한 것으로서 설계문서에는 설계 개요로부터 모듈 기술까지 여러 가지 정보가 포함된다[3, 4]. 본 연구에서 제안하는 오류 위치의 결정방법에 필요한 설계문서상의 정보를 통합한다.

(1)구조 모델

모듈과 그들 간의 데이타 흐름이나 제어흐름에 대해 기술한 것으로 시각적인 표기법(예를 들면 DFD)이 적용되고 있다.

(2)모듈 기술

모듈의 기술은 다음의 ①~⑥으로 구성된다.

①모듈의 행위(Module behavior):제공하는 동작을 그 입출력과 외관상의 효과와 더불어 기술한 것으로 모듈이 제공하는 데이타형과 객체의 기술도 포함한다.

②전제(Assumptions):모듈이 올바르게 동작하기 위해 필요한 조건을 의미한다.

③제한과 한계(Constraints and Limitations):성능요구, 최대 메모리 사용요구, 수치정도 등을 의미한다.

④오류 처리(Error handling):오류와 예외조건 및 그 원인과 대처방안 등을 의미한다.

⑤모듈 패키징(Module packaging):제공하는 모든 함수, 함수의 형, 입출력 파라메타의 형, 각 함수의 처리결과와 부작용, 모듈이 제공하는 데이타형과 객체의 특징에 관한 상세한 설명 등을 의미한다.

⑥내부 구조(Internal structure):은폐되어 있는 데이타형과 객체, 내부함수, 함수의 형, 파라메타의 형, 효과 등, 특수한 알고리즘을 이용하고 있는 경우에는 그 기술도 포함한다.

3.3 오류 위치의 결정 방법

어떤 테스트 데이타를 이용하여 테스트를 수행하고 고장이 발생했다고 가정하면 그 고장의 원인 즉 오류를 검출할 필요가 있다. 본 연구에서는 프로그램 개발의 요구분석단계에서 작성된 요구명세서와 설계단계에서 작성된 설계문서를 이용하여 오류 위치를 결정하였다. 이하에 각 단계에 대해 상세히 설명하기로 한다.

(단계 1):테스트 데이타를 이용하여 테스트했을 때 출력 결과의 불일치가 발생한 최초의 출력문을

결정한다. 이 단계에서는 요구명세와 프로그램이 입력으로 필요하다.

(단계 2):(2.1) 최초의 문으로부터 (단계 1)에서 결정한 출력문까지에 해당하는 범위의 프로그램 실행계열과 그것에 대응하는 설계문서로부터 본 실행계열을 구한다.

(2.2) 백트랙기법을 이용하여 프로그램의 실행계열에서 설계문서로부터의 실행계열과 일치 하지 않는 최초의 블록을 검출한다. 동시에 그 블록의 출구에서 값에 오류가 있는 변수를 구할 수 있다.

(2.3) (2.2)에서 검출한 블록에 대응하는 모듈에 주목하고, (2.2)에서 구한 변수에 대해서 D(Dynamic) 슬라이스법을 적용한다.

(2.4) (2.3)에서 구한 슬라이스 중에서 잘못이 있는 문을 결정한다.

(단계 3):(단계 2)에서 결정한 잘못이 있는 문을 정정한다.

3.4 제안 방법의 특징 및 장점

기존의 오류위치 결정방법과 본 논문에서 제안한 방법과 비교하여 개선된 사항으로 다음과 같은 것들이 있다.

①오류의 존재범위를 모듈 단위로 최소화:백트랙 기법의 경우 프로그램의 처음부터 오류의 징후가 나타난 위치까지의 전체를 대상으로 오류를 검출하며, 프로그램 슬라이스 기법은 오류가 발생한 변수에 대해 모든 프로그램에 걸쳐 슬라이스를 찾고 그 슬라이스를 검토하여 오류를 유발한 위치를 결정하는 반면 제안한 방법은 오류의 존재 범위를 설계서와 프로그램의 실행계열을 비교함으로써 실행계열이 일치하지 않는 위치가 속한 하나의 모듈로 제한할 수 있다. 즉, 백트랙 기법에서 오류의 징후가 나타난 위치는 프로그램상에서 실제 오류가 있는 위치가 아니라 출력문 등을 통하여 프로그램에 오류가 존재한다는 사실만을 인지한 위치라 할 수 있으므로 오류의 정확한 위치를 결정하기 위해서 프로그램의 처음부터 오류의 징후가 나타난 위치까지를 대상으로 제어흐름을 역으로 추적하는 것이다.

②다양한 오류의 검출 가능:일반적인 프로그램 슬라이스 기법의 경우 3.1절에서 제시한 대상 오류중 문기술이 누락된 오류에 대해서는 검출할 수 없으나 제안한 방법은 백트랙 기법을 조합함으로써 해결하였다.

③오류위치 결정을 위한 계산량 감소:백트랙 기법이나 프로그램 슬라이스 기법에 비해 제안한 기법은 오류의 존재 범위를 모듈 단위로 최소화함으로써 오류위치를 결정하기 위해 필요한 계산량을 비약적으로 줄일 수 있다.

이상과 같이 기존의 오류위치 결정방법의 문제점과 제안한 방법을 통해 개선한 사항은 다음 장에서 예제 프로그램을 적용하여 상세히 설명하였다.

4. 적용 결과

오류를 포함시킨 프로그램 예 exam을 이용하여 제안 방법의 절차를 설명하고 기존의 방법을 적용하였을 경우의 문제점을 기술하기로 한다. exam은 정수의 비교, 승산, 제산, 자승연산을 정의하고 있고 입력으로 주어지는 2개의 정정수에 대해 지시된 연산을 수행하고 결과를 출력한다.

4.1 제안 방법의 적용

본 절에서는 제안한 오류위치 결정방법에 대한 절차를 예제 프로그램 exam을 이용하여 설명하였다.

4.1.1 요구명세

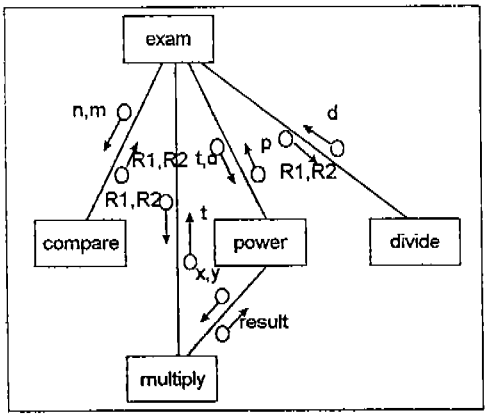
i_1, i_2를 exam의 입력, o_1, o_2를 exam의 출력이라고 가정한다. 문헌[11]에서 소개된 input-output assertions을 이용하여 exam의 요구명세를 이하에 표현하였다.

$\{i_1 > 0 \text{ and } i_2 > 0\}$
exam
$\{((u = i_1 \text{ and } v = i_2 \text{ or } u = i_2 \text{ and } v = i_1) \text{ and } u \geq v)$
$\text{and } o_1 = \text{multiply}(u, v) \text{ and } w = \text{divide}(u, v)$
$\text{and } o_2 = \text{power}(o_1, w)\}$

4.1.2 설계문서

오류 위치의 결정에 필요한 exam의 설계문서 정보는 (그림 2)와 같고 (a)는 exam의 구조 모델을 나타내는 구조도를 기술하였다. 본 예는 설계방법론으로 구

조화 설계를 이용하고 있다. 구조화도 DFD, 자료사전, 미니명세서가 필요하지만 예가 간단하기 때문에 생략하였으며 (그림 2) (b)에 exam의 모듈을 기술하였다. 그림 중의 (A)-(F)는 3.2절의 모듈 기술 항목 (A)-(F)와 대응하고 있으며 일부 항목을 생략하고 있다.



(a) exam의 구조도

exam.
(A) 이 모듈은 프로그램의 main 함수를 포함하고 다른 모듈과의 사이에 데이타를 교환한다.
(E) Main program.
(F) 다음의 처리를 수행한다 :
Read(in/out: n, m);
Min_Max(in: n, m);
t ← Times(in: R1, R2);
Write(in: t);
d ← Div(in: R2, R1);
p ← Power(in: t, d);
Write(in: p);

multiply.
(A) 이 모듈은 입력 파라메타로서 주어진 두 개의 정수의 곱을 되돌린다.
(E) 함수 int Times(int x, int y).
(F) 다음의 처리를 수행한다:
j:=0; w:=x;
while j<y do { w:=w+x; j:=j-1; };

divide.
(A) 이 모듈은 입력 파라메타로서 주어진 두 개의 정수의 상을 되돌린다.
(D) 다음의 오류를 검사하고 처리한다:
- Divisor zero: Output' 제수 0!' and return FAIL.
(E) 함수 int Div(int u, int v).
(F) 다음의 처리를 수행한다:
q:=0; w:=u;
while w >= v do { q:=q+1; w:=w-v; }

power.
(A) 이 모듈은 입력 파라메타로서 주어진 정정수 x와 비부정수 y에 대해서 적절한 승 x^y을 되돌린다.
(E) 함수 int Power(int x, int y).
(F) 다음의 처리를 수행한다:
i:=1; result:=1;
while i<=y do { result:=Times(result, x); i:=i+1; }

compare.
(A) 이 모듈은 입력 파라메타로서 주어진 2개의 정수를 비교하고 작은 편의 수를 외부변수 R1으로 나머지 수를 외부변수 R2에 넣는다.
(E) 함수 void Min_Max(int u, int v).
(F) 다음의 처리를 행한다:
R1 := if x<=y then x else y;
R2 := if x>y then x else y;

(b) exam의 모듈 기술

(그림 2) 설계문서
(Fig. 2) Design document

4.1.3 테스트 대상 프로그램

테스트 대상 프로그램을 (그림 3)에 보였다.

```
#include <stdio.h>
int R1, R2;
int Times(x, y)
    int x, y;
{
    int j=1;
    int w=x;
    while (j<y)
        {
            w+=x;
            j++;
        }
    return w;
}

int Power(x, y)
    int x, y;
{
    int i;
    int result=1;
    if (y>0)
        for (i=1; i<=y; i++)
            result *=x;
    return result;
}

void Min_Max(x, y)
        int x, y;
{
    extern int R1;
    extern int R2;
    R1=x;
    R2=x;
    if (x<y)
        R1=y;
    else
        R2=y;
}
```

```
int Div(u, v)
    int u, v;
{
    int q=0;
    int w=u;
    if (v==0)
        printf("제수 0!\n");
    else
        while (W>=v)
        {
            q+=;
            w=w-v;
        }
    return q;
}

main()
{
    int n, m;
    int t, d, p1, p2;
    scanf("%d", &n);
    scanf("%d", &m);
    Min_Max(n, m);
    t=Times(R2, R1);
    printf("t = %d\n", t);
    d=Div(R2, R1);
    p=Power(t, d);
    printf("Power = %d\n", p);
}
```

(그림 3) 테스트 대상 프로그램
(Fig. 3) Test program

4.1.4 오류 예

본 논문에서 대상으로 하고 있는 오류는 3. 1에서 제시한 바와 같이

①문기술 누락 오류
②문기술 과다 오류
③문기술 오류
④명칭 기술 오류

이다. 이중에서 문기술 오류는 오퍼레이터(operator)에 오류가 발생한 것으로 (그림 3)의 테스트 프로그램에서 제시하였으며 오류 위치 결정과정을 설명하는 예를 문기술 오류를 이용하여 제시한다. 테스트 프로그램에 존재하는 오류는 함수 Min_Max 중의 문 if(x<y)에서 오퍼레이터에 해당하는 관계연산자 '<'이며 뒤에서 설명하는 것처럼 문 if(x〈=y)가 올바른 문이다.

4.1.5 오류 위치의 결정

(그림 3)의 프로그램 출력은 main의 출력문 Printf ("t=%d \n", t); 와 printf("Power=%d\n", p); 에 의해 얻어진다. 테스트 데이타 (x, y)=(5, 2)에 대한 프로그램의 출력은 (10, 1)이다. 한편, 요구명세에 의한 출력은 (o_1, o_2)=(10, 100)이 되므로 달라져 있다. 따라서 오류가 존재하는 것을 이 단계에서 알 수 있다. 다음에 제안한 방법을 이용하여 오류 위치를 결정하는 과정을 기술하였다.

(단계 1): 요구명세서에 따라 얻어지는 출력과의 비교에 의해 프로그램 상에서 불일치가 발생한 최초의 출력문이 printf("Power=%d\n", p);라는 것을 알 수 있다.

(단계 2): (2.1) 테스트 데이타 (5, 2)에 대해서 문 printf ("Power=%d\n", p);까지의 프로그램의 실행계열을 (그림 4)(b)에, 대응하는 설계문서에 의한 실행계열은 (그림 4)(a)이다. 여기에서 인수(5, 2, 2, 5) 등은 어떤 문을 실행할 때 그 문이 속하는 모듈의 상태(정의되어 있는 모든 변수의 값)를 나타낸다. 또, (그림 4)(a)의 생략기호(:)는 그 사이에서의 상태가 미지인 것을 의미한다. 한편, (그림 4)(b)의 생략기호(:)는 문의 반복실행을 나타낸다. 설계문서에 의한 실행계열과 프로그램에 의한 실행계열에 대한 상태의 대응관계를 파선으로 표시하고 있다. 파선상의 레벨의 숫자는 (2.2)에서 백트랙을 설명할 때 이용한다.

(2.2) 우선 2개의 실행계열상에서 문 printf ("Power=%d\n", p);를 실행한 후의 상태(레이블 1의 파선으로 맺어져 있다)와 점선으로 연결되어 있는 상태가 같은것을 비교한다. 만약 조건 C "레이블 i의 점선으로 맺어져 있는 상태가 서로 떨어져 있고, 그럼에도 불구하고 레이블 i+1의 점선으로 맺어져 있는 상태가 동일하다."가 성립한다면, i번째의 상태와 i+1 번째의 상태간에 고장의 원인이 존재하는 것을 알 수 있기때문에 백트랙은 중지된다. 조건 C가 성립하지 않는 경우는 i를 1 증가하고, 조건 C가 성립할 때까지 반복한다. 본 예의 경우, 레이블 13의 파선으로 맺어져 있는 상태가 다르고, 레이

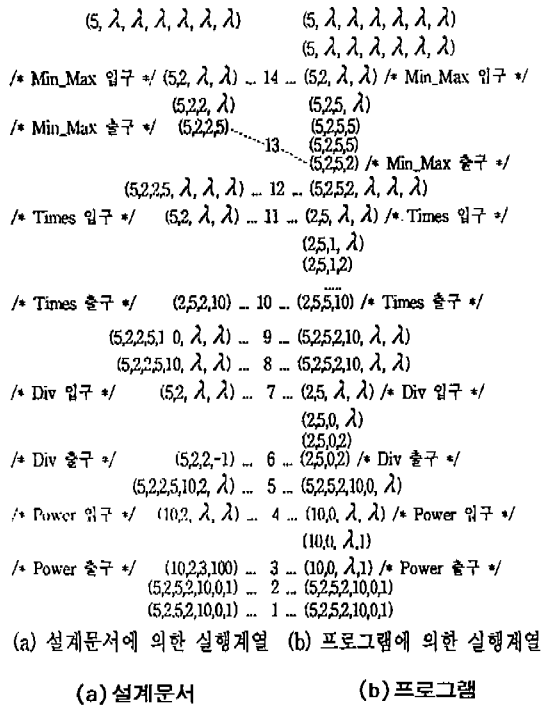
블 14의 파선으로 맺어져 있는 상태가 같게 되어 있기 때문에 고장의 원인이 그 두 상태 사이에 존재하는 것을 알 수 있다. 결국, 오류가 모듈 Min_Max에 있는 것이 확인된다.

(2.3) R1, R2에 대한 모듈 Min_Max의 D 슬라이스를 구하면 (그림 5)에 보인 문의 집합이 구해진다.

(2.4) 그 슬라이스 중에서 조건문 if(x<y)가 잘못되어 있다는 것을 알 수 있다.

(단계 3): 조건문 if(x<y)를 if(x>=y)로 변경하여 P를 변경한다. 다시 테스트를 행한다.

(그림 4)에서 λ는 해당 시점에서 아직 변수의 값이 결정되어 있지 않음을 나타낸다. (그림 4)에서 모듈 Min_Max 출구를 보면 설계문서의 실행계열과 프로그램의 실행계열에서 각각 (5, 2, 2, 5)와 (5, 2, 5, 2)로 두 변수의 값이 달라져 있음을 알 수 있다. 이들 변수의 값에 영향을 미칠 수 있는 슬라이스를 모듈 Min_



(a) 설계문서에 의한 실행계열 (b) 프로그램에 의한 실행계열

(a) 설계문서 (b) 프로그램

(그림 4) 설계문서와 프로그램의 실행계열

(Fig. 4) Execution order of design document and program

Max에서 찾아 구성하면 그 슬라이스 중에서 오류의 원인을 찾을 수 있다. 이러한 방법을 통해 기존의 백트랙기법이나 프로그램 슬라이스기법에 비해 오류의 존재범위를 제한할 수 있다.

```
void Min_Max(x, y)
     int x, y;
{
   extern int R1;
   extern int R2;
   R1=x;
   R2=x;
   if (x<y)
      R1=y;
   else
      R2=y;
}
```

(그림 5) R1, R2에 대한 Min_Max의 D 슬라이스

(Fig. 5) D-slice of Min_Max for R1, R2

4.2 기존 방법의 적용

예제 프로그램 exam에 대해 기존의 오류위치 결정 방법들을 사용하였을 경우의 문제점을 살펴보았다.

4.2.1 프로그램 슬라이스 기법의 적용

예제 프로그램 exam에서는 문기술 누락 오류에 대한 예를 제시하고 있지 않으나 프로그램 슬라이스 기법을 적용할 경우에는 오류가 발생한 변수 p에 대한 슬라이스를 검출하였을 때 누락된 문이 슬라이스에 속하지 않을 경우 누락 오류를 발견할 수 없게 된다.

또한, 제안한 오류 위치 결정방법을 앞 절의 예제 프로그램에 적용하면 이미 기술한 형태로 오류의 범위는 (그림 5)에 보인 6행의 문집합이 되지만 D 슬라이스기법만을 이용하면, 프로그램 전체를 대상으로 슬라이스를 결정하므로 오류의 범위는 (그림 6)에 보인 58행의 문집합이 되고 결국 프로그램 전체가 오류의 범위가 된다. 따라서 오류의 위치를 결정하기 위해서는 프로그램 전체를 검토해야 하고 오류 결정을 위해 많은 계산이 필요하게 되므로 대규모 프로그램의 경우에는 적용하기 곤란한 경우가 발생할 수 있다. 문기술 누락 오류를 검출할 수 없는 D 슬라이스 기법의 단점을 개선한 C 슬라이스 기법을 적용할 수도 있으나 오류의 범위가 줄어드는 개선은 기대할 수 없다.

```
#include <stdio.h>
int R1, R2;
int Times(x, y)
    int x, y;
{
    int j=1;
    int w=x;
    while (j<y)
        {
            w *=x;
            j++;
        }
    return w;
}

int Power(x, y)
    int x, y;
{
    int i;
    int result=1;
    if (y>0)

    return result;
}

void Min_Max(x, y)
        int x, y;
{
    extern int R1;
    extern int R2;
    R1=x;
    R2=x;
    if (x<y)
        R1=y;
    else
        R2=y;
}

int Div(u, v)
    int u, v;
{
    int q=0;
    int w=u;
    if (v==0)
        printf("제수 0!\n");
    else
        while (w>=v)
    return q;
}

main()
{
    int n, m;
    int t, d, p1, p2;
    scanf("%d", &n);
    scanf("%d", &m);
    Min_Max(n, m);
    t=Times(R2, R1);
    printf("t = %d\n", t);
    d=Div(R2, R1);
    p=Power(t, d);
    printf("Power = %d\n", p);
}
```

(그림 6) 출력변수 p에 대한 exam에서의 D 슬라이스
(Fig. 6) D-Slice in exam for output variable p

4.2.2 백트랙 기법의 적용

제안한 오류 위치 결정방법에서는 오류의 범위가 함수 Min_Max로 제한되고 오류가 발생한 변수에 대한 프로그램 슬라이스를 검출하여 오류의 범위가 더욱 축소되는 반면, 백트랙 기법에서는 프로그램의 최초의 문에서 오류의 징후가 나타난 위치인 변수 p에 관한 출력문까지의 모든 실행계열이 오류의 존재 범위가 되므로 프로그램 슬라이스 기법과 마찬가지로 오류를 결정하기 위해 필요한 계산량이 많아지고, 오류의 징후가 나타난 위치로부터 제어 흐름을 역으로 추적해 나가는 과정에서 원시 문장의 수가 증가함에 따라 가능한 역행 경로의 수가 기하급수적으로 증가할 수 있으므로 대규모 프로그램에는 적용하기 곤란한 경우가 발생할 수 있다.

5. 평 가

이번 장에서는 예제 프로그램을 적용한 것을 바탕으로 제안한 오류위치 결정방법과 기존의 방법들을 비교·평가해 보았다.

5.1 평가기준

프로그램 오류의 결정방법을 평가하는 주된 속성으로서는 일반적으로 결정이 가능한 오류의 종류수, 오류의 존재범위, 오류를 결정하기 위해서 필요한 계산량 등이 고려될 수 있다. 예를 들면, 실제의 오류가 오직 1개의 문에서 나타나고 있기 때문에, 결정한 오류의 존재범위가 거의 프로그램 전체가 된다면 그 방법은 실용적이라고 말할 수 없을 것이다. 또, 프로그램이 복수의 모듈로 구성되어 있는 경우, 오류가 존재하지 않는 모듈에 대해 필요없는 검사나 계산을 하는 불필요한 노력을 소비하는 것이 된다. 특히, 대규모 프로그램의 오류에 대처하려면 오류의 존재범위를 보다 정확하게, 보다 좁게, 그리고 보다 빠르게 결정할 수 있어야 한다.

5.2 비 교

본 논문에서 제안한 오류 위치의 결정방법과 백트랙 기법, D 슬라이스법, C 슬라이스법의 비교결과를 〈표 1〉에 나타내었다. 제안방법에서는 3.1의 기준에 따라 오류를 4종류로 구분할 수 있다. 즉, 설계서를 이용하여 잘못된 모듈과 변수를 결정한 후, 그 모듈 내에서 백트랙기법이나 C 슬라이스기법을 적용하면 4종류의 오류를 결정할 수 있다. 만약, 거기에서 C 슬라이스기법을 대신하여 D 슬라이스기법을 적용하면 문이 누락된 오류를 결정할 수 없게 된다.

그리고 오류가 존재하는 범위에 있어서 제안하는 방법에서는 오류가 존재하는 모듈내에서의(값이 잘못된 변수에 대한) 슬라이스까지 결정할 수 있다. 반면, 백트랙기법에서는 최초의 문에서 출력결과의 불일치가 발생한 위치까지의 모든 실행계열이 오류 존재범위가 되며 슬라이스기법에서는 모든 프로그램 전체의(어떤 변수에 대한) 슬라이스까지밖에 결정할 수 없다. 또, 계산량에 관해서는 제안한 방법은 설계서의 어려움이 따르지만, 슬라이스를 구하는 방법이 다른 2개에 비해 무시할 수 있는 정도로 적어진다. 따라서 본 논문에서 제안한 오류 위치의 결정방법은 대규모 프로그램에 대해서도 충분히 적용가능하다고 볼 수 있다.

〈표 1〉 오류 결정방법의 비교
〈Table 1〉 Comparison of error decision methods

방 법	A1	A2	계산량
D 슬라이스법	3	모든 프로그램의 어떤 슬라이스	많음
C 슬라이스법	4	모든 프로그램의 어떤 슬라이스	많음
백트랙 기법	4	최초의 문에서 출력결과의 불일치가 발생한 문까지의 범위	많음
제안 방법	4	어떤 모듈의 어떤 슬라이스	적음

(주) A1:결정할 수 있는 오류의 종류
A2:결정할 수 있는 오류의 존재범위

6. 결 론

소프트웨어 생명주기는 크게 개발과정과 유지보수 과정으로 나눌 수 있으며, 유지보수 과정에서 소요되는 비용이 생명주기에서 소요되는 총 비용의 80% 이상을 차지하고 있고 지속적으로 증가하는 추세이므로 유지보수의 중요성에 대한 인식이 매우 높아지고 있으며 관련 연구가 활발히 진행되고 있다. 또한, 개발과정에서는 40% 이상의 비용이 테스트 단계에서 소비되고 있으므로 본 연구에서는 이와같은 현실에서 체계적인 테스트 기법을 개발하고 테스트 결과 발견된 오류에 대한 위치를 찾기 위해 효율적인 디버깅 기법을 활용하는 것은 개발비용을 절감할 수 있는 효과적인 방법이라 할 수 있다.

기존에 백트랙 기법이나 프로그램 슬라이스 기법 등의 오류 위치를 결정하는 기법들이 개발되어 사용되어 왔으나 각각 그 단점으로 인한 개선의 필요성이 증대되고 있다. 즉, 백트랙 기법은 디버거에 대한 상당한 경험을 필요로 하며 프로그램 슬라이스 기법의 경우에는 오류 위치결정에 많은 노력이 들고 대규모 프로그램에 대한 디버깅에는 적용하기 곤란하다는 문제점이 있다. 최근 소프트웨어의 규모가 계속적으로 확대되어 가는 점을 감안하면 해결해야 할 문제점이라 생각할 수 있다.

따라서, 본 연구에서는 기존의 백트랙기법과 프로그램 슬라이스기법 등의 문제점을 관련 연구를 통해 제시하고 이러한 문제점을 개선하기 위해 백트랙기법과 프로그램 슬라이스기법을 조합시킨 새로운 오류위치의 결정방법에 대해서 제안하였다. 제안한 방법에 대해서는 오류를 포함시킨 예제 프로그램을 이용하여 오류위치의 결정 절차를 설명하였다. 또한, 본 논문에서 제안하는 방법 또는 시스템의 가이드에 의한 결정방법을 적용한 결과를 소개하고, C 슬라이스 기법 및 D 슬라이스 기법과의 비교를 통해 대규모 프로그램에 대한 적용 가능성을 보였다. 향후 연구과제로는 제안한 오류 결정방법을 대규모 프로그램에 적용하여 적용가능성을 검증하고 본 오류 결정방법을 지원할 수 있는 도구의 개발을 들 수 있다.

참 고 문 헌

[1] G. J. Myers, "The Art of Software testing," Wiley, 1982.

[2] H. Agrawal, R. A. Demillo and E. H. Spafford, "Debugging with dynamic slicing and backtracking," Software:Practice and Experience, Vol. 23, No. 6, pp. 589-616, 1993.

[3] W. B. Frakes, C. J. Fox and B. A. Nejmeh, "Software Engineering in the UNIX/C Environment," Prentice-Hall, 1991.

[4] Keith Brian Gallagher and James R. Lyle, "Using Program Slicing in Software Maintenance," IEEE Trans. Software Engineering, Vol. 17, No. 8, 1991.

[5] 下村隆夫, "プログラムバグ潜在域の最適化に關する考察," 情報處理學會論文誌, Vol. 35, No. 8, pp. 1591-1601, 1994.

[6] Ian Sommerville, "Software Engineering," Addi-

son-Wesley Publishing Company, 1989.

[7] 下村隆夫, "變數値Bエラ-における Critical Sliceに基づくバグ究明戰略," 情報處理學會論文誌, Vol. 33, No. 4, pp. 501-511, 1992.

[8] Roger S. Pressman, "Software Engineering," 3rd. Ed., McGraw-Hill, 1992.

[9] M. Weiser, "Programmers use slices when debugging", Communications of the ACM, Vol 25, No. 7, pp. 446-452, 1982.

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

[11] C. Ghezzi, M. Jazayeri, and D. Mandrioli, "Fundamentals of Software Engineering, Prentice Hall, 1991.

[12] 최은만, 이금석, 홍영식, "프로그램 이해를 지원하는 소프트웨어 유지보수 도구세트 개발", 한국 정보과학회논문지, 제21권, 제5호, pp. 900-908, 1994. 5.

[13] 이원영, 최은만, "소프트웨어 유지보수 단계에서의 프로그램 이해", 한국정보과학회 소프트웨어 공학연구회, 제8권, 제1호, pp. 85-87, 1995. 3.

[14] 양해술, "테스트 비용의 감소율에 주목한 리뷰 평가척도의 제안과 평가", 한국정보과학회논문지, 제20권, 제6호, pp.821-829, 1993. 6.

[15] 이하용, 이용근, 양해술, "프로그램 디버깅에 의한 오류위치의 결정," 한국정보처리학회 추계학술발표논문집 제3권 2호, pp. 425-430, 1996. 10.

양 해 술

1975년 홍익대학교 공과대학 전기공학과 졸업(학사)
1878년 성균관대학교 정보처리학과 정보처리 전공(석사)
1991년 日本 오사카대학교 기초공학부 정보공학과 소프트웨어공학 전공(공학박사)
1975년~1979년 육군중앙경리단 전자계산실 시스템 분석장교
1986년~1987년 日本 오사카대학교 객원연구원
1980년~1995년 강원대학교 전자계산학과 교수
1993년~1994년 한국정보과학회 학회지 편집부위원장
1994년~1995년 한국정보처리학회 논문지 편집위원장
1994년~현재 한국산업표준원(KISI) 이사
1995년~현재 한국소프트웨어품질연구소(INSQ) 소장
관심분야: 소프트웨어공학(특히, S/W 품질보증과 품질평가, 품질감리, 품질컨설팅, OOA/OOD/OOP, CASE, SI), 소프트웨어 프로젝트관리

이 하 용

1993년 강원대학교 전자계산학과 졸업(학사)
1995년 강원대학교 대학원 전자계산학과 소프트웨어공학 전공(이학석사)
1996년~현재 경희대학교, 경원대학교 공과대학 전자계산공학과 강사
1995년~현재 한국소프트웨어품질연구소(INSQ) 선임연구원
관심분야: 소프트웨어공학(특히, S/W 품질보증과 품질평가, 품질감리, 객체지향 프로그래밍, 객체지향 분석과 설계, CASE)