Journal of Korea Multimedia Society Vol. 7, No. 7, July 2004(pp. 985-997)

플랜정합과 프로그램 실행을 통한 프로그래밍 오류분석에 관한 연구

송종수[†], 임순범[††]

요 약

본 논문에서는 플랜 정합과 프로그램 실행결과를 이용하여 초보자들의 C언어 프로그램을 이해하고 오류를 지적해 주는 프로그래밍 오류 분석시스템을제시하였다. 프로그램 실행 결과를 이용함으로써 프로그래밍 플랜간의 연관관계를 유연하게 표시할 수 있고, 플랜 정합의 차이점이 정확한지를 검증할 수 있게 해주며, 한 플랜의 오류가 연관된 타 플랜에 어떤 영향을 주었는지를 파악할 수 있게 해준다. 플랜 간의 연관관계에 따라 오류의 원인과 파급 효과를 지적하고 예제나 반례에 해당하는 사례를 구체적으로 제시하여, 사용자가 이해하기 쉽게 오류에 대한 설명을 제공한다. 14종의 다양하고 난이도 있는 실습예제에 대해 학생들이 작성한 프로그램에 있는 오류를 분석함으로써 안정되고 신뢰성 있는 오류분석 시스템임을 보여주고 있다.

Automatic Programming-Error Detection by Plan Matching and Program Execution

Jong-Soo Song[†], Soon-Bum Lim[††]

ABSTRACT

In this paper, an automatic programming error-diagnosing system is provided for novice C programmers by plan matching and program execution. Program execution results are used to provide flexibility in describing the relationship between programming plans, to verify the correctness of the plan matching differences, and to detect the influence of a plan's error to the related plan. We can give easy and informative explanations to the students according to a plan's error and the resulting effects to related plans. The students are consulted to check their program's correctness with the given test data. Our error-diagnosing system is tested with student's programs for the 14 various and difficult problems and gives acceptable recognition results.

Key words: Programming Error Detection(프로그래밍 오류 분석), Plan Matching(플랜정합), Program Execution(프로그램 실행)

1. 서 론

지능형 교사 시스템(ITS: Intelligent Tutoring System)은 학생 개인차에 따라 학습내용을 적절하게 선택하여 효과적인 개별학습을 제공하고자 많은 연구가 이루어졌으며[1,2], 학생이 현재 알고 있는 지식의 상태나 능력 및 배경을 나타내는 학습자 모델의 연구에 많은 노력을 기울여 왔다. 컴퓨터 프로그래밍 언어 학습의 경우 프로그래밍 언어와 프로그래밍 기술에 관한 지식을 어떻게 표현할 것인가 하는 전문가 모델에 관한 연구와 더불어 학생이 작성한 프로그램에 있는 잘못된 점을 찾아내는 진단 능력을 어떻게 갖출 것인가에 많은 연구가 있어 왔다. 학생의 상태

※ 교신저자(Corresponding Author) : 송종수, 주소 : 경기도 안양시 동안구 안양우체국 사서함 9호(431-600), 전화 : 031)341-5208, FAX : 031)341-5208, E-mail : jssong@en4n.com

접수일 : 2003년 8월 26일, 완료일 : 2003년 12월 12일

[†] (주)엔포엔

[††] 종신회원, 숙명여자대학교
(E-mail : sblim@sookmyung.ac.kr)

를 파악하기 위해서는 학생이 알고 있는 내용과 무엇을 모르는지를 알 수 있어야 하기 때문이다. 따라서 효과적인 프로그래밍 언어 교육을 위한 지능형 교사 시스템에서는 프로그램의 오류를 파악할 수 있는 능력이 필수적이라 하겠다.

학생 프로그램의 오류파악을 위해 초기에는 특정 테스트 데이터를 가지고 프로그램을 실행시켜서 도중에 얻어지는 결과(behavior)를 바탕으로 오류의 증상(symptom)을 찾아서 오류의 원인을 유추하여 잘못을 지적해 주는 동적인 분석방법론(Dynamic Analysis)이 많았다. 이러한 방법론은 프로그램의 내용을 알지 못해도 증상과 원인에 관한 상관관계를 잘 정리할 수 있으면 효과적인 시스템이 될 수 있으나, 적용범위가 한정이 되고 조금만 복잡한 문제에도 적용할 수 없는 문제점을 안고 있다.[3]

그 후 프로그램의 이해(understanding)라는 관점에서 오류를 지적해 주는 시스템들이 많이 등장하였다. 이들은 각자의 프로그램 이해 방법론에 입각하여 초보자의 프로그램을 분석하고, 해석이 되어지는 프로그램은 맞는 프로그램으로 받아들이고 해석이 되지 않는 부분을 오류로 판정하고 그 원인을 찾는데 초점을 맞추었다. 여기서는 일반적인 방법론으로 가능한 넓은 범위의 프로그램을 이해하도록 지식표현이나 추론에 관한 연구가 제시되었고, 이를 바탕으로 해석되지 않는 부분을 오류로 판단하였는데, 이러한 오류를 어떻게 판단할 것인가와 그 오류에 대해 어떻게 적절한 조언을 제공할 것인가에 연구의 초점이 맞추어져 있었다.

TALUS[4]는 리스프 언어를 위한 프로그램 분석 시스템으로, 입력 프로그램을 리스프의 핵심형태(core dialect)로 단순화 시키고, 이를 상징적 계산(symbolic execution)을 통해서 입력 프로그램과 모델 프로그램간의 차이를 구하여 오류를 알아내고, 오류의 원인을 찾고자 증명법과 경험칙(heuristics)을 활용하였다. 이 시스템의 장점은 프로그램의 논리적 구조에 기반 해서 프로그램을 분석하고, 로직 트리 끝단에 있는 코드들 간의 차이점을 찾아내므로, 차이점들이 프로그램 전체적인 것이 아닌 말단의 부분적인 경우에 이 방식은 모델과 입력 프로그램간의 차이를 명확하게 지적할 수 있다. 그러나 프로그램이 TALUS의 모델 프로그램에서와 동일한 형태로 구성된 경우만을 분석할 수 있어서 다양한 형태의 여러 개의 서브함수로 구성된 큰 프로그램을 해석하는 데는 어려움을 갖고 있다. 게다가 TALUS는 제한된 자료 구조만 처리하는 관계로 크기가 크거나 명령문 스타일의 프로그램은 처리하는데 어려움이 있다.

CAMUS[5]는 자동화된 역 공학(reverse engineering) 방법을 이용해서 프로그램을 분석하고 있다. 문제정의서와 같은 특정한 사전 지식이 없이 프로그램을 분석한다. 그 결과는 학생의 프로그램이 무엇을 하고 있다는 상위 레벨의 프로그램 정의서 정도로 분석이 나온다. 정답 프로그램 역시 동일한 방식으로 분석이 된다. 이 두개의 상위 레벨의 정의서를 비교해서 학생에게 주는 피드백을 생성하는 것이다. 상위 레벨로 추상화하는 과정이 프로그래밍 지식을 활용하는 인식론 적인 방법이긴 하지만, 상위 레벨의 정의서가 종종 대단히 복잡해지는 문제점을 안고 있다. 30~40라인 정도의 코드 프로그램에 대해서도 정의서가 지나치게 복잡하다는 단점을 보였던 것이다[6].

PROUST[7]는 파스칼 언어를 위한 프로그램 분석시스템으로, 정형화된 프로그래밍 코드인 플랜을 이용하여 다양한 형태의 학생 프로그램을 해석할 수 있다. 학생이 해결해야 하는 문제 정의서에 있는 목표(goal)를 기반으로 학생 프로그램을 플랜들의 결합으로 재구성하여 분석한다. 플랜상의 코드와 학생 프로그램 코드간의 차이점을 찾아내서 그 차이점이 오류 목록에 있는 경우 오류로 판정하는 플랜정합 방법론의 원조가 되는 시스템이다. 목표와 플랜에 의한 지식표현에 따라 다양한 형태의 알고리즘을 쉽게 추가할 수가 있으며, 문제 정의서를 이용하여 프로그램이 무엇을 하고자 하는지를 알고, 오류의 원인 분석 및 설명에 이를 적절히 활용하고 있다.

PROUT 이후의 연구들은 플랜 표현의 일반성을 높이고, 정합 과정을 좀더 형식화하는 데 초점이 맞추어졌다. DUDU[8]는 플랜 내에 코드패턴뿐만 아니라 플랜이 구현하고자하는 목표가 구현되기 위한 연관 관계를 표시함으로써, 플랜 내 특정 어떤 부분이 구현되어야 할 핵심 파트인지를 알려주게 된다. 이를 이용해서 코드 패턴과 약간의 차이에 대해서는 플랜 자체의 해석 과정에서 흡수할 수 있는 장점이 있다.

UNPROG[9]는 학생 프로그램을 계층적인 그래프 형태로 바꾸어서 구조화되지 않은 프로그램의 콘트롤 개념을 이해하는 시스템이다. 그래프에는 데이터 흐름과 콘트롤 흐름을 모두 나타내고 있다. 변형된 표현에서 특징적인 형태의 작은 단위의 서브 그래프

를 찾아낸 후에, 동일한 그래프 형태와 조건이 기술된 프로그래밍 플랜과 정합을 통해서 플랜을 인식하게 된다. 주된 장점은 콘트롤 흐름을 잘 인식한다는 점이라 하겠다. 플랜 정합법의 범주에 들어가는 방법론들이 지식표현과 플랜을 인식하는 과정을 일반화하기 위해 여러 가지 방법론을 제시하고 있지만, 근본적인 한계점은 플랜을 인식하는 과정이 형식화된 방법이 아니기에 경험적인 지식을 필요로 한다는 점과, 이렇게 경험적 지식을 이용하여 해석된 결과가 최적의 해석이라는 보장이 없다는 문제점을 공통적으로 안고 있다.

프로그래밍 언어의 특성에 따른 변화를 극복하고 좀더 형식화된 지식표현과 인식방법에 초점을 맞추어 연구가 진행되면서 그래프 파싱(Graph parsing)에 의한 프로그램 해석과 오류의 자동분석 방법론이 등장하였다. 프로그래밍 언어의 차이점을 흡수할 수 있고, 경험적 지식에 근거하지 않고 프로그램 자체를 파싱하는 방법에 의해 자동적으로 상위의 플랜단위의 패턴을 인식하는 방법론이다. 제대로 구현된 다양한 형태의 정답 프로그램을 인식하는 데는 큰 효과를 발휘하고 있으나, 형식적인 문제는 없으나 그래프 파싱이 완료되지 않는 오류 부분을 어떻게 인식하고 오류의 원인을 찾아 줄 것인지에 대한 연구가 아직 미진한 상태이다.[10,11]

본 연구에서는 기존에 제시된 해석방법론 중에서 C/C++언어와 같은 절차적 언어(Procedural Language)에 적합한 플랜정합법을 기반으로, 초보자를 위한 오류검증 기능의 보완에 초점을 맞추었다. 기존의 방법들이 프로그래밍 지식을 바탕으로 학생이 작성한 프로그램 코드만을 분석하는 데 반해, 테스트 데이터를 이용하여 학생 프로그램을 실행시킨 결과를 활용하였다. 프로그램 자체를 실행시켜 얻은 동적인 정보를 활용하여, 프로그램 해석과 오류분석의 효율성을 높이고 초보자들이 이해하기 쉽도록 구체적이고 실례에 바탕을 둔 피드백을 제공하는 오류분석 시스템을제시하고자 한다.

2. 플랜인식과 프로그램 실행의 필요성

2.1 프로그래밍 플랜

사람들이 프로그램을 어떻게 해석하는 지에 대한 그 동안의 인지론적 연구에 따르면 프로그램을 해석하는 데 있어서 전문가들은 전형적이고 기본적인 패턴을 이용한다는 것이다. 정형화된 코드 블록인 프로그래밍플랜(Plan)을 이용하며, 이는 전문가들의 프로그램에서도 자주 볼 수 있다는 것이다[12,13]. 그림 1의 예제 프로그램 A를 살펴보도록 하자. 3번 줄과 8번 줄의 코드는 (밑줄이 그어진 코드) 'sum'이라는 변수가 0으로 값이 초기화된 후에 'rain'이라는 변수의 값들이 계속 더해지는, 특정 변수의 합계를 구하는 전형적인 패턴임을 알 수 있다.

또 다른 예제로 5번, 6번, 11번 코드는(이태릭체로 기울여 쓰여진 코드) 반복적으로 입력 값을 받아들이는 전형적인 코드 패턴이다. 5번 줄의 초기 입력문에서 값을 읽어 들인 후에 6번 줄의 반복문에서 특정 값인지 비교하여 종료할 것을 결정하는 형태이며 반복문 내에 추가적인 입력문이 있어서 (11번 코드) 계속해서 입력 값을 읽어 들이는 등, 종료 값이 주어질 때까지 입력 값을 받아들이는 코드의 전형적인 패턴이라 할 수 있다.

프로그램을 이해하는 데 있어서 경험이 많은 전문가들은 코드 내에 있는 이러한 몇 줄의 전형적인 코드들을 찾아내고, 이를 바탕으로 프로그램의 전체적인 디자인이 어떻게 되어 있는 지를 파악하게 된다. 코드 패턴들이 어떻게 결합되고 연결되는 가를 바탕으로 전체의 구조를 재구성함으로써 프로그램을 해독하는데, 이 과정에서 사용되는 가장 기본이 되는 전형적인 코드 패턴을 '프로그래밍 플랜(plan)'이라

```
1  main() {
2     int valid=0, rainy=0 ;
3     float   rain, sum=0.0, max=0.0, mean ;

4     printf ("₩n Amount of rainfall : ₩n") ;
5     scanf ("%f", &rain) ;
6     while (rain != 99999) {
7        valid = valid + 1 ;
8        sum = sum + rain ;
9        if (rain > 0.0)   rainy = rainy + 1 ;
10       if (max < rain) max = rain ;
11       scanf ("%d", & rain ) ;
12    }
13    mean = sum / valid ;
14    format ("The average rainfall : %f₩n", mean) ;
15    format ("The number of rainy day : %d₩n", rainy) ;
16    format ("The number of valid inputs : %d₩n",valid)
17    format ("The maximum rainfall : %f₩n", max) ;
   }
```

그림 1. 프로그래밍 플랜이 나타난 예제 프로그램 A

부르는 것이다.

2.2 다양한 형식의 플랜간 결합 관계

프로그래밍 플랜들이 결합되는 방식은 다양하며 경우에 따라서는 동일한 플랜을 사용하는데도 결합되는 방식의 차이 때문에 구성이 달라 보이기도 한다. 그림 2에 있는 예제프로그램 B는 예제 프로그램 A와 합계를 구하는 방식이나 입력 값을 받아들이는 반복문으로 'while'문을 쓰는 것 까지 동일한 명령문을 사용하고 있으나 형태가 달라 보이는 느낌을 준다. 두 프로그램의 차이점은 프로그램 A가 합계를 구하는 코드와 입력 값을 받아들이는 코드가 합쳐져서 최적화 되어 있는 반면에, 프로그램 B는 각기 나누어져 있어서, 구성이 다르다는 차이점이 있다.

입력 값을 받아들이는 반복문에서 'while'문 대신에 'do-while'문을 쓰는 경우에는 비교하는 부분이 반복문 아래에 있는 관계로 구조가 달라질 것이며, 'goto'문을 사용하는 경우에는 또 다른 구조의 프로그램이 만들어 질 것이다. 그림 3과 4는 이러한 예제를 보여주고 있다. 그런데 프로그램 전체적으로는 형태와 구조가 여러 가지임에도 불구하고 합계를 구하는 플랜은 4개의 프로그램에서 모두 동일한 명령문으로 나타나고 있다(그림 1~4에서 밑줄 친 코드). 또한 종료 값을 만날 때까지 입력 값을 받아들이는 부

```
main() {
    int    i, rainday, validin ;
    float  mean, amount, sum, max, value[100] ;

    printf ("How much rain fell a day?₩n") ;
    scanf ("%f", &amount) ;
    while (amount<0) {
        scanf("%f", &amount) ; }
    validin = 0 ;
    while (amount != 99999) {
        value[validin++] = amount ;
        scanf ("%d", &amount ) ;
    }     /* number of rainy day */
    for (i=0, rainday=0; i<validin; i++ ) {
        if (value[i] != 0) rainday++ ;
    }
    printf("The number of rainy days : %d", rainday) ;
        /* average rainfall per day */
    for (i=0; i<validin; I++ ) {
        sum += value[i] ;
    }
    mean = sum / validin ;
        . . .
```

그림 2. 플랜이 분산된 예제 프로그램 B(부분)

```
float   rain, sum=0.0, max=0.0, mean ;

do {
    printf ("₩n Amount of rainfall : ₩n") ;
    scanf ("%f", &rain) ;
    if (rain == 99999) break ;
        . . .
    sum = sum + rain ;
}   while (rain != 99999)
```

그림 3. do-while문을 사용한 예제 프로그램 C(부분)

```
float    rain, sum=0.0, max=0.0, mean ;

retur:
    printf ("₩n Amount of rainfall : ₩n") ;
    scanf ("%f", &rain) ;
    if (rain<0) goto retur ;
    if (rain != 99999) {
        n++ ; . . .
        sum = sum + rain ;
        goto retur ;
    }
```

그림 4. goto문을 사용한 예제 프로그램 D(부분)

분도 약간의 차이는 있으나 4개의 프로그램에서 대체로 비슷한 패턴으로 구성되어 있음을 알 수 있다.

이처럼 동일하거나 비슷한 플랜을 사용하면서도 전체적인 외관상으로 차이가 있어 보이는 여러 개의 프로그램을 해독하고 프로그램에 포함된 오류를 지적하기 위해서는, 프로그래밍 플랜을 찾아내는 것뿐만 아니라 플랜들 간의 다양한 결합관계를 표현하고 실제 결합된 형태를 파악하는 능력이 중요해진다. 즉, 플랜간의 결합관계를 파악하면서 오류도 찾아내야 하는 것이 프로그램 오류분석 자동화의 큰 과제인 것이다.

2.3 실행 결과를 활용한 플랜간 관계 표현 방법

본 연구에서는 플랜 간의 결합 형태를 별도로 지정하지 않으면서도, 문제 자체에서 정의된 오브젝트의 활용을 통해서 플랜간의 연관관계를 파악할 수 있는 표현방식을 제시하고자 한다. 그림 1의 예제 프로그램 A에 대해 프로그래밍 플랜을 분석해낸 결과를 바탕으로 설명하도록 하겠다. 이 프로그램에서 3가지의 프로그래밍플랜을 찾아낼 수가 있으며, 그 결과가 그림 5에 주어져 있다.

특정한 종료 값이 주어질 때까지 입력을 받아들이

는 "Sentinel-ReadProcess-While Plan", 합계를 구하는 "Running-Total Plan" 및 평균값을 구하는 "Average Plan" 3가지이다. 각각의 플랜에 대한 표현식이 그림 5의 왼쪽에 나타나 있다. 플랜 표현에서 "**Template**"항목에 플랜에 대응되는 코드가 주어져 있으며, 이 코드를 프로그램에서 찾아서 해당 플랜을 파악했음을 화살표로 나타내 주고 있다.

이 3개의 플랜은 문제의 정의에 의하면 공통의 오브젝트를 사용하는 연관관계에 있다. 종료 값이 주어질 때까지 입력 값을 받아들여서 이들의 평균을 구하는 프로그램으로, 이들 3개의 플랜은 '입력 값'이라는 공통의 오브젝트를 처리하고 있는 것이다. 입력 값을 받아들이는 'Sentinel-ReadProcess-While Plan'에서는 이 오브젝트를 '?New'로 정의하고 있다. 이 플랜은 입력 값을 받아들이되 종료 값이 주어지는 경우 끝낸다는 플랜의 의도에 따라서, ?New라는 오브젝트에 대해서 "입력 값(:input-data)"이면서 "종료 값(?Stop)은 배제 한다"는 2가지 조건을 생성하게 된다. 이처럼 플랜에서 생성된 특정한 조건들은 '***ObjectConds***:' 항목으로 표현되고 있다.

평균을 구하는 'Average Plan'은 코드는 나눗셈 문장 하나이지만, 합계를 구하는 'Sum'이라는 서브목표와 연계되고, "합계"를 구현하는 한 예제인 'Running-Total Plan'과 연관성을 지니게 된다. 따라서 합계를 구하는 플랜의 ?Total과 평균 값 플랜의 ?Sum은 동일한 오브젝트를 나타낸다는 연관성이 주어지게 된다. 이들 2개의 플랜은 모두 '입력 값'에 대해서 합계와 평균을 구하는 플랜이다. 따라서 입력 값을 구하는 플랜과 연관성을 지니게 되는 것이다.

이러한 플랜간의 연관관계는 사용자 프로그램에서 동일한 변수를 사용하여 이들 플랜들이 구현되어 있는 경우 이를 쉽게 파악해 낼 수 있다. 그림 5의 경우, 'rain'이라는 동일한 변수에 대해서 입력 값 플랜과 합계를 구하는 플랜에서 같이 사용되고, 'rain' 값이 더해지는 'sum'이라는 변수가 평균을 구하는 플랜에서 사용되는 것을 통해서 플랜 사이에 연관관계가 있음을 파악할 수 있다.

그런데 그림 2에 있는 예제 프로그램 B의 경우는 입력 값을 받아들이는 플랜에서는 변수 'amount'를 사용하고, 합계를 구하는 플랜에서는 다른 변수 'value[i]'를 사용하고 있다. 그러나 'value[i]'의 값들을 살펴보면, 입력으로 주어진 값이면서 종료 값이 아님을 실행 결과를 이용하여 알 수 있다. 테스트 데이터를 가지고 프로그램을 실제 실행시키면서 'sum'에 더해지는 'value[i]'값이 "입력 값"인지 "종료 값"과 다른 지를 판정할 수 있기 때문이다. 이처럼 프로그램의 변수 이름만이 아니라 변수 값의 변화를 감안해서 실제적으로 공통의 오브젝트를 처리하는 지를 판단할 수 있게 되는 점이 프로그램 실행결과를 감안한 플랜 표현 방식의 장점이라 하겠다.

프로그램 실행 결과를 이용하지 않고 플랜간 연관관계를 표현할 수 있는 방법에는 크게 2가지가 있다. 첫째는 여러 개의 플랜이 결합된 형태를(복합 플랜을) 하나의 새로운 플랜으로 등록하는 방법이다. 이

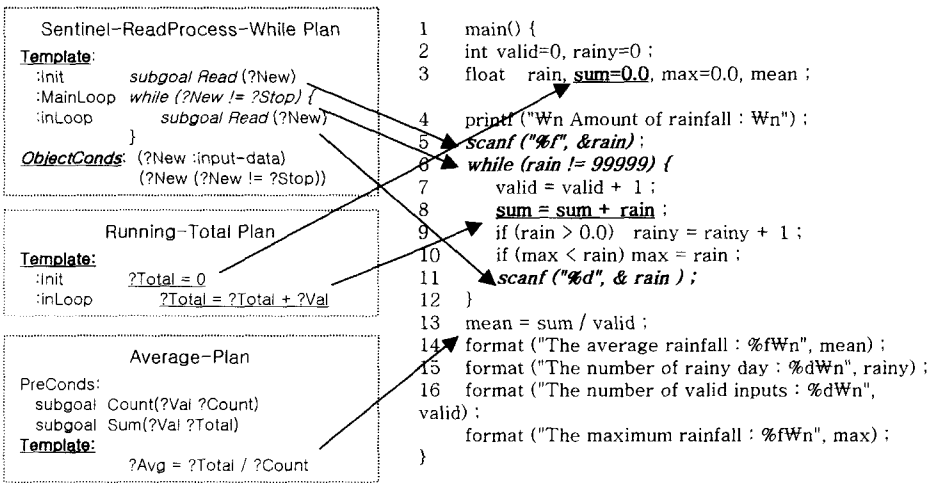

그림 5. 예제 프로그램 A에 대한 플랜 정합 결과

경우의 문제점은 추가적으로 등록해 주어야 할 복합 플랜의 개수가 기하급수적으로 증가한다는 점이다. 입력 값을 받아들이는 플랜의 개수와 합계를 구하는 플랜의 개수의 곱으로 추가할 플랜이 늘어난다. 이는 2개의 플랜이 결합된 경우의 수이며, 여러 개의 플랜이 결합되는 경우에는 추가할 플랜이 기하급수적으로 증가하게 될 것이다.

플랜정합을 이용한 프로그램 해석의 대표격인 PROUST[7]는 개별적인 플랜을 바탕으로 위치를 제한하여 플랜간의 결합관계를 표현하고 있다. 그림 5의 프로그램을 예로 들면, 입력 값을 받아들이는 프로그램의 반복문 내에 합계를 구하는 플랜이 위치하도록 지정하는 방식을 통해서 여러 가지 플랜간의 결합관계를 해석해 내고 있다. 이 경우 그림 2의 예제 프로그램 B와 같이 합계를 구하는 플랜이 입력 값을 받아들이는 반복문 밖에서 구현된 경우를 적합한 결합 관계로 판단하지 못하게 된다. 또한 프로그램의 변수 이름만을 바탕으로 플랜에서 사용되는 오브젝트에 대한 연관성을 따지는 관계로 변수 이름은 달라지지만 실제적으로 동일한 오브젝트를 표현하는지를 알 수 없게 된다. 게다가 'goto'문을 사용하는 프로그램은 분석에서 아예 제외를 하는데, 이는 프로그램 소스만을 이용해서 파악할 수 있는 정보에 한계를 있기 때문이라 할 수 있다.

2.4 새로운 플랜 표현방식의 장점

본 논문에서 플랜 내에 프로그래밍 코드를 포함한다는지, 학생 프로그램과의 코드 정합을 통해서 플랜을 인식한다는 지, 문제정의서를 활용하는 등의 플랜 정합법의 기본 아이디어는 PROUST 시스템의 사례를 활용하고 있다. 프로그래밍 실행 결과를 이용하여 새로운 플랜 표현 방식을 제안하고 있다. 이를 이용하여 정합과정의 차이점을 실행 결과로 검증하고, 오류의 원인과 그 여파를 예제를 제시하여 구체적으로 지적할 수 있는 장점이 나타나게 되는 데, 이는 3장에서 예를 들어 설명하도록 하겠다.

플랜간 결합 관계를 나타내는 새로운 표현 방식을 도입할 경우의 예상되는 장점은 크게 3가지를 들 수가 있다. 첫째로는 다양한 형태의 플랜 간 결합 관계를 인식 가능하다는 점이다. 그림 4의 예제에서 살펴본 바와 같은 'goto'문을 포함한 특수한 형태의 프로그램도 해석이 가능하게 된 것이다. 따라서 개별적인 플랜의 표현에 영향을 미치는 일이 없이 PROUST와 같이 위치 정보를 활용한 표현 방식보다 적은 수효의 플랜으로 보다 많은 플랜 간의 결합관계를 해독할 수 있다는 장점이 있다.

둘째로는 플랜 간의 연관관계가 좀 더 명확하게 표현됨으로써 이를 이용한 오류의 설명도 보다 구체적이고 직접적으로 가능해 졌다는 점이다. 위의 예에서 입력 값 검사 플랜과 합계 플랜 간에는 공통의 오브젝트를 매개로 해서 인과 관계가 성립한다는 것을 새로운 표현 방식에서는 알 수 있다. 이를 이용하면 입력 값 검사 플랜의 해석 과정에서 오류가 발견되고 합계 플랜의 해석과정에서 종료 값이 더해지는 덧셈의 연산문에 사용된 것을 확인하게 되면, '입력 값 검사의 오류로 인하여 합계 계산에 잘못된 값이 사용되었다'라는 설명이 가능하게 된 것이다. 즉, 인과관계에 의해 구체적인 사례를 바탕으로 오류의 원인을 설명할 수 있게 된 점이 장점이라 하겠다.

셋째로는 플랜 해석 과정의 효율성이라 하겠다. 새로운 표현방식에서는 플랜간의 결합관계를 직접 표현하는 일이 없으므로, 개별 플랜의 해석에 전념하게 되며 타 플랜과의 결합 관계는 공통으로 사용하는 변수의 정합성을 따져서 판단하게 되므로 탐색 공간이 기하급수적인 아닌 프로그램의 크기에 비례하게 되어 해석의 효율성이 증대된다 하겠다.

3. 실행 결과를 이용한 프로그래밍 오류 분석

플랜정합 방법론에 프로그램 실행 결과를 결합시켜서 학생 프로그램에 대한 오류 분석의 과정이 어떻게 진행되는 지를 구체적으로 설명하고자 한다. 그림 6의 예제는 매일 매일의 강우량을 읽어서 평균값과 최대값을 포함한 여러 가지 결과를 출력토록 요구하는 '강우량 측정 문제'에 대해 학생이 제출한 프로그램이다. 이 프로그램은 대부분의 학생들이 입력 값을 읽어 들이는 반복문 내에서 합계나 최대 값의 계산이 이루어지게 작성한 것과는 달리, 입력 값을 읽어서 배열에 저장해 두었다가 문제에서 요구하는 목표들 각각에 대해 따로따로 처리하고 있다. 여러 목표를 한꺼번에 효율적으로 처리하지 못하고 있지만 초보자 입장에서 하나씩 문제를 처리해 나가는 전략으로 나름대로 효과적인 아이디어라고 할 수 있다.

강우량 문제: 노아는 방주를 띄울 날자를 결정하

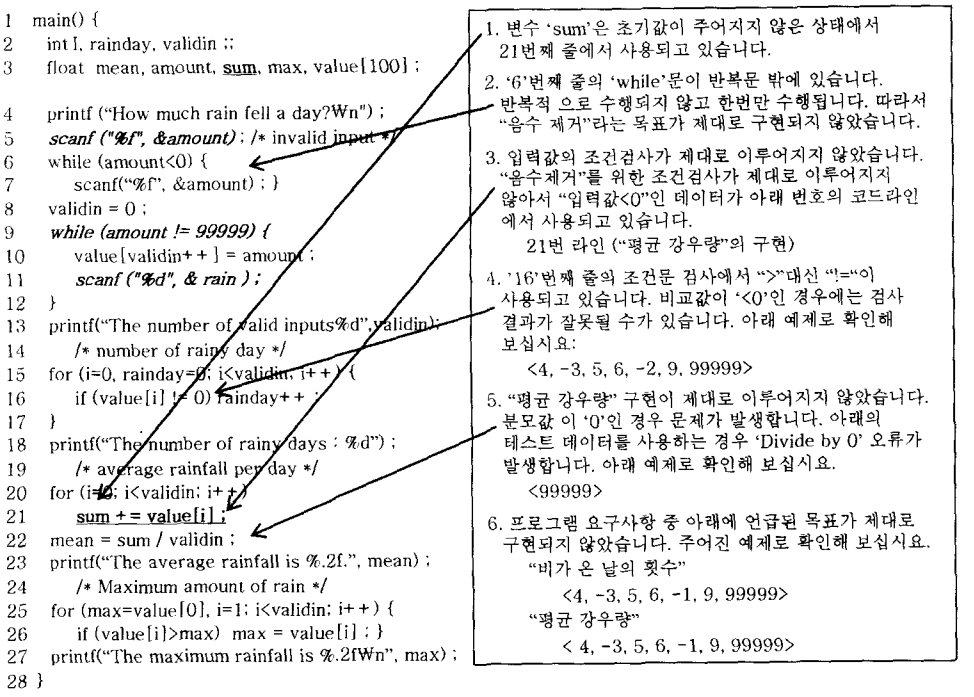

그림 6. 강우량 문제 프로그램에 대한 분석 결과

기 위해 매일매일의 강우량을 기록하고자 한다. 이를 위한 C프로그램을 작성하라. 매일매일의 강우량을 읽어 들이되 입력 값의 끝임을 나타내는 '99999'가 입력되면 끝내도록 하며, 이 값은 계산에서 제외한다. 음수 값이 입력되면 거부하고 다시 입력토록 작성하여야 한다. '타당한 숫사가 입력된 날의 횟수', '비가 온 날의 횟수', '평균 강우량', '최대 강우량'을 출력하는 프로그램을 작성하라.

그림 6에서는 이 예제 프로그램에 대한 오류 분석 결과를 제시하고 있는데, 크게 3가지로 나누어 분류할 수 있다. 첫째는 1번의 초기화하지 않은 변수를 참조하는 오류나 5번의 0으로 나누는 오류와 같이 프로그램 실행 과정에서 발생하는 오류를 설명해 주는 것이다. 둘째는 주어진 목표 자체에만 해당하는 메시지와 목표와 목표간의 연관관계를 나타내어 설명하는 메시지로 나눌 수가 있다. 2번 4번 5번의 메시지는 '음수 제거'와 '평균값 계산'이라는 하나의 목표 자체에 대한 분석과정에서 찾아진 오류의 내용을 보여주고 있다. 그에 반해 3번째 메시지로 주어진 내용은 '음수 제거'라는 목표가 제대로 구현되지 않은 결과가 다른 목표인 '평균값 계산'에 미치게 된 여파에 대해 설명을 해주고 있다.

3.1 문제정의서의 활용

그림 6에 있는 오류 메시지를 보면 "음수 제거"나 "평균 강우량"과 같이 문제에서 요구하는 내용이 자연어로 명확히 표기되어 있고, 4, 5, 6번 메시지의 경우 구현되지 않는 목표의 내용을 지적하고 예제 데이터를 이용하여 테스트해 볼 것을 권고하고 있다. 이는 학생이 프로그램에서 구현해야 할 목표들이 어떠한 조건을 만족해야 하는 지가 명기되어 있는 문제정의서가 입력 자료로 주어지기 때문이다. 그림 7은 강우량 문제에 대한 문제정의서의 내용으로, 'Loop-Input- Validation'과 'Average' 목표를 표현하는 문장에서 "음수 제거" 및 "평균 강우량"이라는 목표 이름이 주어진 것을 이용하여 메시지가 제공되는 것이다.

여기에서 'Loop Input Validation' 목표에 있는 '?DailyRain<0'이라는 파라메터는 DailyRain이라는 오브젝트에 대한 조건을 나타내고 있다. 이러한 조건은 오브젝트의 속성에 저장되는데, 이를 통해서 이 오브젝트를 이용하는 다른 모든 목표에도 같은 조건이 부과되는 효과가 있다. 이처럼 오브젝트에 부과되는 조건을 이용하여 목표들 간의 인과관계를 지정할 수 있다. 즉, 음수제거(Loop-Input-Validation) 목표

```
DefProgram Rainfall
DefObject   ?DailyRain ObjectClass Scalar
DefResultVar     ?Mean, ?ValidDays, ?RainyDays, ?Max
DefGoal     Sentinel-Controlled Input Sequence (?DailyRain, 99999, "최종입력값 검사", "입력 값=99999")
DefGoal     Loop Input Validation (?DailyRain, ?DailyRain<0, "음수제거", "입력값<0")
DefGoal     Output (Bind ?Mean = Average (?DailyRain, "평균 강우량"))
DefGoal     Output (Bind ?ValidDays = Count (?DailyRain, "유효한 입력의 횟수"))
DefGoal     Output (Bind ?RainyDays = Guarded-Count (?DailyRain, ?DailyRain>0, "비가온 날의 횟수"))
DefGoal     Output (Bind ?Max = Maximum (?DailyRain, "최대 강우량"))
TestData    7, 2, 6, 4, 8, 3, 99999      →   4, 6, 6, 8
            8, 0, 5, 4, 0, 3, 6, 99999   →   4, 7, 5, 8
            4, -3, 5, 6, -1, 9, 99999    →   6, 4, 4, 9
            -3, 4, 0, 5, 8, -1, 99999    →   3.4, 5, 3, 8
            99999                        →   :Undefined, 0, 0, :Undefined
```

그림 7. 강우량 문제에 대한 문제정의서

는 DailyRain이라는 오브젝트에 대한 속성을 제한하는 조건을 제공하는 원인적인 목표이며, 평균강우량(Average) 목표는 이러한 조건이 붙은 오브젝트를 사용하는 목표이므로 둘 사이에는 인과관계가 형성이 된다. 이를 이용하여 '음수 제거'가 제대로 구현되지 않아서 '평균 강우량'을 구하는데 영향을 미쳤다는 그림 6의 3번 메시지를 제시할 수 있게 된다.

테스트 데이터 예제 역시 문제정의서를 이용하여 표현하고 있다. 'TestData'문은 입력 값으로 주어지는 예제 데이터의 값을 나타내고 있고, 'DefResultVar' 문에 의해 선언된 오브젝트들이 갖는 최종 값에 대한 결과를 나타내고 있다. 즉, 3번째 테스트 데이터 (4 3 5 6 1 9 99999)가 입력 데이터로 주어진 경우에는 (?Mean ?ValidDays ?RainyDays ?Max)의 순서대로 결과값이 (6 4 4 9)임을 보여준다. 이를 이용해서 특정 테스트 데이터에 대해서 특정 목표가 제대로 구현되지 않았음을 추정할 수가 있으며, 이를 바탕으로 그림 6의 마지막 메시지를 제공할 수 있게 된다.

3.2 프로그램 실행

프로그램의 실행은 문제 정의서에서 주어진 테스트 데이터를 이용하여 프로그램에서 입력을 요구하는 경우 입력 값으로 제공하면서 실제로 프로그램을 실행하게 된다. 프로그램 실행의 중간과정에서 발생되는 변수 값의 변화라든지, 출력되는 값, 조건식의 결과 등이 모두 저장된다. 이 과정에서 발생하는 오류들도 모두 저장된다.

예를 들면, 초기화되지 않은 값의 참조라든지, 주어진 테스트 데이터 이상으로 입력을 요구한다든지, 나눗셈의 분모가 0이라든지, 배열의 범위를 벗어나는 인덱스를 사용한다든지 등, C언어 실행 시 오류 이외에도 초보자에게 조언을 하는데 유용하게 제시할 수 있는 실행상의 오류에 대해서도 모두 저장해 두고 있다. 그림 6의 5번 메시지는 실행상의 오류가 특정 목표의 오류 설명에 포함되어 제시된 경우이며, 1번 메시지는 특정 목표에 포함되지 않는 관계로 일반적인 오류에 대한 별도의 항목으로 사용자에게 설명된 경우이다.

프로그램 실행이 끝난 후 예상 결과값으로 주어진 값을 프로그램내의 특정 변수가 일관되게 갖게 되는지를 통해서 문제 정의서에 제시된 목표들이 제대로 구현되었는지를 먼저 점검한다. 예로 위에서 언급한 (4 3 5 6 1 9 99999)를 입력 값으로 실행한 후에 예상 결과값인 (6 4 4 9)의 값을 갖는 프로그램 내 변수를 찾게 되는 것이다. 모든 테스트 데이터에 대해서 프로그램에 있는 max라는 변수가 최대 강우량에 해당하는 ?Max의 예상 결과값과 항상 일치하는 결과가 나오는 것을 근거로, '최대 강우량' 목표가 제대로 구현이 된 것으로 판단한다. 플랜 정합을 수행할 때 이러한 목표부터 먼저 시작한다. 이러한 목표들이 플랜 정합에서 오류가 없는 정확한 정합이 이루어질 가능성이 높기 때문이다.

?Max와 달리 다른 목표의 경우는 테스트 데이터에 따라 부분적으로 예상 값과 일치하는 결과값을 갖는 변수가 나타나게 된다. 이러한 부분적인 변수 연결 정보는 나중에 플랜 정합 과정에서 후보를 검토할 때 평가 자료로 이용하게 되고, 최종적인 판정 후에는 목표가 제대로 구현되지 않은 테스트 데이터 사례를 선택하는데 이용하게 된다.

3.3 플랜 정합과 오류 판정

플랜정합은 오류분석의 핵심부분으로 문제 정의서에서 주어진 목표들이 학생 프로그램에 제대로 구현되었는지를 플랜내의 'Template' 항목에 주어진

코드와의 정합(matching)을 통해서 분석한다. 플랜 정합의 과정은 문제 정의서에 주어진 목표에 대하여 그 목표에 소속된 모든 플랜에 대해 학생 프로그램과 코드를 비교하여 가장 차이점이 적은 플랜이 구현된 것으로 판단한다. 정확하게 구현 되지 않는 경우에는 그 차이점을 기록하고, 버그 목록에 이러한 차이점이 포함되어 있는지 (즉, 예상 가능한 오류인지를 따지고), 프로그램 실행결과를 이용해서 이러한 차이점이 타당한 지를 검증하게 된다.

그림 6에 있는 4번 오류메시지는 '비가 온 날의 횟수'에 대한 분석 결과이며, 양수인 값의 횟수를 계산하는 'Guarded-Counter-Plan'에 대해 예제 프로그램을 정합한 결과이다. 이는 플랜 자체에 한정된 정합의 오류에 대한 판정 결과로 이를 판정하는 과정이 그림 8에 나타나 있다. 플랜 정합의 결과, 입력 값이 0보다 큰지를 검사하는 16번의 if문에서 '>0'대신에 '!=0'을 사용하는 차이점이 나타났다. 이 차이점은 버그 목록에 포함된 관계로 예상 가능한 오류로 판정하고, 16번 조건문이 실행될 때의 실제 value[i] 값을 가지고 '>0'과 '!=0'을 실행해서 그 결과가 같은지를 비교하게 된다. 문제 정의서에 있는 세 번째 테스트 예제(4 3 5 6 1 9 9999)에 대해서 2번째 (value[1] 값 3)와, 5번째(value[4]값 1) 입력 값에 대해 '>0'과 '!=0'의 조건검사 결과가 같지 않음이 밝혀 졌다.

이를 바탕으로 사용자 프로그램에 조건 검사식의 오류가 있으며 테스트 데이터를 통해 문제가 있음을 확인하고, 그 사례를 이용하여 사용자에게 4번 오류 메시지를 제공하게 된 것이다. 이는 개별 플랜에 한정된 오류에 대하여 정합상의 차이점을 실행 결과를 통해서 검증한 경우이며, 5번 오류 메시지도 동일한 과정을 통해서 판정하게 된 결과이다.

플랜정합 단계에서도 한 플랜에만 연관되는 개별적인 오류에 대해서는 원인과 그 파급효과가 확인되기도 하지만, 타 플랜에까지 영향을 미치는 오류에 대해서는 정합 과정에서 판단하기가 어렵다. 따라서 정합이 끝난 후 개별 플랜별 오류를 파악하여 해당 플랜의 개별적인 오류를 정하고, 플랜간에 연관된 오류의 내용과 그 여파에 대해 별도로 판정해서 학생에게 제시할 오류 메시지를 작성하게 된다.

'음수 제거'를 구현하는 플랜에 대한 정합 결과 그림 6의 2번 메시지와 같이 음수를 제거하는 비교문이 반복문 내에 있지 않다는 정합의 차이점이 발생하고, 프로그램 실행 결과도 1번만 실행되는 것이 확인되어 플랜 단독적인 오류로 판정하게 된다. 그런데 이 플랜에서는 "?DailyRain < 0"을 배제하는 조건을 생성하고, 21번 줄의 합계를 구하는 'Running-Total Plan'은 이 오브젝트를 참조하는 데 "value[i]<0"인 사례가 발견된 것이다. 이처럼 조건을 생성하는 플랜의 정합에 오류가 발생하였고, 그와 연관된 플랜에서 조건에 어긋난 잘못된 변수 값이 사용된 경우에 3번 오류 메시지가 제시되게 된다. 그림 9는 이러한 오류를 판정하는 과정을 정리하여 보여주고 있다.

마지막으로 문제 정의서에 예상 결과값이 주어지는 오브젝트를 변수로 포함하는 플랜에 대해서는 예상 결과값과 실제 프로그램 실행 후 값을 비교하여 제대로 구현되었는지를 판단할 수 있다. 이를 이용하여 정합 과정에 오류가 없더라도 결과 값이 예상 값과 다르게 나온 경우의 테스트 예제를 가지고 학생 스스로 검토해 보도록 조언을 제시할 수가 있다. 그림 6의 6번 메시지는 이러한 경우의 예제를 보여주고 있다.

4. 테스트 결과 분석

프로그램 실행 결과와 플랜정합을 활용한 프로그램 분석 방법론의 활용 가능성을 판단하기위해, 초보

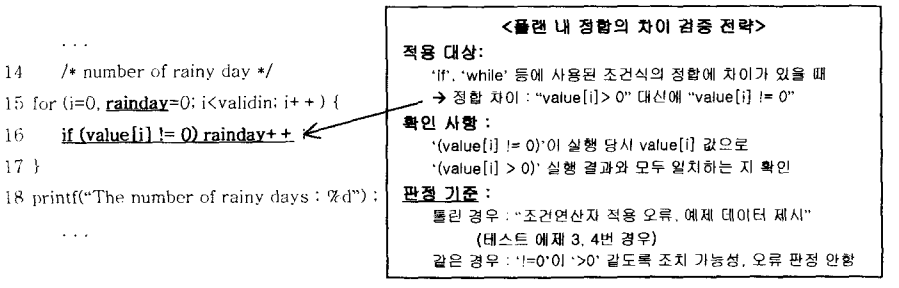

그림 8. 조건문 검사와 관련된 플랜 자체 오류에 대한 판정 전략

<연관 플랜 간 정합의 차이 검증 전략>

적용 대상:

변수의 조건을 매개로 인과 관계가 있는 플랜 간에, 조건을 가하는 플랜의 정합에 차이점이 있을 때

→ 'Loop Input Validation'에서 '?DailyRain'에 조건을 가하면서, 정합에 차이점이 있고

확인 사항 :

'?DailyRain'을 파라메타로 받는 플랜의 계산에서 조건에 위배되는 값이 계산에 사용됐는지 검사

→ 'Running-Total Plan'에서 조건에 위배되는 값 '-3 < 0'가 연산에 사용되었다.

판정 기준 :

예제 발견 경우 : 연관 플랜 간 오류의 전파 설명토록 메시지 제공

→ **"음수 제거" 조건 검사 오류가 "평균 강우량" 구현에 영향을 미치는 문제 발생**

예제 발견하지 못하는 경우 : 오류 판정 안함.

그림 9. '평균 강우량'과 관련된 연관 플랜 간 오류에 대한 판정 전략

자용 C언어 오류분석 시스템인 ExBug(Execution-guided deBugger)를 구현하고, 다양한 프로그래밍 실습 문제에 대해 학생들이 실제로 작성한 프로그램에 대해 적용하였다[14]. '강우량 문제'를 포함하여 '알기 쉬운 C언어'[15] 교재에 있는 연습문제 등 총 14가지에 대해서 예제 프로그램 397개를 수집하여 테스트 하였다.

이 문제들에는 문자열 처리라든지, 숫자의 조건 및 변형 처리에 대한 문제, 출력문 위주의 문제 등 다양한 문제를 포함하고 있다. 프로그램의 형태는 'For/while'문을 사용한 단순한 반복문에서 'switch-case'를 사용한 구조라든지, 숫자 문자를 정수로 바꾼다든지, 문자 및 정수 배열(array) 사용, 복잡한 조건식이나 산술식이 포함된 프로그램을 가지고 테스트하였다. 수집된 프로그램은 프로그램 전문가가 수작업으로 검토하여 오류의 내용을 정리하였으며, 동일한 프로그램을 ExBug에 실행 시켜 얻은 결과와 비교하여 표 1에 정리하였다.

프로그램 397개 중에서 90%에 해당하는 357개의 프로그램에 대해 ExBug는 분석을 정상적으로 완료하였다. 이에는 'goto'나 'case'문 및 배열의 사용을 포함한 다양한 형태의 프로그램이 포함되어 있다. 그러나 10%에 해당하는 40개 프로그램에 대해서는 분석을 완료하지 못했는데, 학생 프로그램이 무한 루프에 걸려서 끝나지 못한 경우나 서브루틴을 사용하는 경우 및 'getchar()'를 이용하여 불필요한 입력을 추가로 받을 건지를 물어보는 경우 등 이었다.

분석에 성공한 357개의 프로그램 중에서 오류가 없이 제대로 작성된 프로그램의 수효는 128개 이며, 오류를 포함하고 있는 것은 229개 였으며 이를 수작업으로 분류한 결과 오류의 수효는 총 679개 였다. 이중 81%에 해당하는 552개에 대해서 제대로 지적해 냈으나, 19%인 127개에 대해서는 지적을 하지 않았다. 틀리지 않았는데 틀렸다고 하거나, 오류의 내용을 엉뚱하게 지적하는 허위경보(False alarm)의 수효는 36개가 있었다.

잘못된 메시지를 제공하는 허위경보의 사례는 학생이 변수를 잘못 사용한 여파로 다른 플랜이 구현되지 않은 것으로 판정한 경우와, 합계 계산에 종료 값을 포함시켰다가 나중에 빼주어서 실행상에 문제가 있어 보이게 한 경우, 'case'문에서 범위 값(예: 60~69) 대신에 10으로 나눈 몫을 검토한 경우 등의 특이한 형태로 플랜을 구현한 경우가 많았다. 몇 가지는 보완이 필요한 부분도 있으나, 대체로 특수한 형태의

표 1. ExBug 테스트 결과

총 프로그램 수		397	
	분석을 중단한 프로그램 수	40	10.1%
	분석을 완료한 프로그램 수	357	89.9%
	오류가 없는 정확한 프로그램 수	128	
	오류가 있는 프로그램 수	229	
총 오류의 수 (229개 프로그램의 오류 합계)		679	
	지적이 된 오류의 수	552	81.3%
	지적되지 않은 오류의 수	127	18.7%
	허위경보의수	36	

프로그래밍사례로 개선의 필요성이 급하지 않다고 판단된다. 플랜정합을 이용한대표적인 프로그래밍 오류분석 시스템인 PROUST의 사례에 비추어 허위 경보 발생율이 많이 낮은 결과를 보여 주고 있다.

오류 판정 율이 80%대로 크게 높아 보이지 않는 요인은 몇가지로 분석할 수가 있다. 첫째는 구현되지 않은 목표가 있다고 지적하면서 테스트 데이터로 검사할 것을 권고하는 (그림 6의 6번 메시지와 같은) 경우는 정확히 오류 내용을 지적한 것이 아닌 관계로 오류를 지적한 것으로 평가하지 않았다. 이를 포함하는 경우 지적율은 90%이상이 된다. 둘째로는 if문의 조건식에서 값이 같은지 검토하는 '=='라는 비교연산 대신에 실수로 값을 바꾸는 '=' 연산을 사용하는 경우처럼 정합상의 오류는 확인했으나 이를 실행결과와 연계하여 판정할 수 있는 방안이 아직 구현되지 않아서 지적하지 못하는 경우가 있었다. 또한 문자열을 출력하는 문제에 있어서, 문자열의 마지막을 의미하는 '\0'을 포함시키지 않아서 실행상의 오류는 발생이 되었으나 정합상의 오류를 명확히 찾아내지 못한 경우도 있다.

플랜정합에서의 차이점은 있으나 이를 검증하는 실행결과가 없거나, 실행결과 검사에서 문제점은 있으나 플랜정합에서 문제가 없으면 오류를 확정치 않는 오류판정 전략이 오류를 지적하지 않는 큰 요인이라 하겠다. 이러한 보수적인 판정전략의 배경은 플랜이 정확히 일치하지는 않았으나 사전에 문제가 되는 데이터를 제거해서 제대로 구현되어 있을 가능성이 있고, 실행 결과가 이상하게 보이고 있지만 이것이 다른 플랜에서 문제가 발생해서 그 여파의 결과일 가능성이 있기 때문이다. 이러한 경우도 대체적으로는 특정 테스트 데이터에 대한 결과값이 정확하지 않는 경우에는 예제를 주고 테스트 해 볼 것을 권고하고 있는 관계로 사용자 입장에서는 나름대로 문제점을 지적하는 효과는 있다고 판단된다.

정합상의 차이점만 있거나 실행과정의 오류만 있는 경우도 모두 피드백으로 제시하도록 사용자의 선택에 따라 오류의 가능성을 적극적으로 제시해 주도록 하는 것도 개선책의 한 방안이라 판단된다. 이를 통해 사용자가 테스트 데이터를 이용해 결과가 틀린 것을 확인하고, 모든 오류의 가능성에 대해 사용자로 하여금 검토해 볼 수 있는 기회를 줄 것이다.

5. 결론 및 향후 연구 과제

본 논문에서는 플랜간의 다양한 결합관계를 유연하게 표시하고 플랜 정합의 효율성을 기하고자 프로그램 실행 결과를 결합시키는 아이디어를 제시하였다. 프로그램 실행 결과를 이용하여 플랜 간의공통 오브젝트에 대한 조건이 만족되었는지를 조사함으로써 플랜간의 연관관계를 확인할 수 있게 되었다. 플랜 정합 과정에서 발생하는 차이점을 실행 과정의 중간 값을 이용하여 확인해 볼 수 있으므로, 프로그램 자체의 정보로 뒷받침된 오류 검증이 가능하게 된 것이다. 다양한 플랜간의 연관관계를 유연하게 표시할 수 있으면서 분석 과정의 효율성과 효과를 높인 프로그램 오류 분석 방법이라 하겠다.

사용자에게 제공하는 오류에 대한 설명 메시지는 플랜간의 연관 관계를 활용하여 오류의 원인과 그 여파를 구체적으로 지적해 줄 수 있으며, 예제와 반례를 제시하여 설명함으로써 초보자가 이해하기 쉬운 설명을 제공해 주고 있다.

ExBug는 프로그램 실행 기능을 결합시키기는 하였으나 플랜 정합에 프로그램 코드를 그대로 사용하는 관계로 플랜정합의 기본적인 한계점은 동일하게 안고 있는 상태이다. 형태적 변화를 흡수하지 못하고 코드만 차이가 있는 비슷한 플랜을 추가하는 문제점을 안고 있다 (조건식이나 복잡한 계산식의 경우 전처리를 통해 표준화를 이루고 있으나 형태적인 변화를 모두 흡수하지는 못하고 있다). 따라서 그래프 파싱(graph parsing)과 같이 좀더 형식화된 플랜 해석 방법론을 결합하는 것이 필요한 상태이다. 문제정의서가 사전에 필요한 관계로 일반 교사들이 이용하기 어려운 문제점을 해결하기 위해 정답 프로그램으로부터 문제 정의서를 자동적으로 만들어 주는 시스템을 개발하였으며[16], 이를 ExBug와 결합하여 일반 사용자가 효과적으로 이용할 수 있도록 개선할 예정이다.

ExBug는 학생이 작성한 프로그램의 오류를 파악함으로써 학생이 무엇을 모르는 지에 대한 학습자 정보를 제공해 줄 수 있다. 따라서 프로그래밍 플랜에 대한 지식을 활용하여 전문가 모델 및 효과적인 교수법과 연계하여 통합적인 지능형 교사 시스템으로의 발전이 가능하다. 특히 테스트 데이터를 이용해서 예제를 제시할 수 있는 기능을 잘 활용하고 사용

자와의 상호 대화형 인터페이스에 대한 최근의 연구 결과를 활용할 경우[17], 분석된 프로그램의문제점을 암시하고서 예제를 가지고 실습을 하면서 스스로 해결책을 찾아보도록 하는 일종의 '소크라테스식' 교사 시스템으로의 발전이 가능할 것이다.

참 고 문 헌

[1] J. Self, "The defining characteristics of intelligent tutoring systems research: ITSs care, precisely". International Journal of Artificial Intelligence in Education, 10, pp. 350-364, 1999.

[2] D. Sleeman and J. Brown (eds), Intelligent Tutoring Systems, Academic Press, New York, 1982.

[3] E. Shapiro, Algorithmic Program Debugging. MIT Press, MA., 1982.

[4] W. Murray, Automatic Program Debugging for Intelligent Tutoring Systems. Morgan Kaufmann, San Mateo, Calif., 1988.

[5] P. Vanneste, A reverse engineering approach to novice program analysis. Ph.D. dissertation, KU Leuven Campus Kortrijk (KULAK), Holland, 1994.

[6] K. Bertels, P. Vanneste, and C. De Baeker, "A cognitive approach to program understanding". In Proc. The Working Conference on Reverse Engineering, Baltimore, Md, pp. 1-7, 1993.

[7] W. Johnson, Intention-Based Diagnosis of Novice Programming Errors. Morgan Kaufmann, Los Altos, Calif., 1986.

[8] D. Allemang, Understanding programs as devices, Technical Report, Ohio State University, 1990, Ph.D. Thesis.

[9] J. Hartman, Automatic control understanding for natural programs, Technical Report AI91-161, University of Texas Austin, 1991, Ph.D. Thesis.

[10] L. Wills, "Flexible control for program recognition", in Proc. The Working Conference on Reverse Engineering, (Baltimore Maryland), pp. 134-143, 1993.

[11] S. Kim, Algorithm Recognition for Programming Tutoring, Ph.D. dissertation, KAIST, Daejeon, 1994.

[12] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge", IEEE Transactions on Software Engineering, vol. 10, pp. 595-609, 1984.

[13] S. Letovsky, "Cognitive Processes in Program Comprehension", in Empirical studies of Programmers: Second Workshop, Ablex Publishing, 1987.

[14] 송종수, "플랜정합과 프로그램 실행에 의한 학생 프로그램 오류분석에 관한 연구", KAIST 박사학위논문, 대전, 1998.

[15] 이광형, 알기 쉬운 C-언어, 홍릉출판사, 서울, 1991.

[16] S. Hahn and J. Kim, "Automatic problem description from model program for knowledge-based programming tutor". Automated Software Engineering, Vol.4 No.4, pp439-461, 1997.

[17] A. Grasser, K. VanLehn, and D. Harter, "Intelligent Tutoring Systems with Conversational Dialogue", AI Magazine 22(4) Winter 2001, pp. 39-52, 2001.

송 종 수

1982년 서울대학교 계산통계학과(학사)
1987년 서울대학교 계산통계학과(석사)
1998년 한국과학기술원 전산학과(박사)
1983년~1998년 (주)삼보컴퓨터 기술연구소 부장
1998년~2001년 (주)핸디소프트 기술연구소 수석연구원
2001년~현재 (주)엔포엔 대표이사
관심분야 : 인공지능, 지능형교사시스템(ITS), e-Learning, 임베디드시스템

임 순 범

1982년 서울대학교 계산통계학과(학사)
1983년 한국과학기술원 전산학과(석사)
1992년 한국과학기술원 전산학과(박사)
1989년~1992년 (주)휴먼컴퓨터 창업 / 연구소장
1992년~1997년 (주)삼보컴퓨터 프린터개발부 부장
1997년~2001년 건국대학교 컴퓨터과학과 교수
2001년~현재 숙명여자대학교 멀티미디어과학과 교수
관심분야 : 컴퓨터 그래픽스, 웹 멀티미디어 인터페이스, 전자출판 (폰트, XML, 전자책, e-Learning)