論文
4-10-2

효과적인 프로그래밍 오류분석을 위한 지식표현연구 (A Study of Knowledge Representationfor Effective Programming Error Detection)

송 종수 (Jong-Soo Song)*
송 두헌 (Doo-Heon Song)**

요 약

프로그래밍 초보자의 개인차에 따라 효과적인 개별학습을 제공하는 프로그래밍 언어 교사 시스템 구축을 위해서는 초보자의 프로그램에 있는 오류를 분석하고 그 원인을 파악할 수 있어야 한다. 본 연구에서는 플랜 정합과 프로그램 실행에 기반한 효과적인 프로그래밍 오류분석 시스템을 제안하고자 한다. 프로그램 실행 결과를 이용하여, 연관된 프로그래밍 플랜간의 연결 관계를 유연하게 표시하고, 플랜 정합의 차이점을 검증하며, 플랜간 공유변수의 조건에 따른 인과관계의 파악할 수 있게 된다. 또한 초보자에게 제공하는 오류 메시지는 플랜 간의 인과관계에 따라 오류의 원인과 그 파급 효과를 지적하고 예제나 반례에 해당하는 사례를 구체적으로 제공할 수 있게 되어, 사용자가 쉽고 명확하게 이해할 수 있게 제공이 가능해졌다.

ABSTRACT

Automation of programming-error detection is an important part of intelligent programming language tutoring systems. In this paper, a new programming error detection approach for novice programmers is proposed by plan matching and program execution. Program execution result is used to resolve the restricted programming plan representation and to provide a confirming evidence for the plan matching differences. By checking the values of shared variable between the related plans, we can detect the cause-effect relationship between the plans. With this relationship and the test data, we can explain the program's unexpected behaviors according to the bug's cause and resulting effects.

* 정회원 :㈜엔포엔 대표이사
**종신회원 :용인송담대학
컴퓨터소프트웨어과 교수

논문접수 : 2003. 9. 2.
심사완료 : 2003. 9.30.

1. 서론

컴퓨터 프로그래밍 언어 학습에서는 프로그래밍 언어와 프로그래밍 기술에 관한 지식을 어떻게 표현할 것인가 하는 전문가 모듈에 관한 연구와 더불어 학생이 작성한 프로그램에 있는 잘못된 점을 찾아내는 진단 능력을 어떻게 갖출 것인가에 많은 연구가 있어 왔다. 학생의 상태를 파악하기 위해서는 학생이 알고 있는 내용과 무엇을 모르는지 상태를 파악할 수 있어야 하기 때문이다. 따라서 효과적인 프로그래밍 언어 교육용 컴퓨터 지능교사시스템에서는 학생 프로그램의 오류를 파악할 수 있는 능력이 필수적이라 하겠다[1].

초창기 프로그래밍 오류 분석 시스템 중에는 특정 테스트 데이터를 가지고 프로그램을 실행시켜서 도중에 얻어지는 결과(behavior)를 바탕으로 오류의 증상(symptom)을 찾은 후에 오류의 원인을 유추해서 학생의 잘못을 지적해 주는, 전형적인 진단에 의한 오류를 찾는 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 있었다[2].

그 후 프로그램의 이해(unders- tanding)라는 관점에서 오류를 지적해 주는 시스템 들이 많이 등장하였다. 이들은 각자의 프로그램 이해 방법론에 입각하여 초보자의 프로그램을 분석하여 해석이 되어지는 프로그램은 맞는 프로그램으로 받아들이고 해석이 되지않는 부분을 오류로 판정하고 그 원인을 찾는데 초점을 맞추었다. 따라서 가능한 넓은 범위의 프로그램을 이해할 수 있는 방법론을 제시하고 어떻게 효율적인 오류지적이 가능한 가에 연구의 초점이 있었다. 여기서는 일반적인 방법론으로 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[3-5].

근래에는 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 [6-7].

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 방법들에서 사용된 학생 프로그램과 프로그래밍 지식 외에 추가로 제공되는 테스트 데이터로 프로그램 자체를 실행시켜 얻은 중간 결과라는 다이나믹한 정보를 활용하여, 프로그램 해석과 오류분석의 효율성을 높이고 초보자들이 이해하기 쉽게 구체적이고 실례에 바탕을 둔 설명을 제공하는 오류분석 방식을 제안하고자 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 전형적인 기본 패턴, 즉 정형화된 코드 블록인 프

로그래밍 플랜(Plan, Clich)을 이용한다는 것이며, 이는 전문가들의 프로그램에서 자주 볼 수 있다는 것이다[3][9]. 컴퓨터 프로그램이 비록 순서적인 명령문의 나열이지만 일정한 역할을 담당하는 단위들을 바탕으로 위/아래의 깊이와 옆으로의 넓이를 갖는 계층적인 네트워크 구조를 갖는다는 분석인 것이다[4]. 사람들의 머릿속에는 계층적인 구조에 의해서 프로그램이 구성되어 있고 이러한 네트워크의 노드들은 정형화된 플랜의 형태로 구현이 되어지며, 이러한 플랜을 단위로 해서 프로그램을 작성할 뿐만 아니라 해독하는 과정에도 사용하고, 이러한 해독과정도 무작정 밑에서부터 위로 올라간다기 보다는 프로그램에 주어진 문제의 명세에 근거하여 나름의 가설을 세우고 이에 근거하여 특정 플랜들을 찾아가는 과정을 밟는 다는 것이 전문가들이 프로그램을 인식하는 패턴이라 하겠다.

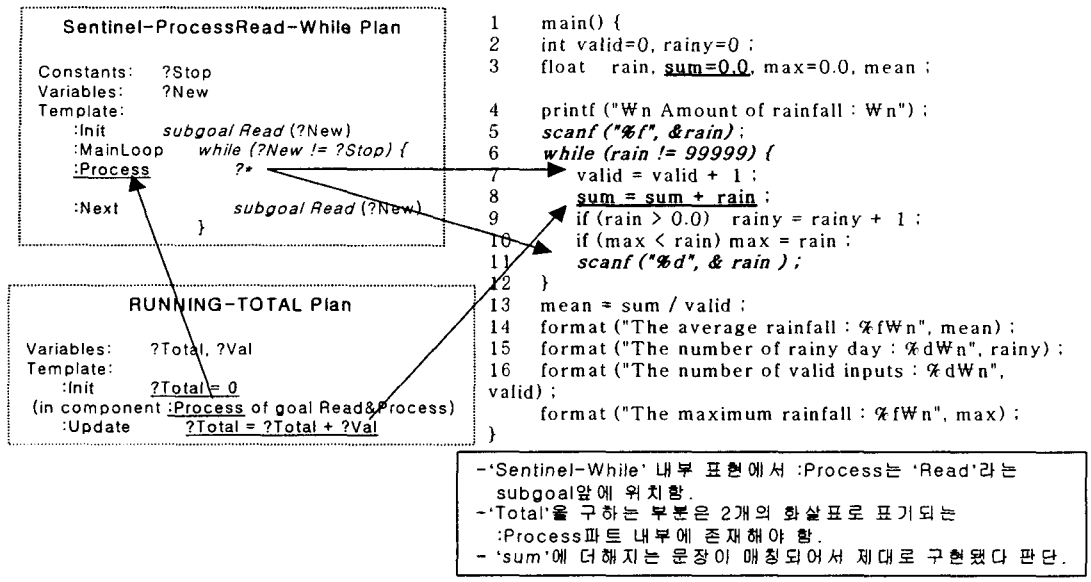
이러한 프로그래밍 플랜의 예는 아래의 그림 1을 통해 쉽게 살펴볼 수 있다. 3번 줄과 8번 줄의 코드는 (밑줄이 그어진 코드) 'sum'이라는 변수가 0으로 값이 초기화된 후에 'rain'이라는 변수의 값들이 계속 더해지는, 특정 값의 합계를 구하는 전형적인 패턴임을 알 수 있다. 또 다른 예제로 5번, 6번, 11번 코드는 (이태릭체로 기울여 쓰여진 코드) 반복적으로 입력 값을 받아 들이는 전형적인 코드 패턴의 일부분이다. 5번의 초기 입력문에서 값을 읽어 들인 후에 6번의 반복문에서 특정 값인지를 비교하여 종료할 것인지를 결정하는 형태이며, 반복문 내에서 (11번 코드) 추가적인 입력문이 있어서 계속해서 입력값을 읽어 들이는 전형적인 형태라 하겠다.

논문 [10]에서는 그림 1의 예제와 같이 반복 입력문과 입력값의 합계를 구하는 코드가 포함되면서도 구조와 형태가 크게 달라 보이는 경우 4가지를 제시하고 있다. 그림 1과 형태는 비슷하나 구조가 다른 프로그램, 'do-while'문을 이용하여 형태가 약간 다르나 구조는 비슷한 프로그램, 'goto'문을 사용하여 형태와 구조가 모두 다른 프로그램을 제시하여 학생들의 경우 다양한 형태로 프로그램을 작성하는 것을 보여 주고 있다. 이처럼 프로그램의 형태와 구조가 다양함에도 불구하고 합계를 구하는 플랜의 형태는 4개의 프로그램에서 모두 동일한 형태를 취하고 있었으며 (그림 2에서 3번과 18번의 밑줄친 코드), 특정 값을 만날 때까지 입력값을 받아 들이는 부분도 약간의 차이는 있으나 예상이 가능한 패턴들로 4개의 프로그램에서 모두 찾아볼 수 있었다. 이처럼 동일하거나 비슷한 플랜들을 사용하면서도 외관상 전체적인 구조의 차이가 보이는 여러 개의 프로그램을 해독하고 프로그램에 포함된 오류를 지적하기 위해서는 프로그래밍 플랜을 찾아내는 것 뿐만 아니라 플랜들 간의 가능한 결합관계를 표현하고 실제 결합된 형태를 파악하는 능력이 중요해진다. 즉, 플랜간의 결합관계를 파악하면서 오류도 찾아내야 하는 것이 프로그램 오류분석 자동화의 큰 과제인 것이다.

'프라우스트(PROUST)'는 플랜간 결합관계를 적극적으로 이용하여 프로그램을 인식하는 플랜정합법의 원조가 되는 오류분석 시스템으로, 프로그램 코드를 직접 사용하여 플랜을 표현하고 있으며 이러한 코드를 패턴 매칭에 의해 인식하고 있다. 플랜의 위치 관계를 지정하는 연결사를 활용하여 플랜간의 가능한 결합관계를 표현하고 이를 이용하여 예상되는 플랜의 위치를 알고 빠른 프로그램 해석에 적극적으로 활용하였다. 플랜 내 매칭의 오류를 플랜간 위치 관계와 연결지어 사용자에 효과적인 메시지를 제공하는 데도 활용하고 있다[8].

그림 1은 프라우스트가 합계를 구하는 플랜과 특정 값이 주어질 때까지 입력값을 받는 플랜을 이용하여 프로그램을 해석하는 예를 보여주고 있다. 주목할 점은 합계를 구하는 'RUNNING-TOTAL Plan'의 경우, 합계를 구하는 부분(:Update라는 제목의 덧셈문)에 "(in component :Process of goal *Read&Process*)" 라는 타 플랜과의 연결관계를 지시하고 있는

이러한 구체적이고 직접적인 위치관계의 표



[그림 1] 예제 C 프로그램의 플랜 매칭 결과 (부분)

[Fig. 1] Plan matching result of sample C program (partial)

데, 특정값이 주어질 때 까지 입력을 받는 플랜의 입력문('while'문) 내에서 (8~10번 문장 범위 내에서) 덧셈문을 찾게 된다는 것이다. 즉, 반복하여 입력값을 받아 들이는 플랜(Sentinel-While Plan)을 먼저 해석하게 되면 합계를 구하는 플랜이 나타나게 될 위치를 사전에 미리 알 수 있게 된다는 것이다.

이처럼 프라우스트는 플랜간의 위치관계를 이용하여 플랜과 플랜이 어떻게 결합될 수 있는지를 표현하고, 이를 이용하여 관련된 플랜이 구현되어 있을 위치가 어디인지를 알게 됨으로써 프로그램 해석이 효율적으로 진행될 수 있는 장점을 갖고 있다. 또한 플랜과 플랜간의

위치 관계가 주어지므로 어느 한 플랜의 구현에 있어 오류가 있는 경우, 그 여파가 타 플랜에 미치게 되는 것을 알 수 있다. 이러한 플랜간 관계를 이용하여 효과적인 오류 메시지를 제공하는 장점을 얻게 된 것이다.

현은 위에서 언급한 장점을 갖고 있지만 동시에 여러 가지 단점을 내포하게 된다. 첫째로 해석 가능한 프로그램의 형태가 제약을 받게 된다는 점이다. 즉 복잡한 구조를 구체적으로 표현할 수 있다는 장점과 형태가 제약된다는 단점이 동시에 나타나고 있는 것이다. 두 번째 문제점은 플랜간에 위치에 따른 연관관계를 갖게 됨으로써, 한 플랜의 해석은 타 플랜의 해석 결과가 주어져야 끝낼 수가 있으며, 이는 연관된 플랜의 해석과 연계 해석해야 하므로 연관된 플랜의 깊이와 넓이에 따라 검토해야 할 탐색 공간의 크기가 기하급수적으로 증가할 가능성을 안고 있다. 세번째 문제는, 연관된 플랜들의 해석에 있어서 초기에 내린 결론이 잘못되어진 경우 그 후의 플랜들의 해석에 오류가 전파될 가능성이 크다는 점이다. 따라서 초기 선정된 플랜의 해석에 신중을 기해야 하며, 가능한 여러 가지 해석이 나타나지 않아

야 탐색공간을 줄이며 일관된 해석이 가능하게 된다.

이러한 이유로 프라우스트에서는 프로그램의 전체 골격을 나타내는 특징적인 플랜부터 먼저 해석하는 전략을 취하고 있다. 이는 초기 플랜 해석이 정확해야 하며, 만일 정확히 일치하거나 전형적인 오류의 패턴으로 확인되지 않는 경우에는 뒤로 갈수록 문제점이 파급되어 오류가 발생할 가능성이 대단히 커지게 되는 문제점을 안게 된다.

2.2 실행 결과를 활용한 새로운 플랜간 관계 표현 방법

플랜 간의 위치정보를 직접적으로 표현하는 것은 장점과 단점을 동시에 내포하고 있는데, 이러한 위치정보를 활용하면서 단점을 제거할 수 있는 방법은 없는가? 이를 위해서는 과연 위치정보에 의해 유지된 내용이 무엇인지, 이를 유지하면서 단점을 없앨 수 있는 추가적인 정보는 없는 지를 먼저 살펴보도록 하자.

그림 1의 플랜 정합 결과에서 사용된 위치정보("in component ?)의 의미는 연관된 플랜이 정확하다고 확인이 된 위치에서 구현이 된다는 것을 보장하고 있다. 즉, 'RUNNING TOTAL Plan'의 합계 구하는 덧셈문의 경우 연관 된 플랜인

'Sentinel-ReadProcess-While Plan'에 의해 입력 값을 받아 들이고 있는 것이 보장된 영역 내에서(8~11번 코드) 적합한 코드가 있는 지를 찾으라는 것이다. 결국 8번 코드와 같이 특정 변수('rain') 합계를 구하는 연산문을 찾는데, 그 특정 변수가 입력문에 의해 입력 값을 나타내야 한다는 것이다. 이를 위해 플랜간 위치관계를 제한해서, 입력을 받아들이는 반복문 내의 특정 영역(8~11번 코드)에서만 비슷한 연산문을 찾도록 하여 제대로 된 결과를 효과적으로 찾자는 것이 프라우스트의 플랜 표현의 의도라 하겠다. 본 연구에서는 플랜 간의 위치관계를 대신하여 연관관계를 나타내주는 방안으로 두 플랜 사이에 공통으로 사용되는 변수가 취하는 값을 검토하는 조건 검사를 대신 사용하자는 것으로, 그림 1의 형태와 대비하여 요약하여 표현하면 그림 2와 같다.

그림 2에 나타난 예로 설명을 하면,

'Sentinel-ReadProcess-While Plan'에서 입력된 값을 나타내는 변수 '?New'는 'RUNNING TOTAL Plan'의 '?Val'이라는 변수와 동일하며, 입력값 플랜에서 종료값 '99999'가 아니라는 조건이 제시되고, 'subgoal Read'에 의해 입력 값이라는 조건이 주어졌다고 할 수 있다. 따라서 합계를 구하는 연산문의 해석과정에서 더해지는 변수의 값이 입력값인지, '99999'는 아닌 지를 살펴보아 두 플랜 간의 조건 검사가 유지 됐는지를서는 덧셈 연산문의 더해지는 값에 대해 입력값인지, 특정한 값은 아닌지를 검사할 수 있는 방법이 필요하다. 프로그램을 실행하면서 중간 결과를 저장해 놓으면 이를 이용해서 필요 시점에서 더해지는 값들을 살펴보아 판단할 수 있을 것이다. 프로그램 실행 결과가 필요한 이유는 여기에 있다.

즉, 연관된 플랜이 나타나야 할 위치를 지정하는 대신에 2개의 연관된 플랜 간에 지켜야 할 조건이 준수되었는지를 검사하는 것으로 대치하자는 것이다. 플랜 간의 조건을 표현하는 방법은 공통으로 사용하는 변수에 대해 한쪽은 조건을 부과하며, 다른 쪽에서는 그 조건이

제대로 유지되었는지 저장된 프로그램 실행 결과를 이용하여 검사함으로써 결합 관계가 제대로 되었는지를 판단하자는 것이다.

먼저 제기된 조건이 제대로 구현되었는지 여부를 검사할 수 있는 방법이 제시되어야 할 것

이다. 이를 위해서는 테스트 데이터로 프로그램을 실행하면서 중간 값을 저장해 두었다가, 필요 시 조건이 만족되었는지 여부를 검사할 수 있어야 할 것이다. 그림 1의 예제를 가지고 설명을 한다면, 합계를 구하는 플랜의 덧셈문이 8번 라인의 덧셈문과 일치하는 지를 검토할 때, 실제로 8번 라인이 실행되면서 더해지는 'rain'변수의 중간값이 모두 저장되어 있어

야 하고 그 값들이 모두 종료값 '99999'는 아

```
Template:
  :Init          subgoal Read(?New)
  :MainLoop      while(?New != ?Stop) {
  :Process           ?*
  :Next              subgoal Read(?New)
                 }

Template:
  :Init          ?Total = 0
     (in component :Process of 'Read&Process'
  :Update        ?Total = ?Total + ?Val
```

으로 보다 많은 플랜 간의 결합관계를 해독할 수 있

```
Sentinel-ReadProcess-While Plan
Template:
  :Init          subgoal Read(?New)
  :MainLoop      while(?New != ?Stop) {
                     ?*
                     subgoal Read(?New)
                 }
ObjectConds:  (?New, (?New != ?Stop))

   RUNNINT TOTAL Plan
Template:
  :Init          ?Total = 0
  :inLoop        ?Total = ?Total + ?Val
```

- 2 플랜간의 위치 관계 제거
  "Running Total Plan"의 덧셈문에서 반복문 내에 (:inLoop), 자체적으로 필요한 위치 정보

- 2 플랜간에는 변수를 공유 (?New와 ?Val)
  'Sentinel Plan'에서는 ?New에 조건을 가하고 (?New != 99999)
  'Total Plan'에서는 ?Val을 파라메타로 받아 들여서
  ?Val에 부과된 조건이 만족된 값들로 연산이 이루어 지는지 검사한다.

"플랜간 위치 정보" ⟶ **"공통변수간 조건 검사"**

[그림 2] 새로운 플랜간 연결관계 표시의 예

[Fig 2] New representation of plan relationship for Sample C program

닌지 검사할 수 있어야 한다. 이를 위해서 프로그램이 실행이 가능하여 하고, 주어진 테스트 데이터를 바탕으로 프로그램 실행 루틴에서 변수 값의 변화와 조건문에서 검토되는 변수들의 중간 값이 모두가 저장되어져야 할 것이다.

그러면 플랜간 결합 관계를 나타내는 새로운 표현 방식을 도입할 경우의 예상되는 장점은 무엇인가? 첫째로는 다양한 형태의 플랜간 결합 관계를 인식 가능하다는 점이다. 그림 1의 예제 뿐만 아니라 'goto'문과 같이 특수한 형태의 경우에도 실질적으로 반복문임을 파악해서 해석이 가능하게 된 것이다. 따라서 합계 플랜의 표현에 영향을 미치는 일이 없이 위치 정보에 의한 방식보다 적은 수효의 플랜다는 장점이 있다.

둘째로는 플랜 간의 인과관계가 좀 더 명확하게 표현됨으로써 이를 이용한 오류의 설명도 보다 구체적이고 직접적으로 가능해 졌다는 점이다. 위의 예에서 입력값 검사 검사 플랜과 합계 플랜 간에는 공통변수를 매개로 해서 인과 관계가 성립한다는 것을 새로운 표현 방식에서는 알 수 있다. 이를 이용하면 입력값 검사 플랜의 해석 과정에서 오류가 발견되었고 합계 플랜의 해석과정에서 종료값이 더해지는 연산에 사용된 것을 발견하면, '입력값 검사의 오류 결과로 합계 계산에 잘못된 값이

사용되었다'라는 설명이 가능하게 된 것이다. 즉, 위치 관계를 고려해서 오류 전파 가능성이 있다고 설명하는 것보다는 인과관계에 의해 구체적인 사례를 바탕으로 오류의 원인을 설명할 수 있게 된 점이 장점이라 하겠다.

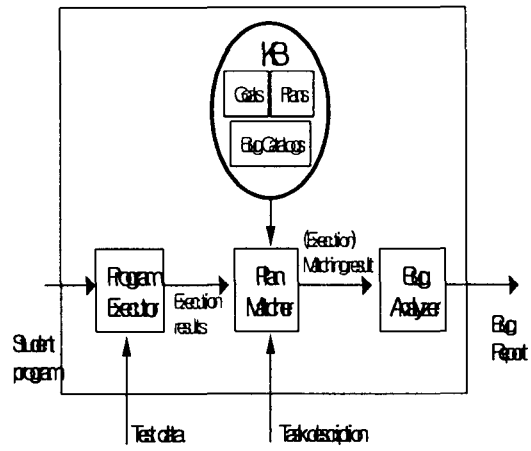
셋째로는 플랜 해석 과정의 효율성이라 하겠다. 새로운 표현방식에서는 플랜간의 결합관계를 직접 표현하는 일이 없으므로, 개별 플랜의 해석에 전념하게 되며 타 플랜 과의 결합관계는 공통으로 사용하는 변수의 정합성을 따져서 판단하게 되므로 탐색 공간이 기하급수적인 아닌 프로그램의 크기에 비례하게 되어 해석의 효율성이 증대된다 하겠다. 앞에서 언급한 바대로 개별 플랜의 해석에 있어서는 해석해야 할 코드의 범위의 확대로 탐색 공간이 커질 가능성이 있으나 프로그램 크기에 비례하는 산술 급수의 증가이며, 연관된 플랜 간의 해석에 따른 트리 탐색이 없어진 관계로 기하급수적인 탐색의 증가가 없어져서 전체적인 해석에 있어서 효율성이 증대되었다 하겠다.

3. 실행 결과를 이용한 프로그래밍 오류 분석 시스템

3.1 시스템의 구성

플랜정합 방법론에 프로그램 실행 결과를 결합 시킴으로써 효과적인 디버깅이 가능함을 보이기 위해 초보자용 C언어 프로그래밍 오류분석 시스템인 익스버그(ExBug: Execution guided deBugger)를 구현하였다. 시스템의 구성은 아래 그림 3과 같다. 2개의 주요한 구성부분은 프로그램을 실행시켜 주는 실행부와 플랜정합을 진행하는 정합부이다. 실행부는 초기 입력 자료로 주어진 테스트 데이터를 가지고 학생의 프로그램을 실행시키며, 그 과정에서 변수들의 값의 변화나 입출력된 데이터 값을 모두 저장해 둔다. 플랜정합부는 초기 입력 자료로 주어진 과제명세서(Programming Task Description)에 있는 요구사항에 근거하여 플랜 매칭을 시도하고, 그 차이점들을 프로그램 실행 결과를 이용하여 오류를 분석하게 된다.

익스버그에서 사용하는 정보에는 문제에 관한 정보와 프로그래밍 지식에 관한 정보 및 오류와 관련된 것들로 구성되어 있다. 문제에 관한 정보는 학생에게 제시된 특정 프로그램에 대한 정보로서 과제명세서 형식으로 표현되어 있다. 따라서 학생에게 주어지는 문제별로 과제명세서가 하나의 입력 자료로 주어진다. 이 명세서에는 학생이 프로그램에서 구현해야 할 목표들이 어떠한 조건을 만족해야 하는 지가 명시되어 있으며, 프로그램 실행에 사용되어질 테스트용 입력 데이터와 연관된 변수에 대한 최종값 등이 표현되어 있다.



[그림 3] 익스버그의 시스템 구성도
[Fig. 3] Overview of ExBug

프로그래밍 지식은 일반적인 프로그래밍에 관한 지식으로 고울이나 플랜의 형태로 저장되어 있다. 앞에서 언급한 합계를 구하는 것과 같은 전형적인 프로그래밍 패턴들은 C언어의 형태로 표현되어 플랜이란 이름으로 저장되어 있다. 형태는 다르지만 동일한 목표를 처리하는 플랜들을 모아서 고울이라는 이름으로 저장했다. 고울과 플랜의 구성과 표현 양식은 프레임의 형태로 표현하였으며, 프라우스트에서

표현된 내용을 기본적으로 사용하고 있으며 프로그램 실행과 연관되어 필요한 항목이 추가된 상태이다. 프로그래밍 오류에 관한 정보는 버그 목록으로 주어지며 이에는 일반적인 오류의 형태들이 저장되어 있다. 플랜 매칭 시 학생 프로그램에 있는 코드가 플랜에 있는 코드 패턴과 차이점이 발생할 경우, 예상 가능한 오류인지에 대한 판정은 이 차이가 버그 목록에 있는 지 여부로 판정하게 된다. 버그 목록의 확인을 받는 경우 학생이 오류를 범한 것으로 간주하며, 버그 목록에 의해 확인을 받지 못한 매칭의 차이는 플랜 적용이 잘못된 것으로 판정하게 된다.

3.2 오류 판정 전략

익스버그는 플랜정합이 끝난 후 오류를 확정하게 된다. 정합 단계에서도 개별적인 오류에 대해서는 원인과 그 여파가 확인되기도 하지만, 고울과 고울간에 영향을 미치는 오류에 대해서는 정합 과정에서는 전체적인 윤곽을 판단하기가 어렵다. 따라서 정합이 끝난 후 개별 플랜별 오류를 파악하여 해당 고울의 개별 오류를 정하고, 고울간에 연관된 오류의 내용과 그 여파에 대해 별도로 판정해서 학생에게 제시할 오류 메시지를 작성하게 된다. 이를 위해서 플랜정합 단계에서 코드 매칭 외에도 프로그램 실행 결과 중 과제명세서에서 제시된 요건이 모두 충족되었는지 여부도 검사해서 그 결과를 넘겨주게 된다. 익스버그에서 사용하는 오류 판정 전략 중 중요한 것 3가지를 정리하면 다음과 같다.

첫째는 매칭상의 차이점에 대해 실행 결과로 검증이 되는 경우 발하는 플랜 개별적인 판정 전략으로 아래 그림 4에 요약 정리되어 있다. 적용 대상은 'if', 'while', 'for',

● 플랜 내 매칭의 차이 검증 전략

적용대상:

'if', 'while' 등의 사용된 조건식

복잡하게 사용된 계산식의 매칭에 차이가 있을 때

eg) if (?New > 0) ?Counter++ ←→ if (data != 0) valid++

매칭 차이 : "> 0" vs. "!= 0"

확인사항:

'(data != 0)'이 실행된 모든 결과에 대해, 실행 당시 data값으로 '(data > 0)' 연산 진행한 결과와 모두 일치하는 지 확인

판정 기준:

틀린 경우 : "조건연산자 적용 잘못의 오류, 예제 데이터 제시"

같은 경우 : "'!=0'이 '>0'과 같도록 조치 가능성, 오류 판정 안함"

[그림 4] 익스버그의 오류판정 전략 1

[Fig. 4] Error Confirmation Strategy I of ExBug

'do-while' 문과 같은 조건문의 조건식에서 예상 가능한 매칭의 차이가 발생한 경우에 실행 결과를 이용하여 매칭의 차이를 검증하는 전략인 것이다. '(data != 0)'이 실행된 경우의 모든 'data' 값에 대해서 '(data > 0)'을 적용해 보아 두 조건식의 결과가 차이가 나면 조건 연산자 적용을 잘못하여 오류가 발생한 것으로 판정하고, 이 때의 'data' 값을 오류를 발생시킨 예제 데이터로 제시하게 된다. 그러나 두 조건식의 결과가 모두 같다면 사전에 음수인 데이터는 배제시키도록 조치를 취해서 제대로 구현된 프로그램임 가능성이 있으므로 오류 판정을 보류하게 된다.

통해서 변수에 가하는 플랜과 그 변수를 참조하는 플랜 간에는 인과관계가 주어짐을 알 수 있다. 이러한 경우 조건을 가하는 플랜의 매칭에 오류가 발생하는 경우 그 여파가 연관된 플랜에서 변수의 조건에 어긋난 잘못된 변수값이 사용될 가능성이 있다고 하겠다. 이처럼 인과관
계를 이용해서 플랜과 플랜간의 오류 전파를 설명해주는 전략이다.

그림 5에 있는 입력값을 검사하는 'Sentinel-ReadProcess-While Plan'은 특정 값이 주어질 때까지 입력 값을 받아들이는 입력

● 연관플랜 간 매칭의 차이 검증 전략

적용대상:

변수의 조건을 매개로 인과관계가 있는 플랜간에
조건을 가하는 플랜의 매칭에 차이점이 있는 경우
eg) 'Sentinel-ReadProcess-While Plan' 매칭에 차이점이 있고
변수 ?New에 종료값 99999를 제외토록 조건을 가하는 경우

확인사항:

?New를 파라메타로 받는 모든 플랜('Total Plan')의 계산에서
?New로 매핑된 변수의 값중 99999가 계산에 사용됐는지 검사

판정 기준:

예제 발견 : "Sentile-Plan 구현 오류에 따라 Total 계산에 오류"
예제 데이터를 제시하여 검토토록 메시지 제공
미발견 경우 : 오류 판정 안함, 전략 1에 따라 처리

[그림 5] 익스버그의 오류판정 전략 2
[Fig. 5] Error Confirmation Strategy II of ExBug

둘째는 인과관계가 주어진 프랜 간의 매칭 오류와 관련된 전략으로 그림 5에 요약되어 있다. 앞에서 언급한 대로 공통 변수의 사용을

값 검사 플랜으로 여기에 사용된 '?New'라는 변수에 대해 '(?New != 99999)'라는 조건을 가하는 플랜인 것이다. 만일 이 플랜의 매칭

결과에 조건연산자 적용 미스와 같은 오류가 발생했다면, 그 여파는 이와 연관된 'RUNNING-TOTAL Plan'에서 연산 과정에서 종료값이 사용될 가능성이 크다 하겠다.

이 경우 합계를 구하는 연산문의 매칭 과정에서 '?New'라는 변수에 가해진 조건에 어긋난 값이 사용되어 졌는 지를 따져서 오류를 판정하게 된다. 즉, 종료값 '99999'가 합계 구하는 연산에 사용된 예가 있으면 'Sentinel-ReadProcess-While Plan' 구현의 잘못에 따라 합계 계산에 종료값이 잘못 사용되는 오류가 야기되었다는 것을 예제 데이터를 제시하여 설명할 수 있게 되는 것이다. 만일 오류를 확인해 주는 예제가 나타나지 않는 경우에는 매칭과정의 오류 종류와 내용에 따라서 판정을 내리게 된다. 이때는 판정 전략 1에 의해 플랜 자체에 한정된 내용이 제시되며, 연관된 플랜과의 관련성은 전혀 언급하지 않게 된다.

세번째 오류 판정 전략은 개별 플랜의 목표 달성 여부와 관련된 부분이다. 시스템 개요를 설명하면서 언급한 대로, 익스버그는 테스트 데이터를 이용하여 실행 결과를 미리 예상할 수 있다. 따라서 최종 결과치가 제시된 변수를 포함하는 플랜에 대해서는 예상 값과 실제 프로그램 실행 후 결과를 비교하여 제대로 구현되었는지를 판단할 수 있다. 이를 이용하여 매칭의 오류가 없더라도 결과값이 예상치와 틀리게 나온 경우의 테스트 예제를 가지고 학생 스스로 검토해 보도록 조언을 제시할 수가 있다. 위의 그림 6은 이러한 전략의 적용 과정을 정리한 내용이다.

● 개별 플랜의 목표 달성 여부 확인 전략
적용대상 :
매칭 완료 후 과제 명세서에 예상값이 제시된 플랜에 대하여
확인사항 :
각 테스트 데이터에 대해 결과값이 예상값과 차이가 있는지 검사
판정 기준 :
- 틀린예제 : "예제 데이터로 확인해 보도록 요청",(조언)
- 같은예제 : "제대로 구현 되었다"판정.

[그림 6] 익스버그의 오류판정 전략 3
[Fig. 6] Error Confirmation Strategy III of ExBug

4. 적용 결과 분석

실제 프로그램 오류 분석 능력을 평가하기 위해 프라우스트에서 사용된 '강우량 문제'와 '알기 쉬운 C언어'[11] 교재에 있는 연습문제 13가지에 대해서 예제 프로그램 총 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. 실험 결과는 표 1에 정리하였다.

프로그램 397개 중에서90%에 해당하는 357개의 프로그램에 대해 익스버그는 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 357개의프로그램 중에서 오류가 없이 제대로 작성된 프로그램의 수효는 128개 이며, 오류를 포함하고 있는 것은 229개였으며 이를 수작업으로 분류한 결과 오류의 수효는 총 679개 였다. 익스버그는 이 오류 중 81%에 해당하는 552개에 대해서 제대로 지적해 냈으나, 19%인 127개에 대해서는 지적을 하지 않았다 (구현되지 않은 고울이 있다고 지적하면서 테스트 데이터로 검사할 것을 권고하는 경우는 버그를 지적한 것으로 판정하지 않았으며, 이를 포함하는 경우 지적율은 90%이상이 된다). 틀리지 않았는데 틀렸다고 하거나, 버그의 내용을 엉뚱하게 지적하는 허위경보(False alarm)의 수효는 36개가 있었다.

<표 1> 익스버그(ExBug) 테스트 결과
<Table 1> Test Result of ExBug

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%
_____	허위경보의 수
36	

이를 프라우스트의 실험결과와 비교하면 버그의 지적율에서는 81%대 94%로 상대적으로 떨어지나, 허위 경보에 대해서는 36개 (5%)대 66개 (11%)로 상대적으로 절반 이하로 줄어든 결과를 보여주고 있다. 익스버그의 지적율이 떨어진 것은 아직 개선의 여지가 있다는 점을 반영하는 것으로 판단할 수 있으나, 또 하나 상대적으로 보수적인 오류 판정 전략에 따른 결과이기도 하다. 즉 익스버그는 매칭의 차이점이 발생하더라도 실행 결과에 의해 뒷받침 되는 증거가 있는 경우에 한해 버그의 내용을 사용자에게 구체적으로 제공하는 보수적인 전략을 채택하고 있다는 것이다. 따라서 사용자를 혼란스럽게 만드는 허위 경보가 발생할 가능성을 줄였다. 게다가 구현되지 않은 고울이 있다고 조언을 해 주고 있는 관계로 틀린 프로그램을 맞았다고 오해하도록 내버려두지 않은 관계로 실제 사용자가 느끼는 오류의 지적율은 높다고 할 수 있다.

5. 결론

플랜정합은 프로그래밍 플랜이라는 지식을 바탕으로 학생의 프로그램을 해석하고 강력한 오류지적 기능을 갖고 있으나, 코드외 추가적인 정보가 없이 패턴 매칭에 의존하는 관계로 플랜 간 위치 관계나 특징적인 고울의 정확한 인식이 필요하다는 점 등의 약점을 안고 있으며, 결과적으로 분석이 정확치 못한 경우 허위경보를 사용자에게 발생시키는 문제점을 노출시키고 있다.

프로그램 실행 결과를 이용하면 패턴 매칭에서 야기된 플랜 간의 강한 연관관계를 줄이고 플랜 매칭을 효율적으로 할 수 있게 된다. 즉, 플랜 간의 강한 위치관계라는 연관관계의 표현 방식을 줄이는 대신, 프로그램 실행 결과를 이용하여 플랜 간 공통변수에 대한 조건이 만족되었는지를 조사함으로써 서로간의 인과관계를 표시할 수 있게 되었다. 또한 플랜 정합 과정에서 발생하는 차이점을 실행 과정의 중간값을 이용하여 프로그램 자체의 정보로 뒷받침된 오류 검증이 가능하게 됨으로써 유연하게 플랜간의 관계를 표시할 수 있으면서

분석 과정의 효율성과 효과를 높일 수 있게 되었다. 사용자에게 제공하는 오류에 대한 설명 메시지도 고울간의 인과 관계를 활용하여 오류의 원인과 그 여파를 구체적으로 지적해 줄 수 있으며, 예제와 반례를 제시하여 설명함으로써 초보자가 이해하기 쉬운 설명을 제공해 줄 수 있게 되었다.

향후 과제로는 플랜의 해석 과정에서 코드 매칭을 이용함에 따라 형태적 변화를 흡수하지 못하고 코드만 차이가 있는 비슷한 플랜을 양산하는 문제점을 안고 있다 (조건식이나 복잡한 계산식의 경우 전처리를 통해 일부 표준화를 이루었으나 의미는 같으면서 형태가 약간 차이가 있는 각기 다른 플랜으로 정의해야 한다). 따라서 그래프 파싱과 같이 좀더 형식화된 플랜 해석 방법론을 결합하는 것이 필요하다 하겠다.

참 고 문 헌

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

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

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

[4] S. Hahn, "Extraction of problem description from sample program for knowledge-based program tutoring", KAIST, 1997 Ph.D. Thesis

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

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

[7] S. Kim, "Algorithm Recognition for Programming Tutoring", KAIST, 1994 Ph.D. Thesis

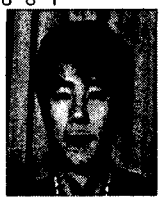
[8] W. Johnson, Intention-Based Diagnosis of Novice Programming Errors, Morgan Kaufmann Publishers, 1986

[9] P. Vanneste, "A reverse engineering approach to novice program analysis, KU Leuven Campus Kortrijk, Holland, 1994, Ph.D dissertation.

[10] 송종수, "플랜정합과 프로그램 실행에 의한 학생프로그램 오류분석에 관한 연구", KAIST 박사학위논문 1998, pp 29-33

[11] 이광형 외, 알기쉬운 C언어, 홍릉출판사, 1990.

송종수



1982년 서울대학교 계산통계학과 (학사)

1987년 서울대학교 계산통계학과 (석사)

1998년 한국과학기술원 전산학과 (박사)

1983년-1998년 (주)삼보컴퓨터연구소 부장

1998년-2001년 (주)핸디소프트 기술연구소 수석연구원

2001년-현재 (주)엔포엔 대표이사

관심분야 : 인공지능, e-Learning,지능형교사시스템(ITS), 임베디드 시스템

송두헌

1981년 서울대학교 계산통계학과 (학사)

1983년 한국과학기술원 전산학과 (석사)

1994년 UC Irvine 전산학과(박사 수료)

1983~1986 KIST 연구원