논•문15-1-5

테스트 시스템에서 혼합 디버깅 시스템

(A Mixed Debugging System in Test Systems)

고 훈 준, 유 원 희

인하대학교 전자계산공학과

g2001418@inhavision.inha.ac.kr

whyoo@inha.ac.kr

요 약

테스트 시스템은 반도체 제품을 웨이퍼(wafer) 또는 완성된 제품 상태 하에서 전기적 특성과 성능을 검사하고 그 결과를 산출해내는 검사장치이다. 테스트 시스템은 하드웨어와 소프트웨어로 구성되어 있으며, 특히 시스템을 제어하고 사용자 인터페이스 및 각종 자료를 처리하는 소프트웨어는 그 중요성이 한층 더 부각되고 있다. 특히, 테스트 시스템 내에서 프로그램을 컴파일하고 디버깅하는 기술 개발은 매우 어려운 실정이다. 디버깅 기술은 엔지니어가 테스트 프로그램의 오류를 빨리 찾아 수정하게 함으로써 정확한 제품 생산과 생산량을 높이게 한다. 따라서 테스트 시스템의 디버깅 기술은 매우 중요하다.

본 논문에서는 기존 알고리즈믹 디버깅 이론의 문제점을 지적하고 이 문제점을 해결할 수 있는 혼합 디버깅 방법을 제안한다. 그리고 테스트 관리 프로그램에 혼합 디버깅 방법을 적용함으로서 엔지니어가 테스트 프로그램을 편리하고 효율적으로 디버깅하여 산업현장에서 일의 능률을 높이고자 한다.

1. 서론

최근 반도체 산업의 발전으로 반도체 종류의 다변화와 생산량이 크게 증가하고 있으며 반도체가 고집적, 고성능화 됨에 따라 세계 반도체업계의 테스트 기술에 대한 관심과 기술은 점차 발전하고 있다.

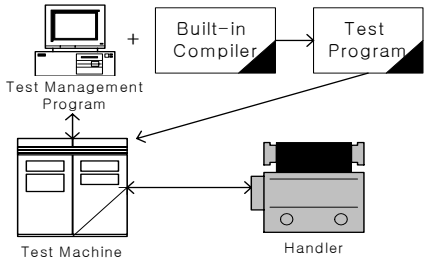
테스트 시스템(test system)은 반도체 제품을 웨이퍼(wafer) 또는 완성된 제품 상태 하에서 전기적 특성과 성능을 검사하고 그 결과를 산출해내는 검사장치이다. 컴퓨터의 발전으로 테스트를 위한 자동화 시스템의 중요성은 더욱 높아지고 있다. 최근까지도 테스트 시스템은 그 고유의 기술적 장벽으로 몇몇 유명한 외국 업체가 세계시장 및 국내시장을 독점해 왔으며 최근 국내에서 독자적인 기술력으로 테스트 시스템을 개발하여 국산화를 이루는 업체가 자생하고 있다[7,8].

테스트 시스템은 크게 하드웨어와 소프트웨어로 이루어져 있으며, 특히 시스템을 제어하고 사용자 인터페이스 및 각종 자료를 처리하는 소프트웨어는 그 중요성이 한층 더 부각되고 있다. 시스템 관리 프로그램(system management program)은 엔지니어가 테스트 장비를 올바르게 다루는데 중요한 프로그램이다. 테스트 시스템은 다양한 종류와 기능의 반도체 제품을 검사한다. 따라서 여러 가지 특수기능의 하드웨어 모듈(module)과 각 제품의 특성에 맞게 프로그램 된

테스트 프로그램이 필요하다. 또한 그 시스템에 알맞은 프로그래밍 언어와 테스트 프로그램을 실행하기 위한 컴파일러와 디버깅 시스템이 필요하다[7].

전 세계적으로 가장 큰 시장점유율을 갖고있는 TERADYNE INC.는 테스트 프로그램 언어로 파스칼 언어를 기반으로 한 PASCAL/STEPS와 C++ 언어를 기반으로 개발한 언어를 사용하고 있다. 이 언어는 전 세계적으로 TERADYNE 테스트 시스템을 사용하고 있는 수많은 테스트 엔지니어들에 의해 사용되고 있으며 그 편리성과 안전성을 인정받고 있다[8].

국내 벤처기업인 STATEC INC.는 지금까지 고성능의 테스트 시스템을 개발하여 왔으나 테스트 프로그램은 상용 C 컴파일러를 이용하여 테스트 시스템에 적용하여 왔다. 그러나 이 시스템은 첫째는 테스트 엔지니어에게 불편한 시스템 제어 명령어를 제공하고, 둘째는 테스트 실행 시 매번 프로세스를 생성하는 문제점이 있었다. 이와 같은 문제점은 새로운 프로그래밍 언어와 이 언어를 컴파일 하여 실행할 수 있는 컴파일러를 [그림 1]과 같이 설계함으로써 해결하였다[10,11].



[그림 1] 개발한 테스트 시스템의 구성도

프로그래밍 언어는 C언어와 시스템 제어 명령어로 구성된 혼합 언어를 사용하고 컴파일러는 번역기와 가상 기계를 테스트 관리 프로그램에 내장함으로써 프로그램의 컴파일과 실행이 테스트 관리 프로그램 내에서 모두 가능하게 하였다. 그러나 프로그램의 오류에 대한 디버깅 시스템이 없다. 디버깅 기술은 엔지니어가 편리하고 효율적으로 테스트 프로그램의 오류를 빨리 찾아 수정하게 함으로써 정확한 제품 생산과 생산량을 높이게 하는 매우 중요한 기술이다.

최근 함수형 언어를 이와 같은 분야에 적용하는 사례가 많이 발견되고 있다. 프로그래밍 언어 모델로서 함수 언어는 평가 순서에 관계없이 항상 일정한 값을 생성할 수 있는 참조 명료성이 보장된다는 것 때문에 많은 연구가 되고 있다. 이러한 함수 언어는 명령형 언어와는 달리 할당문이 없기 때문에 함수 프로그램은 잠재적인 병렬성을 갖는다. 또한 함수 언어는 부작용이 없으므로 명확한 의미를 갖게 되어 병렬적이고 비결정적인 스케줄링인 경우에도 함수 프로그램의 결과는 항상 일정하고 단일 처리기에서 디버깅되어 다중 처리기에서 수행될 수 있으므로 병렬 프로그래밍 시 매우 어려운 병렬 프로그램의 디버깅도 간단해진다. 또한 함수언어를 다중스레드 모델을 적용한 가상 기계와 함수언어를 자바 바이트 코드로 변환하여 자바 가상기계에서 수행하는 연구가 진행되고 있다. 이와 같은 가상기계를 사용함으로서 쉽게 테스트 시스템에 적용할 수 있을 것으로 예상된다[7,10].

일반적으로 컴파일러에 내장된 파서와 타입 시스템(type system)은 프로그램을 컴파일 할 때 프로그램이 실행되기 전에 발생하는 구문 오류와 그 외 많은 오류를 정적 분석을 통해서 발견한다. 그래서 사용자는 컴파일러에서 발견된 오류를 쉽게 수정할 수 있다. 하지만 컴파일러가 모든 에러를 발견할 수 있는 것은 아니다. 프로그램 실행 도중에 발생하는 실행시간 오류(runtime error)나 논리적인 오류(logical error)는 디버깅 시스템을 이용하여 해결해야 한다.

현재 개발된 테스트 관리 프로그램의 경우, 구문 오류는 새롭게 설계한 컴파일러가 발견해 주지만, 실행시간 오류 등은 기존 상용 컴파일러에 있는 디버깅 시스템에 의존해야 한다. 이는 다음과 같은 문제점이 있다. 테스트 관리 프로그램과 디버깅 시스템의 상호 연동에 굉장히 비효율적이다. 또한 테스트 시스템의 하드웨어는 테스트 프로그램 실행 중에 비정상적인 상태가 될 수 있다. 더욱이 기존 상용 컴파일러에 내장된 디버깅 시스템은 하드웨어의 상태를 엔지니어가 디버깅 중에 쉽게 조절 할 수 가 없다. 이것은 산업현장에

서 엔지니어의 일에 대한 능률을 감소시키는 결과를 초래할 뿐만 아니라 또한 상용 컴파일러의 라이센스(license) 문제도 발생한다. 따라서 엔지니어에게 편리하고 효율적으로 디버깅을 할 수 있는 디버깅 시스템을 개발하는 것이 필요하다.

프로그램의 기본적인 디버깅 방법은 프로그램 내에 프린트 명령과 같은 화면 출력 명령어를 사용하여 원하는 값을 검사하는 기본적인 방법이 있지만 매우 비효율적이다. 좀 더 향상된 디버깅을 위한 연구가 많은 곳에서 진행 되어왔으며 대표적인 방법으로 하향식 방법(top-down method)을 이용하는 알고리즈믹 디버깅 이론(algorithmic debugging theory)이 있다.

알고리즈믹 디버깅 이론은 프로그램의 시작부터 종료까지 모든 프로시저의 호출관계를 실행 트리로 구성하여 실행 트리의 상위 레벨부터 하위 레벨로 탐색을 하고 오류를 발견하는 반자동화 디버깅 이론이다. 프로그램이 실행되면 프로시저의 동작에 대해서 사용자가 디버깅 시스템에게 “예”와 "아니오"로 답변하여 오류를 찾는다. 프로그램에 오류가 검색된다면 이 오류는 자신 함수에 존재하거나 이 함수를 호출한 상위 레벨의 함수에 오류가 있다고 판단한다. 그러나 알고리즈믹 디버깅 이론은 프로그램의 크기가 커지면 실행 트리의 크기가 커진다는 문제점이 있다. 그리고 오류를 프로시저 단위까지만 검색을 할 수 있고 프로그램의 표현식 단위까지는 검색하기 어렵다. 또한 알고리즈믹 디버깅 이론은 한번에 하나의 오류만을 발견하는 문제점이 있다.

본 논문에서는 알고리즈믹 디버깅 이론의 문제점을 해결하기 위한 혼합 디버깅 방법(mixed debugging method)을 제안한다. 혼합 디버깅 방법은 일반적인 디버깅 방법과 알고리즈믹 기반 디버깅 방법을 혼합하여 사용하는 디버깅 방법이다. 이 방법은 우선 알고리즈믹 디버깅 이론은 적용하여 오류가 포함된 프로시저까지 검색을 하고 바로 프로시저 내의 표현식을 순차적으로 실행하여 오류를 발생시키는 표현식을 검색하는 방법이다. 또한 오류가 하나 이상일 경우 연속해서 디버깅이 가능하다는 장점이 있다.

2 장에서는 관련 연구로 일반적인 디버깅 방법과 하향식 디버깅 이론인 알고리즈믹 디버깅 이론을 설명하고 디버깅 이론에서 발생되는 일반적인 문제점들을 지적한다. 3 장에서는 본 시스템에서 효율적으로 사용할 수 있는 알고리즈믹 디버깅 이론을 기반으로 한 혼합 디버깅 시스템을 설명한다. 4 장은 테스트 시스템의 테스트 관리 프로그램에서 사용할 수 있는 디버깅 시스템의 구현에 관하여 설명하고 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

이 장에서는 명령형 언어에서 일반적으로 사용하는 디버깅 방법들과 알고리즈믹 디버깅 이론을 설명하고 이 방법들의 문제점을 설명한다.

2.1. 일반적인 디버깅 방법

디버깅을 하기 위한 가장 일반적인 방법으로 첫 번째는 오류가 발생할 것 같은 프로그램의 중간에 프린트 명령과 같은 화면 출력 명령어 등을 삽입하여 프로그램을 디버깅하는 방법이다. 이 방법은 프로그램의 오류가 발생하는 곳을 정확히 예상할 수 있어야 사용할 수 있기 때문에 사용자가 프로그램을 이해하고 사용하기에는 어렵다. 두 번째는 Step Over, Step Into 등을 사용하여 프로그램을 디버깅하는 순차적 실행 방법이다. 이는 프로그램의 크기가 클수록 디버깅하는 시간이 기하급수적으로 증가하므로 디버깅 시스템을 사용하는 사용자에게는 매우 불편하다. 세 번째는 변수 창을 이용하여 원하는 변수의 값을 검사하는 방법이다. 이 방법은 두 번째 방법과 같은 다른 방법을 같이 사용해야 효과적으로 이용할 수 있는 방법이다[2,4].

이와 같은 방법들은 프로그램의 크기가 커질수록 디버거 사용자에게 너무 많은 예측과 실행 작업을 요구하기 때문에 효율적인 디버깅 작업이 어렵다.

2.2. 알고리즈믹 디버깅 이론

알고리즈믹 디버깅 이론은 Shapiro에 의해 처음으로 제안된 소프트웨어 디버깅을 위한 기술로서 프롤로그(prolog) 언어에서 처음으로 제안되었

다[4]. 그 후 알고리즈믹 디버깅 이론은 다른 언어를 디버깅하는 데도 이용되었다. 명령형 언어에서의 알고리즈믹 디버깅은 Nahid Shahmehri에 의해서 상세하게 표현된 GADT(generalized algorithmic debugging technique)로 알려져 있다[2,3,4].

디버깅 시스템은 프로그램을 실행하고 프로시저 이름 또는 입력/출력 매개변수 값과 같은 유용한 추적 정보(trace information)를 저장하는 동안 프로시저 수준의 추적 실행 트리(trace execution tree)를 생성한다. 실행 트리는 프로시저를 노드로 만들어 프로그램의 실행 순서에 따라 루트로부터 하향식으로 구성된다. 그리고 호출된 프로시저는 자식 노드가 되고 왼쪽에서부터 오른쪽으로 구성된다. 실행 트리가 생성되면 [알고리즘 1]을 이용하여 추적을 시작한다.

[알고리즘 1] Algorithmic Debugging Algorithm

```
input: unit_activation, a node of program's execution tree
output: the erroneous execution tree node or NIL
local: curr_node, error_found are nodes of the same type as the execution tree nodes
other routine : Unprocessed_child_left, True if any child node is remained, False otherwise
Fetch_next_child, Return the execution tree node of the next child.
Correct_Behavior, Ask user. Return True if answer is "yes", False otherwise
procedure Algorithmic_Debugging(curr_node)
begin
   if Correct_Behavior(unit_activation) then
      return NIL;
   else
      error_found := NIL;
      while error_found = NIL and
       Unprocessed_child_left(unit_activation) do
         curr_node := Fetch_next_child(unit_activation);
       error_found:=Algorithmic_Debugging(curr_node);
      od
      if error_found = NIL then
         return parent_node;
      else
         return curr_node;
      fi
   fi
end
```

알고리즈믹 디버깅 시스템은 실행 트리의 루트부터 하향식 방법으로 추적하고 각 프로시저에서 기대할 수 있는 행동에 대해 사용자에게 질문함으로써 동작한다. 사용자는 각 프로시저의 행동에 대해 “예” 또는 “아니오”로 대답할 수 있다. “예”라고 대답할 경우에는 현재 프로시저와 그의 하부 자식 프로시저에 오류가 없다는 뜻이고 더 이상 자식 프로시저로 추적을 하지 않고 형제 프로시저 노드로 이동하여 추적한다. “아니오”라고 대답할 경우에는 현재 노드와 그의 자식 노드 중에 오류가 있다고 보고 자식 노드로 이동하여 질문을 한다. 추적은 더 이상 자식 노드가 없거나 사용자가 “예”라는 대답을 할 때까지 진행이 된다. 검색이 끝나면 오류는 다음과 같은 경우 중 한 가지일 때 [그림 2]의 프로시저 p안에 발생한다.

- 프로시저 p는 프로시저 호출이 없다.
- 프로시저 p의 몸체로부터 수행되는 모든 프로시저 호출은 사용자의 기대를 만족한다.

디버거의 출력은 입력 매개변수와 기대되는 결과에 의해 사용자의 대답으로 생성되고 오류는 어떤 프로시저 몸체 안에서 발견된다.

알고리즈믹 디버깅 이론의 예는 [그림 2]와 같다. 입력 매개변수 a와 c, 출력 매개변수 b와 d를 가지는 프로시저 p를 생각해보자.

```
procedure p(a, c: integer; var b, d: integer)
   procedure q(a: integer; var b: integer)
         • • •
   end;
   procedure r(c: integer; var d: integer)
         • • •
   end;
begin
         q(a, b);    r(c, d);
end;
```

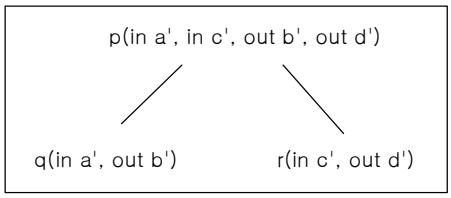
[그림 2] 입력/출력 값이 각각 두 개인 프로시저 p

변수 b의 값은 프로시저 q를 호출함으로서 계산되고 변수 d는 프로시저 r을 호출함으로서 계산된다. 프로그램을 실행하기 위해 다음을 가정하자.

첫째, 입력 값 a′와 c′를 갖는 프로시저 p는 출력 값 b′ 와 d′를 리턴한다.

둘째, 오류는 잘못된 출력 값 d′가 발생하는 프로시저 r에 있다.

디버깅 시스템이 프로그램을 실행하면 [그림 3]과 같이 프로시저 단위로 실행 트리가 생성된다.



[그림 3] 실행 트리(execution tree)

실행 트리를 가지고 동작하는 알고리즈믹 디버깅 시스템과 사용자와의 대화는 다음과 같다.

p(in: a=a′, in: c=c′, out: b=b′, out: d=d′)?
> no
q(in a=a′, out b=b′)?
> yes
r(in: c=c′, out d=d′)?
> no
An error has been localized inside the body of procedure r

굵은 글씨체는 디버깅 시스템의 출력을 나타내고 '>'에 의해 표현되는 것은 사용자의 대답을 나타낸다.

알고리즈믹 디버깅 이론을 명령형 언어에 적용하기 위해서는 먼저 매개 변수로 전역 변수를 사용하는 경우와 goto 명령어의 사용 등 기본적인 부작용(side effects)을 제거해야 한다.

Nahid Shahmehri는 이와 같은 부작용을 제거하기 위해 소스 프로그램을 프로그램 변환(program transformation) 단계를 이용하여 부작용이 없는 변환된 프로그램 트리를 생성하였다. 그리고 변환된 프로그램 트리로부터 실행 트리를 생성하여 알고리즈믹 디버깅을 적용하였다.

이와 같은 디버깅 시스템은 시스템의 질문에 사용자가 정확히 담변을 할 경우는 프로그램의 오류를 쉽게 찾을 수 있지만, 부정확한 담변을 할 경우에는 오류를 발견하기가 어렵다. 따라서 정확히 오류를 발견하기 위해서는 반드시 사용자의 정확한 답변이 중요하다. 또한 사용자가 전체 프로그램을 정확히 이해하지 못한 경우에도 프로시저 단위로 입력과 출력의 값을 예상하여 담변을 함으로써 디버깅을 할 수 있기 때문에 기존 일반적인 디버깅 방법에서 디버깅 시스템이 사용자에게 요구하는 부담이 줄어든다. 그러나 여전히 프로그램의 크기가 클수록 실행 트리를 생성하는 부담이 크고 디버거 사용자가 시스템의 질문에 너무 많은 대답을 해야하는 문제점이 있다. 또한 프로시저 단위까지만 디버깅이 가능하기 때문에 오류의 원인이 되는 표현식을 찾는데는 어려움이 따른다. 알고리즈믹 디버깅 이론은 또한 한번에 하나의 오류밖에 발견할 수 없다. 따라서 하나의 오류를 발견하고 다음 오류를 발견하기 위해서는 처음부터 다시 사용자가 대답을 해야하는 부담이 있다. 또한 처음 실행 트리를 생성하여 오류를 찾았을 때 오류가 발견된 노드의 부모 노드를 포함한 조상노드와 오류가 발견된 노드에 영향을 받는 다른 노드들이 첫 번째 디버깅에서는 디버거 사용자가 "예"라고 대답을 하였을지라도 오류가 없다고 확신할 수 없다. 따라서 프로그램에 대한 정확한 디버깅을 하는 방법은 여러 번 반복하여 오류가 발견되지 않을 때까지 실행해야 한다. 그러나 매번 같은 수의 노드를 가지는 실행 트리를 생성하여 실행하면 디버거 사용자는 전 단계에서 "예"라고 대답한 노드들을 매번 다시 대답하는 문제점이 있다.

3. 혼합 디버깅 방법

혼합 디버깅 방법은 일반적인 순차적 디버깅 방법과 알고리즈믹 디버깅 이론을 혼합하여 윈도우 환경에서 사용하는 방법이다. 이 방법은 알고리즈믹 디버깅 이론에서는 함수내의 표현식까지 오류를 발견할 수 없다는 것과 프로그램 내에 한 개 이상의 오류가 있더라고 한번에 하나 이상의 오류 발견할 수 없다는 문제점을 해결한다. 또한 실행 트리를 재구성하여 디버깅을 할 때 프로그램을 분할함으로서 사용자에게 답을 요구하는 횟수를 줄이는 방법을 제안한다.

3.1. 혼합 디버깅의 실행 구조

혼합 디버깅 방법은 오류를 발견하기 위해 먼저 알고리즈믹 디버깅 이론을 이용한다. 이 방법

을 사용하여 먼저 오류가 있는 함수를 찾아내고, 순차적 디버깅 방법을 이용하여 오류가 발견된 함수내의 표현식들을 순차적으로 실행함으로서 문제가 되는 표현식까지 발견한다. 그리고 다시 실행 트리를 재 생성하지 않고 오류가 발견된 함수 노드부터 디버깅을 계속 진행한다. 디버깅의 시작은 오류가 발견된 함수의 자식 노드들은 제외하고 아직 추적하지 않은 형제 노드들 또는 형제 노드가 없을 경우에는 부모 노드의 형제 노드들이다. 이와 같이 알고리즈믹 디버깅 이론과 순차적 디버깅 방법을 반복 수행함으로써 실행 트리를 한 번만 만들고 프로시저를 모두 검사할 수 있다. 이는 서로 의존성 없이 독립적으로 존재하는 오류들을 찾을 수 있다. 혼합 디버깅의 주요 실행 구조는 [알고리즘 2]와 같다.

[알고리즘 2] Mixed Debugging Algorithm

```
input: root_node, unit_activation, a node of program's execution tree
Algorithmic_Debugging: [Algorithm 1]
Step_Debugging: execute the expression by step-by-step
Get_number: Return the real line number of the source program
Fetch_next_node: Return the execution tree node of the next node except the child node
procedure Mixed_Debugging(root_node)
begin
    node := root_node
    while (node != NIL) do
        node := Algoritmic_Debugging(node);
        line := Get_number(node);
        Step_Debugging(line);
        node := Fetch_next_node(node);
    od
end
```

알고리즈믹 디버깅은 [알고리즘 1]의 기존 알고리즈믹 디버깅 이론을 사용한다. 순차적 디버깅 방법은 알고리즈믹 디버깅으로부터 오류가 발생한 노드의 실제 소스 프로그램의 라인 번호를 리턴 받아 그 라인 번호까지 실행을 한다. 그리고 디버거 사용자의 요구에 따라 하나의 표현식 단위로 프로그램을 실행함으로서 디버깅한다.

[알고리즘 3]의 VM[10]은 개발된 테스트 시스템의 가상 기계를 이용한다.

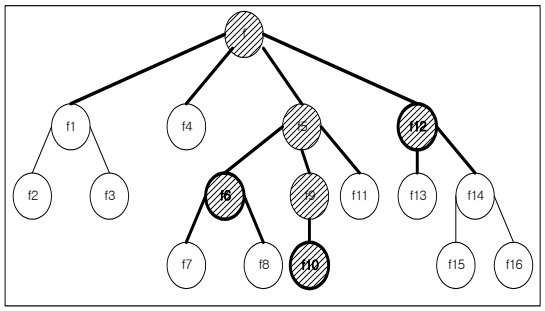
[그림 4]의 실행 트리의 예를 보자. 원은 함수를 나타내는 노드이고 'f'는 시작 함수를 나타낸다. 함수 f는 f1, f4, f5, f12의 호출 함수를 가지고 있고 f1은 f2와 f3의 호출 함수를 가진다. 굵은 선은 알고리즈믹 디버깅 방법을 이용하여 추적되는 간선이고 색칠 된 원은 디버거 사용자가 "아니오"라고 대답한 노드이다. 굵게 표시된 원은 오류가 있는 함수라고 가정한다.

[알고리즘 3] Step-by-step Debugging

```
input: line, the line number of the source program.
root_node, a node of the transformation program tree
VM: the Virtual Machine of the test system
User_answer: True if user answer is GO, False otherwise
Get_next_node: Return a node of program transformation tree
procedure Step_Debugging(line)
begin
        node = root_node
        while (node->line != line) do
                VM(node);
                node = Get_next_node();
        od
        while (user_answer()) do
                VM(node);
                node = Get_next_node();
        od
end
```

[알고리즘 1]을 이용하여 노드 f로부터 노드 f6까지의 추적 경로는 f → f1 → f4 → f5 → f6이다. 노드 f2와 f3은 노드 f1에서 "예"라고 대답했기 때문에 더 이상 자식 노드를 추적하지 않고 다음 노드로 이동을 한다. 노드 f6에서 디버거 사용자의 대답은 "아니오"이므로 디버거 시스템은 자식 노드 f7과 f8을 순차적으로 사용자에게 질문을 한다. 노드 f7과 f8은 오류가 없는 노드이기 때문에 노드 f6이 오류가 있는 노드라고 인식된다. 그리고 현재 노드의 위치를 기억하고 프로그램 트리(program tree)를 생성하여 그 노드까지 실행하고 노드 f6의 함수 내의 표현식을 순차적으로 실행하여 오류를 발견한다. 오류를 발견하고 나면 다시 실행 트리로 전환해서 다음 노드 f9 → f10을 추적한다. 노드 f10는 자식 노드가 없으므로 디버깅 시스템은 노드 f10에 오류가 있다고 판단한다. 이와 같은 방법을 사용하여 디버깅 시스템은 f11 → f12 → f13 → f14를 추적하고 노

드 f12에 오류가 있다는 것을 알게된다. 따라서 실행 트리를 한번 만 만들면 한 개이상의 오류를 발견할 수 있고 오류가 발생되는 표현식까지 추적이 가능하다.

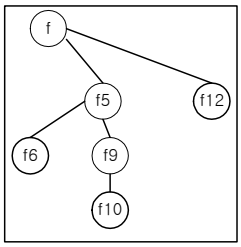


[그림 4] 실행 트리의 예

3.2. 분할

프로그램 분할은 프로그램을 분석하거나 디버깅 할 때 조사해야하는 프로그램 코드의 양을 줄이기 위한 방법으로 M. Weiser에 의해 제안되었다[5].

분할은 실행 트리의 연속적인 생성에서 불필요한 노드들은 제거하는 방법이다. 알고리즈믹 디버깅 이론에서 매번 같은 수의 노드를 가지는 실행 트리를 생성하면 디버거 사용자는 전 단계에서 "예"라고 대답한 노드들을 매번 다시 대답해야하는 문제점이 있다. [그림 4]의 경우, node f1, f2, f3, f4, f7, f8, f11, f13, f14, f15, f16은 실행 트리를 두 번째 생성할 때 불필요한 노드들이다. f1에서 디버거 사용자는 "예"라고 대답을 하였다. 따라서 f2와 f3는 오류가 없는 노드라고 확신 할 수 있다. 노드 f7과 f9의 경우도 단말 노드로서 사용자 의 대답은 "예"이기 때문에 재실행 트리를 생성할 때는 불필요한 노드들이다.



[그림 5] 재실행 트리

만약 재실행 트리가 [그림 4]와 같이 똑같은 트리 구조고 오류가 없는 구조라도 디버거 시스템은 노드 f1과 f4에 대해 디버거 사용자에게 질문을 하게된다. 그러나 [그림 5]의 경우는 전 단계에서 확실한 노드는 제거하여 실행 트리의 크기를 줄였고 노드 f1과 f4와 같은 노드는 존재하지 않기 때문에 디버거 시스템이 디버거 사용자에게 질문을 하는 횟수가 줄어들게 된다. 따라서 디버거 사용자는 시스템에게 대답하는 횟수가 줄어드는 장점이 있다.

재실행 트리를 생성하기 위해서는 이전 실행 트리에서 디버거 사용자의 대답에 대한 정보가 필요하다. 사용자의 대답을 저장하고 다음 실행 트리를 생성할 때 그 정보를 가지고 "예"라고 대답한 노드와 그 자손 노드들은 모두 생성하지 않는다.

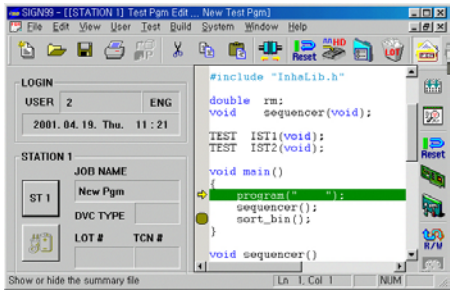
4. 구 현

이 장에서는 테스트 시스템의 프로그램 변환과 가상 기계를 기반으로 한 혼합 디버깅 방법과 분할을 적용한 디버깅 시스템의 구현을 알아본다. 먼저 테스트 시스템의 구조를 알아보고, 테스트 시스템에서 혼합 디버깅의 수행 구조에 대해 알아본다.

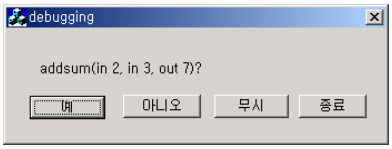
4.1. 혼합 디버깅 시스템의 에디터

이 논문에서 구현한 혼합 디버깅 시스템의 디버거 사용자 화면은 [그림 6]과 [그림 7]이다.

[그림 6]은 테스트 관리 프로그램으로 프로그램 에디터는 순차적 실행이 가능하도록 break point 표시기능과 프로그램 실행 라인 표시 기능이 있다.



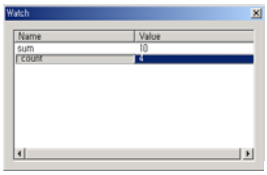
[그림 6] 개선된 테스트 관리 프로그램



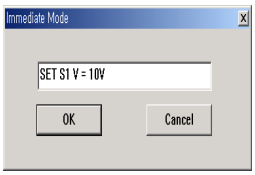
[그림 7] 알고리즘 디버깅의 질문

[그림 7]은 알고리즘 디버깅 이론을 구현한 프로그램으로 실행 트리의 노드를 실행할 때마다 디버거 시스템이 디버거 사용자에서 질문을 하는 메시지 창이다. 사용자는 “예”와 “아니오”를 선택하고 디버깅 시스템은 이를 기반으로 오류가 있는지 검색한다. “아니오”라고 선택한 노드가 오류가 있는 노드라면 디버깅 시스템은 노드의 [그림 6]의 사용자 화면으로 이동하여 오류가 있는 함수부터 순차적 실행을 한다.

그 외의 디버깅 방법으로는 순차적 실행을 효과적으로 사용하기 위해 변수 창을 지원한다. [그림 8]와 [그림 9]은 [그림 10]의 혼합 디버깅 시스템의 etc 에 해당하는 디버깅 시스템이다.



[그림 8] 변수 창



[그림 9] 직접 실행 창

[그림 8]는 순차적 실행 중에 원하는 변수의 값을 감시하여 오류를 생성하는 표현식을 쉽게 찾을 수 있도록 한다.

테스트 시스템의 하드웨어는 디버깅 중에 프로그램의 실행에 따라 상태가 변경된다. 그리고 하드웨어의 상태에 따라 올바른 프로그램도 잘못된 디버깅으로 오류를 발생시킬 수 있다. 따라서 디버거 사용자는 하드웨어의 상태를 감시하고 [그림 9]의 창을 이용하여 하드웨어 제어 명령어를 실행함으로써 사용자가 원하는 상태로 하드웨어를 변경할 수 있다. 실행은 디버깅 시스템을 사용하지 않고 테스트 관리 프로그램의 컴파일러를 이용하여 실행한다.

4.2. 프로그램 변환

프로그램 변환(program transformation)은 소스 프로그램을 입력받아 알고리즘 디버깅이 가능한 프로그램 구조로 변환 프로그램 트리(transformation program tree)를 생성한다. 이 방법은 N. Shahmehri이 제시한 방법을 기반으로 하였다. 테스트 시스템의 소스 프로그램은 현재 C언어 서브셋과 하드웨어 명령어로 구분된다. C언어에서 포인터, 구조체 등은 지원하지 않는다.

변환 프로그램 트리(transformation program tree)를 생성하기 위해서는 기존 프로그램의 구조를 몇 가지 변환해야 한다. 첫째는 하드웨어 제어 명령어는 C 언어 구문이 아니므로 C 언어의 함수 형태로 변환한다. 그러나 이 함수는 라이브러로 구성이 되어 있기 때문에 실행 트리 구성에 포함하지 않는다. 둘째는 전역변수가 함수 내에서 사용될 경우 함수의 입력 변수로 변환한다. 셋째는 리턴 변수는 사용할 것을 권장하고 함수의 출력 변수로 변환한다. 함수의 리턴 변수가 없을 경우에는 출력 값은 ‘x’로 표기한다.

4.3. 실행 트리 생성

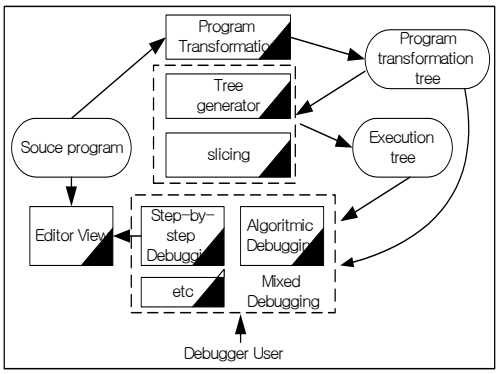
본 논문에서 제시하는 실행 트리 생성기는 트리 생성기(tree generator)과 분할(slicing)로 구분되어 있다. 트리 생성기는 프로그램 변환 트리를 순차적으로 실행하고 실행 결과를 이용하여 함수 단위로 트리를 생성한다. 트리는 프로그램의 시작 함수가 실행 트리의 루트 노드가 된다. 그리고 실행 순서에 따라 왼쪽부터 오른쪽으로 자식 노드와 형제 노드를 구성한다. 프로그램의 실행 트리는 [그림 3]과 같이 프로그램의 실제 실행에 대한 정보를 가지고 있는 트리 구조이다. 이 트리의 노

드는 함수의 이름과 입력 값, 출력값, 함수의 실제 라인 번호 등의 추적 정보를 가지고 있다.

4.4. 혼합 디버깅의 실행 예

혼합 디버깅 시스템의 예로 [그림 11]를 생각해 보자. 디버깅 시스템의 전반적인 구조는 [그림 10]과 같다.

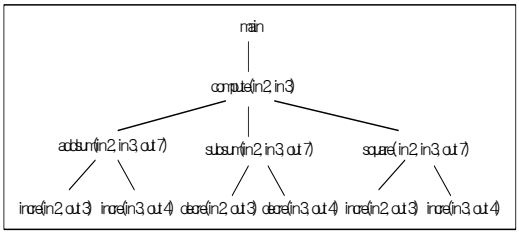
[그림 10]의 혼합 디버깅 시스템의 구조를 이용하여 실제로 실행하는 과정을 설명한다. 디버깅을 시작하면 [그림 11]의 프로그램을 프로그램 변환에 의해 AST(abstract syntax tree) 구조로 생성을 하고 생성된 트리를 루트부터 하향식으로 순차적 실행을 하면 [그림 12]와 같은 실행 트리가 완성된다.



[그림 10] 혼합 디버깅 시스템의 구조

```
int decre(int num) {
   num = num +1; return num;
}
int incre(int num) {
   num = num + 1; return num;
}
int square(int a, int b) {
   int c;
   a=incre(a); b=incre(b); c= a+b;
   return c;
}
int addsum(int a, int b) {
   int c;
   a = incre(a); b= incre(b); c = a+b;
   return c;
}
int subsum(int a, int b) {
   int c;
   a = decre(a); b= decre(b);
   c = a+b; return c;
}
void compute(int a, int b) {
   addsum(a, b); subsum(a, b); square(a, b);
}
void main() {
   int a, b;
   a =2;  b= 3; compute(a, b);
}
```

[그림 11] 프로그램 예



[그림 12] 실행 트리

알고리즈믹 디버깅으로 실행 트리를 디버깅하는 과정은 다음과 같다.

compute(in 2, in 3, out x)? "무시"
addsum(in 2, in 3, out 7)? "예"
subsum(in 2, in 3, out 7)? "아니오"
decre(in 2, out 3)? "아니오"

compute 함수는 리턴 값이 없다. 따라서 출력

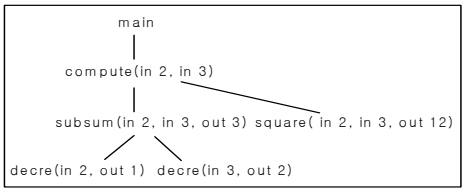
값을 예상할 수 없기 때문에 이 과정에서 사용자가 “무시”라고 답한다. 단말 노드 decre(in 2, out 3) 함수의 대답은 “아니오“이다. 따라서 디버깅 시스템은 그 함수의 오류가 있음을 알고 알고리즈믹 디버깅을 중단한다. 그리고 순차 디버깅 시스템에게 프로그램의 실제 라인번호를 넘긴다. 순차 디버깅 시스템은 라인 번호를 받아 변환 프로그램 트리를 라인 번호에 해당하는 decre 함수까지 실행하고 디버거 사용자 화면에 표시된다. 순차 명령을 하면 실행 라인 표시기는 decre 함수의 ″num=num+1″ 표현식으로 이동하고 디버거 사용자는 오류를 발견한다. 그리고 표현식을 ”num=num-1″로 수정할 수 있다. 계속 실행하여 함수의 끝을 만나면 현재 실행 위치를 저장하고 다시 다음과 같이 알고리즈믹 디버깅을 시작한다.

decre(in 2, out 3)? “예“
square(in 2, in 3, out 7)? ″아니오“

square 함수의 예상하는 출력 값은 12이다. 따라서 디버거 사용자는 “아니오”를 답한다. square 함수는 비단말 노드이므로 알고리즈믹 디버깅을 계속 수행한다,

incre(in 2, out 3)? “예“
incre(int 3, out 4)? ″예“

단말노드에는 오류가 없기 때문에 그의 부모 노드인 square 함수에 오류가 있음을 알 수 있다. 다시 순차 디버깅을 진행하면 [그림 11]에서 square 함수로 이동하고 순차 디버깅에 의해 ″c=a+b″ 표현식이 잘못 되었음을 발견할 수 있다. 이와 같이 첫째 실행 트리를 이용하여 가능한 모든 노드를 검색하였고, 두 개의 오류를 발견할 수 있었다.



[그림 13] 재실행 트리

프로그램 변환 트리를 재실행하여 생성된 재실행 트리는 [그림 13]과 같다. [그림 12]와 비교해서 노드의 수가 많이 감소하였음을 알 수 있다. 디버깅을 시작하여 알고리즈믹 디버깅을 적용하면 다음과 같다.

compute(in 2, in 3, out x)? ″무시“
subsum(in 2, in 3, out 3)? ″예“
square(in 2, in 3, out 12)? ″예“

프로그램의 오류가 없음을 확인하였다. 재실행 트리에서 “예”라고 답한 addsum 함수는 제거되었고 그 결과 디버거 사용자가 디버거 시스템에게 대답하는 횟수를 줄일 수 있다.

5. 결론

시스템 관리 프로그램은 테스트 시스템 하에서 테스트 기계를 제어하고 프로그램을 컴파일 하여 실행하는 프로그램이다. 디버깅 시스템은 프로그램이 실행 시간에 발생할 수 있는 오류를 수정하기 위한 시스템으로 시스템 관리 프로그램에서 매우 중요한 시스템이다. 그러나 기존의 시스템 관리 프로그램에는 디버깅 시스템이 없어 상용 컴파일러의 디버깅 시스템을 사용하였다. 이는 산업 현장에서 엔지니어의 대한 일의 능률을 감소시키는 결과로 나타났다.

프로그램을 디버깅하기 위한 대표적인 기술로 순차적으로 프로그램을 디버깅하는 순차적 디버깅 방법과 하향식 기법을 이용한 알고리즈믹 디버깅 이론이 있다.

순차적으로 프로그램을 디버깅하는 순차적 디버깅 방법은 오류를 찾기 위해 모든 표현식을 순차적으로 실행하면서 변수 창 등을 이용하여 오류를 찾는 방법이다. 이 방법은 디버거 사용자의 명령에 의해 디버깅이 진행되며 모든 값의 변화

를 사용자가 알아야 한다. 따라서 디버거 사용자에게 너무 많은 요구를 하는 단점이 있다.

알고리즈믹 디버깅 이론은 프로그램의 시작부터 종료까지 모든 프로시저 또는 함수의 호출관계를 실행 트리로 구성하여 실행 트리의 상위 레벨부터 하위 레벨로 탐색을 하고 오류를 발견하는 반자동화 디버깅 이론이다. 그러나 이 방법은 오류가 있는 프로시저까지만 찾을 수 있고 한번에 하나의 오류만 찾을 수 있다.

본 논문에서는 시스템 관리 프로그램 내에서 사용자가 디버깅을 좀 더 쉽게 할 수 있는 혼합 디버깅 시스템을 개발하였다. 혼합 디버깅 방법은 알고리즈믹 디버깅 이론과 순차적으로 프로그램을 디버깅하는 순차적 디버깅을 혼합하여 사용하는 방법이다. 이 방법은 알고리즈믹 디버깅을 사용하여 먼저 오류를 생성한 함수를 찾아내고, 순차적 디버깅 방법을 이용하여 오류가 발견된 함수내의 표현식들을 순차적으로 실행함으로서 오류가 있는 표현식까지 발견한다. 형제 노드들 또는 형제 노드가 없을 경우에는 부모 노드의 형제 노드들을 알고리즈믹 디버깅 이론으로 추적한다. 이와 같이 알고리즈믹 디버깅 이론과 일반적인 디버깅 방법을 반복 수행함으로써 실행 트리를 한 번만 만들고 프로시저를 모두 검사할 수 있다. 또한 실행 트리를 재구성할 때 이전 과정에서 "예"라고 대답한 트리는 모두 제거함으로서 디버거 사용자가 쉽게 디버깅을 할 수 있도록 하였다.

향후 연구로는 디버거 사용자가 쉽게 이용할 수 있는 에디터의 연구와 혼합 디버깅 방법을 함수언어를 기반으로 한 병렬처리 시스템에 적용하는 방법의 연구와 함수언어를 기반으로 다중스레드 모델과 자바 바이트 코드로 변환하여 자바 가상기계에 적용하는 연구가 필요하다.

감사의 글

본 논문의 연구는 2000년도 정보통신부 대학기초연구지원 사업에 의해 수행되었으며 관계자에게 감사드립니다.

참고문헌

[1] M. Auguston, "Assert checker for the C programming language based on computations over event traces" AADEBUG 2000

[2] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing," ACM LOPLAS -- *Letters of Programming Languages and Systems*. Vol. 1, No. 4, December 1992.

[3] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models of Automated Debugging," *The Journal of Systems and Software 25*, 1994, pp 223-239

[4] E. Shapiro, "Algorithmic Program Debugging, " MIT Press, May 1982

[5] M. Weiser, "Program Slicing," IEEE Transaction on Software Engineering, Vol. Se-10, No. 4, pp 352-357, July 1984

[6] Kab-Lae Lee, Ki-Tae Kim, Chang-Mo Yang, Weon-Hee Yoo, "Design of Translator for generating Java Bytecode from Thread code of Multithreaded Models", DOCS'2000 (The 2000 International Conference on distributed Objects in Computational science), pp 839-844, 2000

[7] Teradyne Inc. "PASCAL/STEPS Reference Manual ," Boston, 1985

[8] STATEC Inc. "AZ400 Manual ," Korea, 2000

[9] 고훈준, 권영필, 양창모, 유원희, "향상된 스레드를 위한 중간언어와 그래프 생성," 한국산업기술 학회지, 제 1권 창간호, pp67-74, 1997년 7월

[10] 고훈준, 안용균, 조선문, 유원희, "Test System 용 가상 기계 설계," '2001춘계 학술발표 논문집, 한국정보처리학회, pp.255~258, 2001

[11] 고훈준, 류진수, 김기태, 유원희, "Test System 용 언어 및 번역기 설계," '2001춘계 학술발표논문집, 한국정보과학회, 2001.

고훈준

1998년 2월 인하대학교 생물공학과 졸업(학사).
2000년 2월 인하대학교 전자계산공학과 졸업(석사).
2000년 12월 ~ 현재 인하대학교 전자계산공학과 박사과정.
관심분야: 컴파일러, 병렬처리, 디버거, 프로그래밍 언어 등임

유원희

1975년 서울대학교 공과대학 응용수학과 졸업.
1978년 서울대학교 대학원 계산학 전공(이학석사).
1985년 서울대학교 대학원 계산학 전공(이학박사)
1992년 ~ 1993년 University of California, Irvine 객원연구원.
1979년 ~ 현재 인하대하교 전자계산공학과 교수.
관심분야: 프로그래밍 언어, 컴파일러, 실시간 시스템, 병렬 시스템 등임.