碩 士 學 位 論 文

목적 지향 콘콜릭 테스팅

Goal-oriented Concolic Testing

2011年

漢 城 大 學 校 大 學 院

컴 퓨 터 工 學 科

컴 퓨 터 工 學 專 攻

朴 正 圭

碩 士 學 位 論 文
指導敎授 丁仁相

목적 지향 콘콜릭 테스팅

Goal-oriented Concolic Testing

2010年 12月 日

漢城大學校 大學院
컴 퓨 터 工 學 科
컴퓨터工學專攻
朴 正 圭

碩 士 學 位 論 文
指導敎授 丁仁相

목적 지향 콘콜릭 테스팅

Goal-oriented Concolic Testing

위 論文을 工學 碩士學位論文으로 提出함

2010年 12月 日

漢城大學校 大學院
컴 퓨 터 工 學 科
컴 퓨 터 工 學 專 攻
朴 正 圭

朴正圭의 工學 碩士學位論文을 認准함

2010年 12月 日

審査委員長 김 일 민 印

審 査 委 員 정 인 상 印

審 査 委 員 이 민 석 印

목 차

【 표 목 차 】

【 그 림 목 차 】

요 약

콘콜릭 테스트는 높은 테스트 커버리지를 달성하기 위해 실제 프로그램의 실행(Concrete Execution)과 심볼릭 실행(Symbolic Execution)을 결합하여 테스트 데이터를 생성한다. CREST는 콘콜릭 테스트를 구현한 open-source로 C/C++로 작성된 프로그램을 테스트할 수 있는 대표적인 테스트 도구이다. 그러나 현재 CREST는 기본적으로 프로그램의 실행가능한 모든 경로를 탐색하는 것을 목적으로 하고 있기 때문에 특정 분기 또는 블록만을 테스트하는 경우에는 비효율적일 수 있다.

본 논문에서는 프로그램 상의 한 분기 또는 블록을 지정하고 이를 실행할 수 있는 테스트 데이터를 생성하는 목적 지향 콘콜릭 테스트 방법을 제안한다.

제 1 장 서 론

제 1 절 연구 동기

테스트 데이터 생성 방법은 사용자가 테스트할 프로그램 경로를 제공하는지 그 여부에 따라 경로 지향(path-oriented) 방법과 목적 지향(goal-oriented) 방법으로 분류할 수 있다. 경로 지향 방법은 프로그램의 제어흐름이 복잡하거나 반복문이 존재하는 경우에 경로를 선정하는 작업 자체가 용이하지 않을 뿐만 아니라 주어진 프로그램의 경로가 실행이 불가능한 경우, 즉 경로를 실행할 수 있는 입력 값이 존재하지 않을 경우에는 입력 값을 찾기 위해 많은 시간과 노력이 소요될 수 있다는 단점이 있다.

이와는 달리 목적 지향(goal-oriented) 방법은 특정 프로그램의 경로를 제공하는 대신에 프로그램 상의 한 블록 또는 분기를 지정하고 이를 실행할 수 있는 테스트 데이터를 생성하는 방법이다. 따라서 사용자가 일일이 프로그램 경로를 선정하는 부담이 없으며 경로 기반 테스트에서는 사용자가 정한 프로그램 경로가 실행 불가능하다면(즉, 주어진 경로를 실행할 수 있는 테스트 데이터가 존재하지 않는다면) 사용자가 다른 경로를 선택해야 하였으나 목적 지향 테스트에서는 주어진 프로그램 포인트(i.e., 블록 또는 분기)를 실행할 수 있는 다른 경로를 자동으로 탐색하여 입력 값을 생성할 수 있다.

최근에는 특정 프로그램 경로나 포인트 대신에 프로그램의 모든 경로들을 탐색하는 콘콜릭 테스트가 제안되었다. 이 방법은 무작위로 생성된 입력으로 프로그램을 수행한다. 이 때 입력에 의해 실행된 경로를 따라 심볼릭 실행을 하여 프로그램의 경로 제약 조건을 생성한다. 이렇게 생성된 경로 제약 조건은 프로그램의 커버리지를 높이기 위해 이전과는 다른 프로그램 경로를 수행할 수 있는 테스트 데이터를 산출하도록 수정한다.

CREST는 콘콜릭 테스트를 기반으로 작성된 open-source로서 C프로그램을 대상으로 테스트 데이터를 자동 생성하는 대표적인 테스트 도구이다. 그러나 현재 CREST는 기본적으로 프로그램의 실행 가능한 모든 경로를 탐색하는 것을 목적으로 하기 때문에 특정 분기 또는 블록만을 테스트하는 경우에는 비효율적일 수 있다. 본 논문에서는 프로그램 상의 한 블록 또는 분기를 지정하고 이를 실행할 수 있는 테스트 데이터를 생성하는 목적 지향 콘콜릭 테스트(GCT) 방법을 제안한다.

제 2 절 연구 목적

본 논문에서 제안된 방법(GCT)은 프로그램의 자료 흐름 정보를 이용하여 주어진 프로그램 포인트의 실행에 영향을 미치는 문장들을 식별한다. 실험 결과를 통해 제안된 기법이 효율적으로 테스트 데이터를 생성함을 증명하는데 목적을 두며 세부적인 내용은 다음과 같다.

목적 지향 콘콜릭 테스트의 동작원리 이해 및 구현
CREST에서 기본적으로 제공하는 7가지 strategy(dfs, cfg, hybrid등)에 대해 분석하고 목적지향 콘콜릭 테스트의 동작원리를 이해하여 구현, strategy에 GCT 알고리즘을 추가한다.

경로 제약 조건 생성 시 문제점 분석 및 해결
프로그램 경로를 결정하는 분기 또는 블록에서 경로 제약 조건으로 표현 되지는 않지만 경로를 결정짓는 요소(즉, 내부변수)가 발생했을 때 이를 해결할 수 있는 방법을 찾는다.

GCT 알고리즘의 효율성 증명
예제 코드를 대상으로 GCT 알고리즘을 이용해 테스트 데이터를 생성하여 그 결과(실행회수, 실행시간)를 측정, 다른 일고리즘에 비해 효율적임을 보인다.

제 3 절 논문 구성

본 논문의 구성은 다음과 같다.

제 2 장 연구 배경
콘콜릭 테스트를 이용한 테스트 데이터 생성 방법에 대해 알아보고 CREST의 구성 요소, 동작원리에 대해 기술한다.

제 3 장 목적 지향 콘콜릭 테스트
목적 지향 콘콜릭 테스트에 사용되는 용어를 정리하고 예제를 대상으로 그 동작원리에 대해 기술한다.

제 4 장 Goal-oriented Concolic Testing(GCT) 구현
CREST에서 제공하는 strategy중 기본적인 동작 원리가 비슷한 DFS 알고리즘을 모델로 삼아 GCT 알고리즘을 구현하고 그 구조 및 내용에 대해 기술한다.

제 5 장 실험 결과
예제 프로그램을 대상으로 GCT 알고리즘을 이용해 테스트 데이터를 생성하여 그 결과를 기술한다.

제 6 장 결론 및 향후 연구
본 논문에서 제안한 목적 지향 콘콜릭 테스트 방식에 대한 결론을 내리며 향후 연구에 대하여 기술한다.

제 2 장 연구 배경

제 1 절 콘콜릭 테스트

최근에 제안된 콘콜릭 테스트는 특정 프로그램 경로나 포인트 대신에 프로그램의 모든 경로들을 탐색하여 테스트 데이터를 생성한다. 이 방법은 실제 프로그램의 실행(동적 테스트 방법)과 심볼릭 실행을 결합하여 높은 테스트 커버리지를 달성하기 위해 개발되었다. 콘콜릭 테스트는 무작위로 생성된 입력으로 프로그램을 실행한다.

이 때, 입력에 의해 실행된 경로를 따라 심볼릭 실행을 하여 프로그램 경로 제약 조건을 생성한다. 이렇게 생성된 경로 제약 조건은 프로그램의 커버리지를 높이기 위해 이전과는 다른 프로그램 경로를 수행할 수 있는 테스트 데이터를 산출하도록 수정된다. 이러한 과정은 프로그램 상에 존재하는 모든 경로가 실행되거나 사용자가 지정한 종료 조건을 만족할 때 까지 반복된다.

1.1 콘콜릭 테스트 수행과정

아래의 [그림 1]과 같이 C로 작성된 예제 프로그램을 대상으로 5번 분기를 실행하는 테스트 데이터를 생성하고자 할 때, 콘콜릭 테스트의 수행 절차에 대해 알아보자.

표 2.1 C로 작성된 예제 프로그램

```
1:      void foo(int x, int y){
2:              int z = 2*y;
3:              if(x==1000){
4:              if(x<z){
5:                      fprintf(stderr,"GOAL!\n");
6:                      }
7:              }
8:      }
```

랜덤으로 입력 변수 x, y값에 대한 테스트 데이터를 생성한다(과정1). 이때 생성된 x와 y값이 0이라고 가정한다면 2번 분기에서 z의 값이 0으로로 정의되고 3번 분기의 조건인 0≠1000을 통과하지 못하고 테스트에 실패한다(concrete execution). 이때 프로그램의 경로는 [그림 2]의 ①과 같다. 이와 동시에 심볼릭 실행(symbolic execution)을 수행하며 x와 y를 심볼릭 변수로 취급하여 z=2y로 표현하고 3번 분기의 조건에 사용된 부등식 x≠1000을 기록한다(과정2). 이 식을 경로 제약식이라 한다.

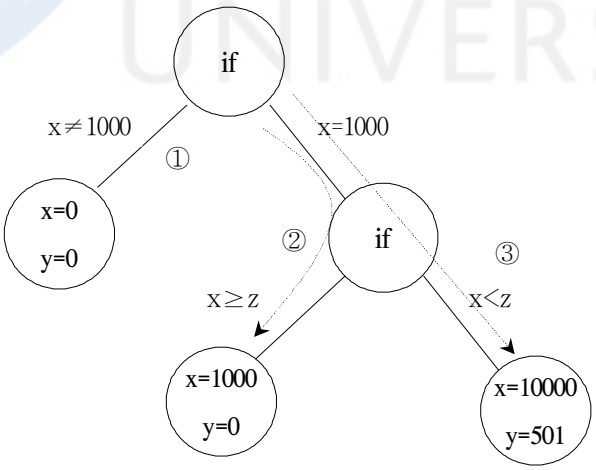

그림 2.1 예제 프로그램의 제어 흐름 그래프

그리고 이전에 실행했던 프로그램 경로의 제약 조건식 x≠1000에 부정(negate) 연산을 수행하여 x=1000을 도출한다(과정3). 이렇게 수집한 경로 제약 조건을 만족하는 테스트 데이터를 생성하면 x=1000, y=0이 된다(과정4). 이때 프로그램의 경로는 [그림 2]의 ②와 같아 여전히 테스트에 실패하게 된다. 새롭게 생성된 테스트 데이터를 이용해 기존의 경로 제약식에 x≥z의 부정(negate) 연산의 결과인 x<z를 추가하여 z=2y, x=1000, x<z라는 경로 제약식을 도출해 낸다(과정5). 마지막으로 이 제약식을 모두 만족하는 테스트 데이터를 생성하게 되고 x=1000, y=501이 되어 분기 5를 실행하게 된다.

제 2 절 CREST

CREST는 UC Berkely 대학에서 재직 중이던 Koushik Sen 교수에 의해 개발된 콘콜릭 테스트 도구이다. 이 도구는 심볼릭 수행을 위해 C로 작성된 프로그램을 입력으로 받아 CIL를 사용하여 분석하기 단순한 프로그램의 형태로 변환하고 CREST가 생성한 입력에 따라 프로그램을 수행시키면서 프로그램 실행 경로가 어떤 분기를 따라가는지 기록한 심볼릭 경로 제약식을 생성한다. 이전 프로그램의 실행 경로를 탐색하기 위해 심볼릭 경로 조건식의 심볼릭 표현식을 부정하여 이전 경로와 다른 분기를 따라가게 될 새 경로 조건식을 만들고 선형 정수 조건식을 지원하는 SMT solver를 사용하여 새로운 제약식을 만족하는 테스트 데이터를 생성한다.

2.1 심볼릭 경로 제약식 생성

CREST는 C로 작성된 프로그램의 소스코드를 입력으로 받아, 먼저 심볼릭 입력을 생성하고 프로그램 실행 경로에 따른 심볼릭 경로 조건식을 생성할 수 있도록 CIL[4]을 사용하여 입력 프로그램의 소스 코드를 수정(Instrumentation)한다. 대상 프로그램의 구문마다 심볼록 변수가 어떻게 바뀌고 어떤 실행 경로를 따라가는지 기록하기 위한 프로브(probe)를 삽입한다.

수정된 프로그램은 CREST가 설정한 심볼릭 입력 값에 따라 실제로 실행되면서 실행 경로를 따르는 심볼릭 경로 제약식을 기록한다. 실행이 종료되면 경로 제약식을 이용하여 다음 테스트 데이터를 생성하는데 사용한다. 프로그램 실행이 처음 시작된 경우에는 각 입력변수에 대한 심볼릭 표현식은 어떤 배정문도 실행되기 전이기 때문에 단순한 심볼릭 값을 갖게 된다. 수정된 프로그램이 실행되면서 심볼릭 수행을 통해 심볼릭 경로 제약식을 생성하고 각 심볼릭 변수는 심볼릭 맵을 통해 관리되며 각 배정문이 실행될 때 마다. 배정문에 해당하는 표현식은 갱신된다.

심볼릭 맵 외에도 실행 경로에 따른 경로 제약식을 저장하기 위한 자료 구조 Φ를 관리한다. Φ는 초기 값으로 'true'가 주어진다. 만약 조건문 'if p then...'을 처리하는 경우에 실제 실행된 경로가 p가 참이 될 때에는 조건식 p에 해당하는 심볼릭 표현식 ξ(p)이 현재 Φ에 있는 식과 'and'로 결합되어 Φ에 저장된다.

실행된 경로가 p가 거짓인 경우에는 심볼릭 표현식 'not ξ(p)'이 결합된다. 즉, Φ를 만족하는 입력 값은 지금까지 실행된 경로를 실행하는 테스트 데이터가 된다.

CREST는 이전에 실행된 프로그램 경로와는 다른 경로를 실행하기 위해 Φ를 구성하는 심볼릭 표현식에서 하나를 선택하여 부정하여 새로운 경로 제약식을 생성한다. 예를 들어 <p0, p1,..., pk-1>이 실제 실행된 경로라고 하자. CREST는 분기 pj(0≤j<k)를 선정한다. 본 논문에서는 분기를 테스트 문장과 테스트 문장이 참 또는 거짓이 될 때 실행되는 첫 번째 문장으로 구성된 쌍으로 정의한다. 이 때 ϕ_l을 이 분기를 수행하기전의 경로제약식이라 하고 ϕ_t를 이 분기에 의해 생성된 심볼릭 표현식이라고 하자.

CREST는 이전 경로와는 다른 경로를 실행하는 테스트 데이터를 찾기 위해 $(\wedge \phi \in \Phi | \phi) \wedge (\text{not } \phi_t)$를 만족하는 해를 구한다. 만약 이 새로운 경로제약식을 만족하는 입력 값으로 프로그램을 실행하면 실행 경로는 <p0, p1,...,paired(pj), p′j+1, ..., p′m>일 것이다. 여기에서 paired(pj)는 pj의 짝 분기를 나타낸다. 즉, 어떤 조건식에서 pj가 참인 분기라면 paired(pj)는 거짓인 분기를 나타내며 pj가 거짓인 분기라

면 paired(pj)는 참인 분기를 나타낸다. CREST는 이 과정을 모든 경로들에 대한 테스트 데이터를 찾을 때까지 반복하거나 지정한 회수만큼 반복한다.

2.2 선형 정수 심볼릭 경로 조건식

CREST가 생성하는 경로 제약식은 선형 정수 표현식으로 표현된다. 선형 정수 표현식에서 각 변수와 상수의 타입은 크기 제한이 없는 정수이고 각 표현식은 일차식의 형태를 갖는다. 즉, 사칙 연산중 변수와 변수의 곱셈 연산이나 나눗셈 연산은 처리할 수 없으며 C 에서 사용하는 bit-wise 연산(&, |, ^등)을 지원하지 않는다. 프로그램을 실행하여 심볼릭 경로 조건식을 만들 때 선형 정수 표현식으로 표현할 수 없는 C 표현식을 만나면 실행 당시 계산된 실제 변수 값을 사용하여 선형 정수 표현식으로 간소화하여 처리한다.

선형 정수 심볼릭 경로 제약식을 사용했을 때 가지는 장점은 상대적으로 빠른 시간에 문제를 풀 수 있다는 점이다. 선형 정수 조건식을 효율적으로 풀기위한 알고리즘[9]이 많이 연구되어 왔으며 이를 구현한 SMT solver 튜닝 기술 또한 발전하였다. 실제 수학 연산을 이진 회로로 인코딩해서 표현하는 것이 아니라 CPU 연산으로 계산하기 때문에 bit-vector를 사용하는 것 보다 더 빨리 조건식을 풀 수 있다.

2.3 CREST의 구성 요소

다음 그림은 CREST의 중요한 구성 요소를 보여준다.

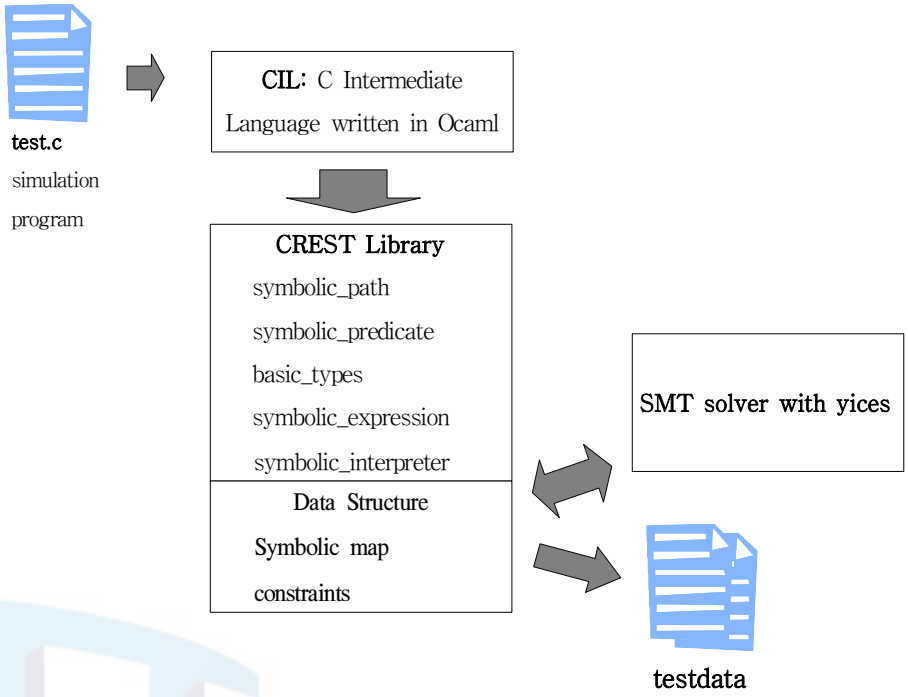

그림 2.2 CREST의 구성도

먼저 Ocaml로 작성된 CIL(C Intermediate Language)언어를 사용하여 테스트 대상이 되는 프로그램에 프로브를 삽입하여 분석하기에 단순한 프로그램으로 변환한다. 그리고 랜덤으로 생성된 테스트 데이터를 이용해 프로그램을 실행하며 경로 제약 조건식을 생성하고 이를 만족하는 테스트 데이터를 생성하기 위해 SMT solver를 이용해 제약 조건을 만족하는 테스트 데이터를 생성하고 이 과정을 반복하며 프로그램 상에 존재하는 모든 경로를 실행하고 테스트를 종료하게 된다.

가. CIL(C Intermediate Language)

CIL은 C프로그램의 코드를 변환하여 분석하기 쉽게 나타내는 일종의 표현 방식이다. 복잡한 C코드의 구조를 나누어 하나의 간단한 구조로 나타내기 때문에 C에 비해 작은 구조를 가지며 low-level에서 처리가 용이하다.

나. CREST Library

CREST에는 다수의 C 또는 C++로 개발된 라이브러리가 포함되어 있으며 입력 프로그램을 분석하여 제어 흐름 그래프(Control Flow Graph)를 생성하거나 테스트 데이터를 이용해 실제 프로그램을 수행하며 경로 제약 조건식을 생성하는데 사용된다.

다. Yices solver

Yices는 SRI 인터내셔널에서 개발한 고성능의 SMT 처리기이다[9]. yices 주어진 명제식의 만족성(Satisfiability)을 결정하며 이러한 명제식들은 선형 실수 이론 및 선형 정수 이론, 배열, 고정 크기의 비트-벡터, 재귀적 데이터 타입, 튜플, 레코드, 람다-식, 한정자등을 지원한다.

Yices의 입력언어는 SAL(Symboloc Analysis Laboratory)언어와 유사하다. 또한 SMT-LIB 언어로 쓰여진 명세도 입력으로 받아들일 수 있다. 일반적으로 SAT 처리기로 사용할 수도 있고 CNF DIMACS 포맷으로 된 불리언 문제들도 처리할 수 있다. Yices에서 사용하는 입력 언어는 표 2.2와 같다.

선언문(Declarations)

선언문의 종류는 아래 표와 같다.

표 2.2 선언문

선 언 문	구 문	설 명
타입 선언	(define type [name])	해석되지 않는(uninterpreted) 타입
타입 정의	(define type [name] [type])	타입을 정의한다. [type]은 잘 정의된(well-formed) 타입 표현(type expre-ssion)이다. [name]은 새로운 식별

		자이고,이것은 타입 표현 [type]에 대한 별칭이 된다.
상수 정의	(define [name]::[type])	상수, 함수, 술어 기호들은 모두 상수처럼 사용된다. [name]은 새로운 식별자이고 [type]은 잘 정의된 타입 표현이다.
상수 정의	(define [name]::[type] [expr])	[name]은 새로운 식별자이고 [type]은잘 정의된 타입 표현이다.

타입(Types)

Yices의 언어는 엄격히 타입화 되어 있으며 이는 모든 표현들이 연관된 타입으로 되어 구성되어 있다는 의미이다. Yices의 타입 시스템은 이름의 동일성 대신 구조적 동일성을 기반으로 한다. 따라서 타입은 집합과 밀접하게 관련이 있고 두 개의 타입이 동일하다면 같은 원소를 가진다. 타입은 표 2.3과 같다.

표 2.3 타입

타 입	구 문	설 명
기본 타입	real, int, nat, bool	기본 타입은 real, int, nat, bool이 있다. real은 실수 타입이고, nat는 자연수, bool은 불리언 표현을 나타내는 타입이다. Yices에서는 리식과 텀 사이에 차이는 없다.
서브 타입	(subtype ([id]::[type]) [expr])	타입 자체가 특정 유형의 입력 타입의 요소가 되는 모든 컬렉션을 서브타입이라 한다.
함수 타입	(> [domain_1] ... [domain_n] [range])	[domain_1]과 [range]는 타입이다. Yices에서는 배열이 함수처럼 표현된다.
튜플 타입	(tuple [type_1] ... [type_n])	n은 최소 2이고, [type_i]는 타입이다.
레코드 타입	(record [id_1]::[type_1] ... [id_n]::[type_n])	n은 최소 1이고, [id_i]는 식별자, [type_i]는 식별자이다. [id_i]는 레코드 타입에서의 필드 타입이라 부른다.
스칼라 타입	(scalar [name_1] ... [name_n])	데이터타입의 특별한 경우이다. n은 최소 1이고, [name_i]는 새로운 식별자이다.
비트-벡터 타입	(bitvector [size])	고정 크기 비트-벡터이다. size는 양수이다.

수식(Expressions)

수식은 불리언 연산자, 동치, if, let, 함수 및 람다 식, 한정자 및 산술 연산, 튜플과 레코드 비트-벡터로 구성되며 아래와 같다.

표 2.4 불리언 연산자, 동치와 비동치

타 입	구 문	설 명
불리언 연산자	(and [expr_1] ... [expr_n]) (or [expr_1] ... [expr_n]) (not [expr]) (=> [expr_1] [expr_2])	일반적인 불리언 연산자로는 and, or, not, =>(implication)이 있다. [expr_i]는 불리언 식이다. 동등 관계와 XOR은 동치와 비동치를 이용하여 나타낸다.
동치와 비동치	(= [expr_1] [expr_2]) (/= [expr_1] [expr_2])	동치(=)와 비동치(/=)를 나타낸다.

표 2.5 if-then-else, Let

타 입	구 문	설 명
if-then-else	(if [c-expr] [t-expr] [e-epxr]) (ite [c-expr] [t-expr] [e-epxr])	조건식을 나타낸다. [c-expr]은 불리언 식이고 [t-expr]과 [e-epxr]은 조건의 타입과 같이 동일한 타입을 요구한다.
Let	(let ([binding_1] ... [binding_n]) [expr])	n은 최소 1이고 각각의 binding_i]는 아래와 같은 형식을 갖는다. ([name]::[type] [expr]) (name expr)

표 2.6 한정자, 산술

타 입	구 문	설 명
한정자	(forall ([name_1]::[type_1] ... [name_n]::[type_n]) [expr]) (exists ([name_1]::[type_1] ... [name_n]::[type_n]) [expr])	수량을 나타내는 식은 forall과 exists를 사용한다. 이러한 식의 타입은 bool이다.
산 술	(define x::int) (define y::int) (define t1::bool (or (< x y) (> x y) (= x y))) (define t2::bool (=> (and (= (+ x (* 2 y)) 5) (>= x 1)) (<= y 2)))	정수와 실수 간의 부등식은 <, <=, >, >= 기호를 사용한다. 부등식의 타입은 bool이 다. 기호 +, , *, /, div, mod는 각각 덧셈, 뺄셈, 곱셈, 나눗셈, 정수 나눗셈, 모듈로 연산에 사용된다.

표 2.7 함수, 람다 식

타 입	구 문	설 명
함 수	(f a_1 ... a_n)	인자값 a_1, ... , a_n을 함수 f에 적용한 다. f는 함수 기호가 없음을 주의하라. 함수는 update 생성자를 이용하여 기능적으로 업데이트 할 수 있다. 구문은 아래와 같다. (update f pos_1 ... pos_n) new-val)
람다 식	(lambda ([name_1]::[type_1] ... [name_n]::[type_n]) [expr])	람다-식은 이름이 없는 함수로 나타낸다.

표 2.8 튜플, 레코드

타 입	구 문	설 명
튜플	(mk-tuple [expr_1] ... [expr_n])	n은 최소 2이고 [expr_i]는 식이다. 튜플 t의 i번째 구성요소를 선택하는 구문은 아래와 같다. (select t i) i는 숫자이고 컴포넌트의 번호는 1부터 시작한다. 튜플은 update 생성자를 사용하여 기능적으로 업데이트 할 수 있다.문법은 아래와 같다. (update [tuple] [index] [new-val])
레코드	(mk-record [name_1]::[expr_1] ... [name_n]::[expr_n])	레코드는 update 생성자를 사용하여 기능적으로 업데이트 할 수 있다. 문법은 아래와 같다. (update [record] [field-name] [new-val])

표 2.9 비트-벡터, 워드-레벨 함수

타 입	구 문	설 명
비트-벡터	(mk-bv [size] [value])	size(비트의 개수)는 양의 정수이고 value는 0을 포함한 양의 정수이다.
워드-레벨 함수	(bv-concat [bv1] [bv2])	비트 결합
	(bv-extract [end] [begin] [bv])	비트 추출
	(bv-shift left0 [bv] [n]) (bv-shift left1 [bv] [n])	왼쪽으로 이동
	(bv-shift right0 [bv] [n]) (bv-shift right1 [bv] [n])	오른쪽으로 이동
	(bv-sign extend [bv] num-copy])	부호 확장

2.4 CREST를 이용한 콘콜릭 테스트 수행 과정

표 2.1에 작성된 예제 프로그램을 대상으로 테스트를 수행한다고 가정하면 입력으로 들어갈 foo.c코드를 표 2.10과 같이 수정하여 CREST에서 사용할 수 있도록 설정한다. CREST에서 제공하는 함수를 사용하기 위해서 헤더파일을 include하고 입력 값에 사용될 변수, 즉 x, y의 값이 정수형이라는 사실을 명시하기 위해 CREST_int( ) 함수를 사용한다.

표 2.10 실제 사용을 위해 수정된 예제 프로그램

```
#include <crest.h>
#include <stdio.h>

int main(void){
    int x, y, z;

        CREST_int(x);
        CREST_int(y);

        z = 2*y;

        if(x==1000){
                if(x<z){
                        fprintf(stderr,"GOAL!\n");
                }
        }
}
```

그리고 나서 CIL을 이용해 프로브를 삽입하면 변환된 결과가 그림 2.3과 같이 foo.cil이라는 파일이 생성되며 변환된 코드에 사용된 함수들을 살펴보면 아래와 같다.

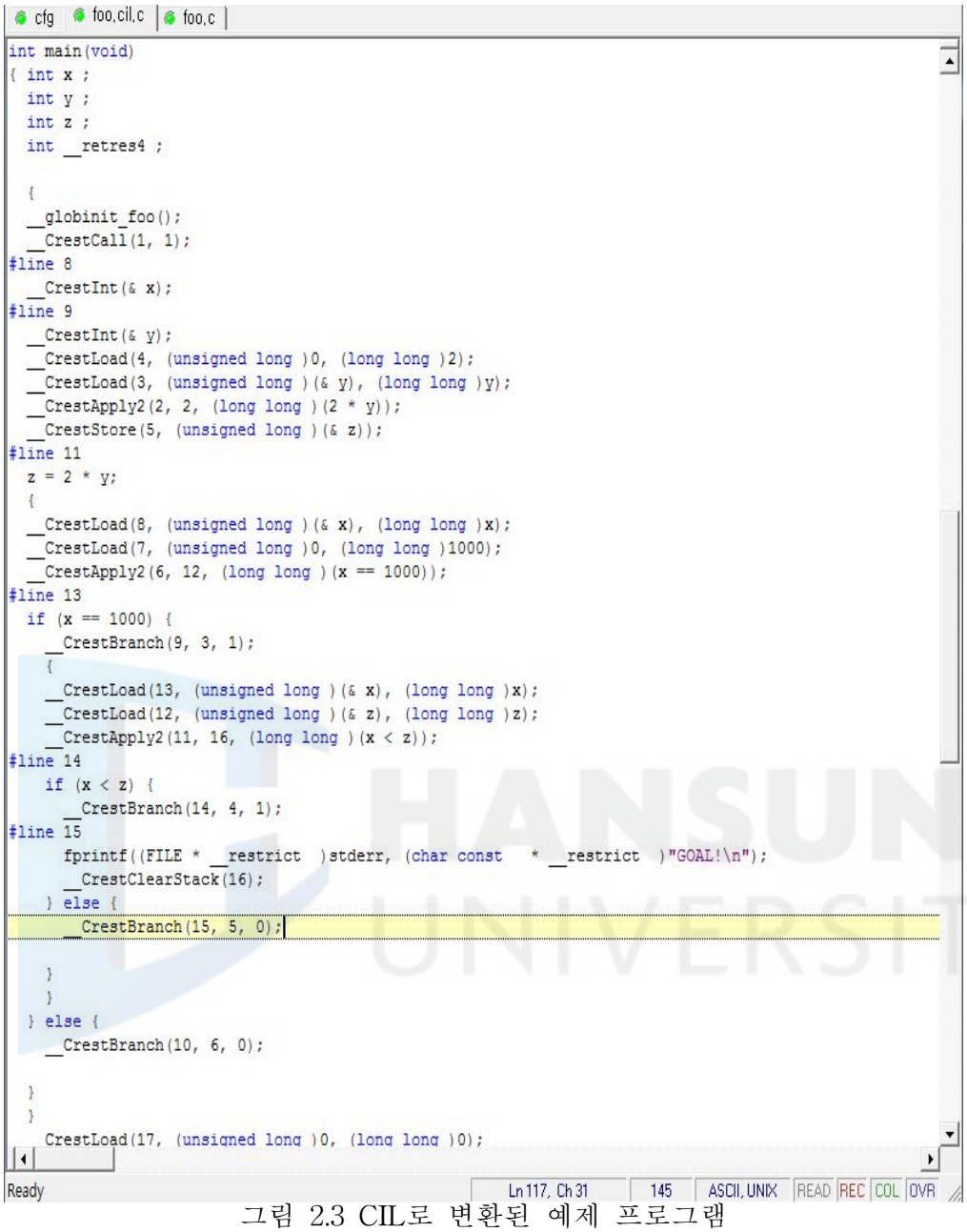

그림 2.3 CIL로 변환된 예제 프로그램

__CrestInt(& x)

int형 변수 x값을 처리하기 위해 메모리에 할당

__CrestLoad(__CREST_ID, __CREST_ADDR, __CREST_VALUE)
변수를 사용하기 위해 __CREST_ADDR에 있는 __CREST_VALUE를 로드

__CrestStore(__CREST_ID, __CREST_ADDR)
사용한 변수를 __CREST_ADDR에 저장

__CrestApply2(__CREST_ID, __CREST_OP, __CREST_VALUE)
연산자(CREST_OP)에 따라 연산을 수행

__CrestBranch(__CREST_ID, __CREST_BRANCH_ID, __CREST_BOOL)
__CREST_BRANCH_ID를 가지는 분기 생성

또한 CREST를 실행하면 foo.cil파일 이외에도 표 2.11과 같은 cfg파일이 생성된다. 이 파일은 예제 프로그램에 대한 제어 흐름 그래프에 관한 정보를 담고 있으며 이 정보는 본 논문에서 구현할 GCT 알고리즘에서 활용될 것이다.

표 2.11 예제 프로그램의 cfg파일

```
1 2
2 3 6
3 4 5
4 7
5 7
6 7
7 8
8
```

cfg파일에서 나타나는 숫자는 각각의 branch id를 나타내며 첫 번째 id는 부모 노드(parent)의 id를 나타내고 두 번째 id는 자식 노드(child node)의 id를 나타낸다. 위 표에 있는 제어 흐름 그래프 정보를 실제로 그려보면 그림 2.4와 같이 나타낼 수 있다.

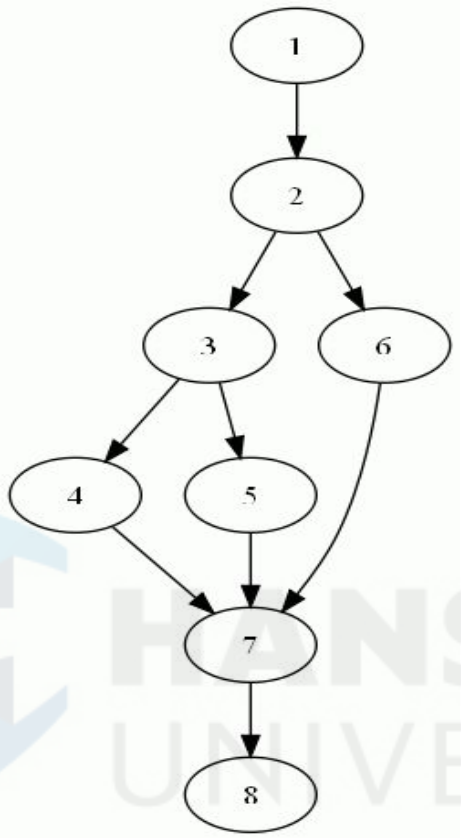

그림 2.4 예제 프로그램의 제어 흐름 그래프

생성된 파일과 symbolic map의 정보를 이용하여 아래 그림 2.5와 같이 테스트를 수행할 수 있다.

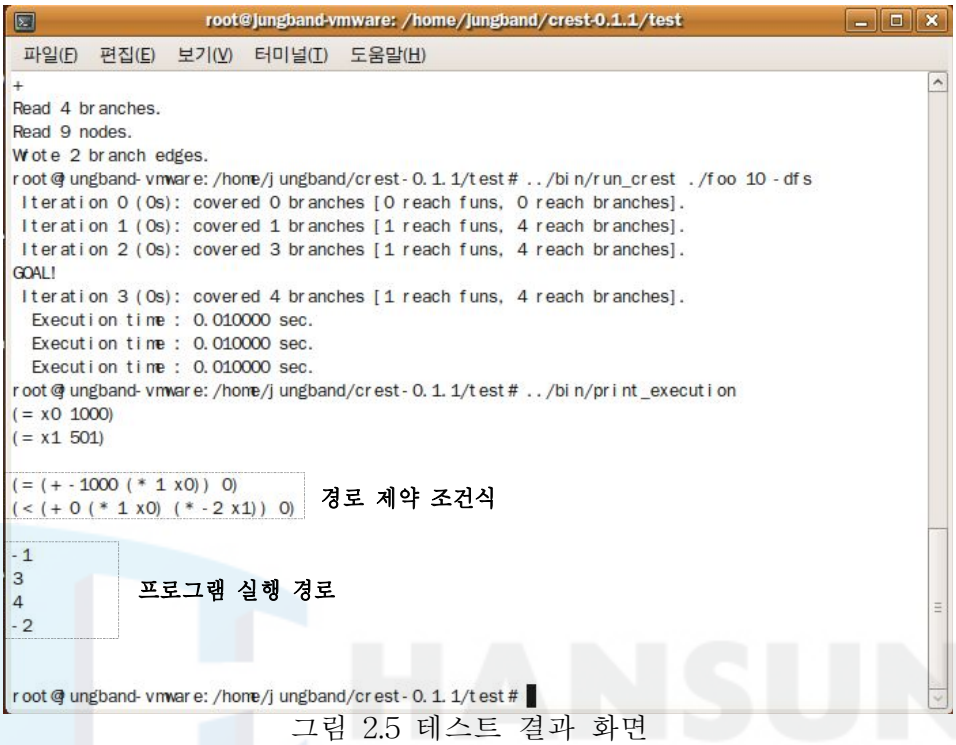

그림 2.5 테스트 결과 화면

테스트는 3번의 실행으로 프로그램상의 모든 경로를 테스트 하고 종료되며 목적 노드를 실행하는 테스트 데이터는 x=1000, y=501이 된다. 이때 경로 제약 조건식과 프로그램 실행 경로를 확인할 수 있으며 각각 그림에 표시된 것과 같다.

제 3 장 목적 지향 콘콜릭 테스트

이 장에서는 프로그램 상에 존재하는 모든 경로를 실행하는 테스트 데이터 생성 방법인 콘콜릭 테스트를 수정하여 특정 프로그램 분기를 실행하는 테스트 데이터를 찾는 목적 지향 콘콜릭 테스트 방법(GCT, Goal-oriented Concolic Test)에 대해 기술한다.

제 1 절 용어 정의

이 절에서는 목적 지향 콘콜릭 테스트, 즉 GCT 수행 절차에 필요한 용어들에 대해서 정의 한다. 분기 I에 대하여 pairend(I)이 실행 경로 π상에 있고 I를 통해 목표 분기 t에 도달할 수 있는 프로그램 경로가 존재할 때 분기 I는 실행 경로 π에 관해 유도 분기(guiding branch)라 한다. $GB_t(\pi)$는 실행 경로 π상에 존재하는 목표 분기 t에 관한 유도 분기들의 집합을 나타낸다. $D(s_i)$는 문장 s_i에서 정의된 변수 집합을 나타내고 $U(s_j)$는 문장 s_j에서 사용 또는 참조되는 변수 집합을 나타낸다. 분기 l=(p, q)에 대해 <l>은 문장 q를 통해 도달할 수 있는 문장들의 집합을 나타낸다. 즉, 테스트 문장 p에 대해 이행적으로 제어의존적(transitive control-dependent)인 문장들 등에서 q가 실행될 때 실행될 수 있는 문장들이다. 다음은 GCT가 기반으로 하고 있는 자료 흐름에 대한 정의이다.

[정의 1] 문장 s_i는 s_j에 다음 조건을 만족할 때 경로 $p=(s_1, s_2, ..., s_k)$를 통하여 직접적으로 영향을 미친다고 말한다.

$v \in U(s_i)$이고 $v \in D(s_j)(i<j)$인 변수 v가 존재한다.

모든 $s_m(j<m<i)$에 대해 $v \notin D(s_m)$

[정의 2] 경로 $p=(s_{i1}, si_2, ..., s_{in})$로부터 다음 조건을 만족하는 시퀀스 $<k_1, k_2, ..., k_r>$를 추출할 수 있다면 문장 s_{i1}는 s_{in}에 *영향을 미친다*고 말한다.

$s_{i1}=k_1$, $s_{in}=k_r$

for all j, 1<j<r, k_j는 k_{j+1}에 직접적으로 영향을 준다.

제 2 절 GCT의 수행 절차

이 논문에서 제안하는 목적 지향 콘콜릭 테스트는 자료 흐름 정보를 이용하여 주어진 분기의 실행에 영향을 미칠 수 있는 문장들을 식별한다. GCT는 이렇게 식별된 문장들의 실행을 제어하여 목표 분기를 실행하도록 유도한다. 다음은 1절에서 기술한 정의에 바탕을 두고 특정 분기를 실행하는 테스트 데이터를 찾아내는 GCT의 수행 절차이다.

1) 랜덤으로 생성된 입력을 사용하여 콘콜릭 테스트를 수행한다. 이때 실행된 프로그램 경로를 π_e라 한다.

2) 프로그램 경로 π_e가 목표분기 t를 실행하였다면 프로그램을 종료한다.

3) 만약 목표 분기 t를 실행하지 않았다면 $GB_t(\pi_e)$로부터 유도 분기(guiding branch)인 l_g를 선정한다. 본 논문에서는 목표 분기와의 거리를 선정 기준으로 사용하였으며 가장 가까운 분기를 선정하였다.

4) $l_t \leftarrow paired(l_g)$

5) 이 때 ϕ_l을 l_t를 수행하기전의 경로 제약식이라 하고 ϕ_t를 l_t에 의해 생성된 심볼릭 표현식이라고 하자.

6) $E \leftarrow (\wedge_{\phi \in \Phi_l} \phi) \wedge (not\ \phi_t)$.

7) E를 만족하는 해를 구하여 이를 입력으로 프로그램을 실행한다. 프로그램 경로가 목표 분기 t를 실행한다면 종료한다. 만약 이전 경로와 다른 경로 π_e'를 실행한다면 $\pi_e \leftarrow \pi_e'$에 대해 2)번 과정부터 다시 수행한다.

8) 만약 E를 만족하는 해를 구할 수 없다면 $GB_t(\pi_e)$가 공집합이 될 때까지 3)번 과정을 수행한다.

9) 만약 이전경로 π_e와 동일한 경로를 실행한다면 분기 l_t의 테스트 문장에 영향을 미치는 문장들을 식별한 후 이 문장들을 $AFF(l_t)$라 한다. 이 때 실행 경로 π_e상에서 분기 l_t 바로 이전에 실행된 분기 l에 대해서 다음을 검사한다.

'

9-1) 만약 분기 l가 l_t의 실행에 영향을 주었다면 즉, $<l> \cap AFF(l_t) \neq \varnothing$라면 $l_t \leftarrow l$로 수정한 후에 5)번 과정부터 수행한다.

9-2) 만약 $<l> \cap AFF(l_t) = \varnothing$이고 $<paired(l)> \cap AFF(l_t) \neq \varnothing$라면 9-1)과 마찬가지로 $l_t \leftarrow l$로 수정한 후에 5)번 과정부터 수행한다.

9-3) 만약 $<l> \cap AFF(l_t) = \varnothing$이고 $<paired(l)> \cap AFF(l_t) = \varnothing$라면 실행 경로 π_e상에서 분기 l 바로 이전에 실행된 분기 l′에 대해서 $l \leftarrow l'$로 수정한 후에 이 단계를 반복 수행한다. 만약 π_e상에 더 이상의 분기가 존재하지 않는다면 절차를 종료한다.

GCT수행 절차에서 9)번 과정은 목표분기 실행 이전에 실행해야 되는 분기들을 자료 흐름 정보를 이용하여 식별한다. 9-1)과 9-2) 과정은 현재 목표 분기에 영향을 미치는 요소들이 목표 분기 실행에 도움이 되지 않았기 때문에 목표 분기 실행에 영향을 미치는 다른 경로를 탐색하는 과정이다.

최근에 콘콜릭 테스트를 이용하여 리그레션 테스트를 수행한 연구 결과가 Rothermel등에 의해 발표되었다. 이 방법은 논문에서 제안된 방법과 유사하게 특정 분기(집합)을 실행하는 테스트 데이터를 생성한다. 그러나 다음과 같은 점에서 차이가 있다. 본 논문은 개방 과정에서 사용되는 테스트 방법인데 반해 앞에서 말한 방법은 수정된 프로그램에 대한 리그레션 테스트라는 점이다. 리그레션 테스트는 이미 한번 테스트가 되어 기존의 테스트 데이터 suite가 있다는 것을 가정하고 있지만 본 논문의 방법은 그러한 가정이 없다. 이는 만약 기존의 테스트 suite가 존재하지 않는 경우에는 콘콜릭 테스트를 이용한 리그레션 테스트 방법을 적용할 수 없다는 것

을 의미한다.

기술적으로도 목적 분기를 찾기 위해 Rothermel의 연구는 기존 테스트 데이터가 수행한 트레이스 정보를 이용하지만 본 논문에서는 자료 흐름 정보를 이용하여 목적 분기를 탐색한다.

제 3 절 예제 프로그램

본 절에서는 아래에 있는 표 3.1의 test함수에서 분기 (8, 9)를 실행하는 테스트 데이터를 GCT 알고리즘을 이용하여 찾는 과정을 살펴보도록 한다. 이 함수는 두 개의 정수형 배열a[]와 b[]를 입력으로 받으며 이때 배열의 크기는 10이라고 가정한다. 이 분기를 실행하기 위해서는 배열a[]의 원소 중 하나가 10이어야 하지만 b[]값은 상관이 없다.

표 3.1 예제 프로그램

```
      void test(int a[], int b[], int n) {
1:    i = 0;
2:    fa = 0;
3:    fb = 0;

4:    while(i < n){
5:          if(a[i] == 10)
6:                fa = 1;
7:          i = i + 1;
      }

8:    if(fa == 1){
9:          fb = 1;
10:         i = 0;
11:         while(i < n){
                  if(b[i] != 10)
12:                     fb = 0;
13:               i = i + 1;
            }
      }

14:   if(fb == 1)
```

```
15:             fprintf(stderr,"GOAL!\n"); /* 목표 분기 */
        }
```

먼저 a[0], a[1], ..., a[9], b[0], ..., b[9]에 랜덤으로 값을 할당하여 프로그램을 실행한다(단계 1). 만약 모두 0으로 할당되어 프로그램이 실행되었다면 첫 번째 실행에서 목표 분기가 실행되지 않고 분기(8, 14)가 실행된다(단계 3).

paired((8, 14))는 목표 분기 (8, 9)와 같아 이 경우에는 유도 분기로도 선정된다. 만약 a[i](0≤i≤9)와 b[i](0≤i≤9)가 각각 심볼릭 값 α_i와 β_i이 할당되었다면 다음과 같은 경로 제약식 E가 생성 된다(단계 6)

경로 제약식: $E=(\alpha_0 \neq 10) \wedge (\alpha_1 \neq 10) \wedge ... \wedge (\alpha_9 \neq 10)$

이때 변수 fa에 대한 심볼릭 표현식, i.e., not(fa≠1)은 E에 반영이 되지 않는다. 그 이유는 fa가 입력 변수에 관한 표현식으로 표현되지 않기 때문이다. 따라서 E를 만족하는 해를 구해 입력으로 프로그램을 실행하여도 이전 경로와 동일한 경로가 실행된다(단계 9).

이 단계에서 실행 경로 상에서 분기 (8, 14) 이전에 실행되었던 분기 l=('a[9]≠10', 'i=i+1')에 대해 테스트 문장(fa==1)에 영향을 미치는 문장을 조사한다. 이러한 문장은 <l>에는 존재하지 않고 <paired(l)>, i.e., 'fa=1'에 존재하기 때문에 9-2)단계가 수행된다. 따라서 5), 6)번 단계로부터 다음과 같은 새로운 경로 제약식이 생성된다.

경로 제약식: $E=(\alpha_0 \neq 10) \wedge (\alpha_1 \neq 10) \wedge ... \wedge \text{not } (\alpha_9 \neq 10)$

이 경로 제약식을 만족하기 위해서는 a[9]가 10이어야 한다. 따라서 새로운 E를 만족하는 입력 값은 목표 분기 (8, 9)를 실행 할 수 있다.

제 4 장 목적 지향 콘콜릭 테스트 구현

이 장에서는 GCT 알고리즘의 구현 내용에 대해 기술한다. CREST의 탐색 알고리즘 중에 DFS(Depth-First Search)의 코드를 수정하여 GCT 알고리즘을 구현하였다.

제 1 절 소프트웨어 구조

CREST는 탐색 알고리즘 모듈을 설계하여 search 모듈에 추가하므로서 그 기능을 확장할 수 있다. CREST의 구조는 그림 4.1과 같다.

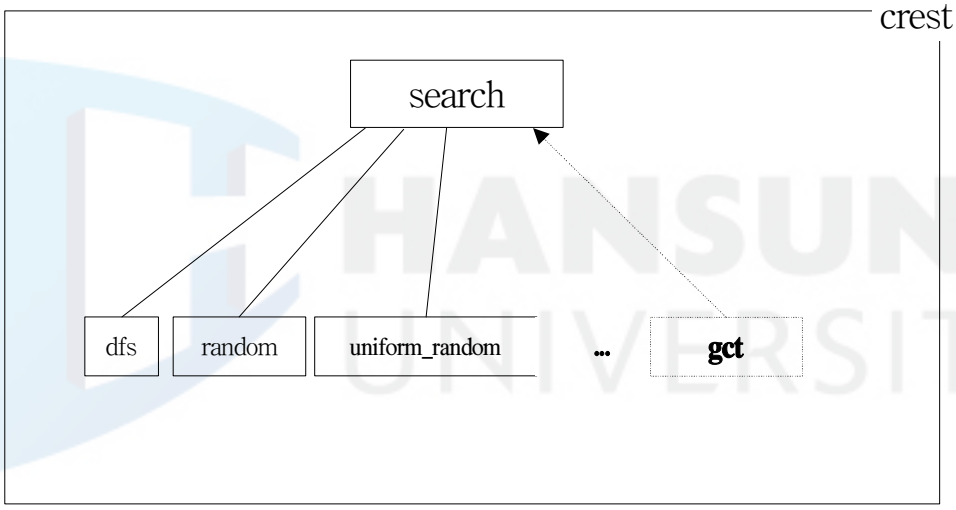

그림 4.1 CREST의 구조

본 논문에서는 GCT 모듈을 추가하여 그 기능을 확장하였으며 프로그램을 구동했을 때 충돌을 막기 위해서 기존에 사용되고 있던 주요 함수들은 그대로 사용하고 필요한 함수는 추가하였다.

제 2 절 Goal-oriented Concolic Test Module

GCT 알고리즘은 기본적으로 프로그램의 제어 흐름 그래프(Control Flow Graph)에서 정보를 추출해 관리함으로서 테스트를 수행한다. 본 절에서는 GCT 알고리즘을 구현하는데 필요한 사항들과 함수에 대해 기술한다.

2.1 CIL 파일 수정

테스트를 수행하기 전에 CREST를 실행하면 표 2.11과 같은 cfg파일이 생성된다. 하지만 이 정보만으로는 내부 변수 문제를 해결할 수가 없기 때문에 필요한 정보를 추출하기 위해서 CIL파일을 수정하여 아래와 같은 cfg파일을 생성하도록 변경한다. 아래 표를 살펴보면 각 분기에 사용된 변수가 cfg파일에 포함된 것을 확인할 수 있다. 이 정보는 자료구조에 저장되어 내부 변수 문제를 해결하는데 이용된다.

표 4.1 변경된 cfg파일

```
1 2  z
2 3 6  x
3 4 5  x z
4 7
5 7
6 7
7 8
8
```

2.2 알고리즘 수행을 위한 자료구조

아래에서는 GCT 알고리즘을 수행하기 필요한 정보 및 자료구조에 대해 기술한다.

가. cfg_

생성된 테스트 데이터로 프로그램을 실행하면서 테스트를 수행하기 위해서 제어 흐름 그래프의 정보가 필요하며 GCT 모듈이 호출될 때 표 4.2와 같이 cfg_벡터에 저장된다. 이때 cfg_벡터의 index는 부모노드의 branch_id를 의미하며 각 요소는 자식노드의 branch_id를 의미한다.

표 4.2 cfg_벡터

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
cfg		2	3, 6	4, 5	7	7	7	8	

나. cfg_rev_

본 자료구조는 내부 변수 문제 해결 시, 프로그램의 경로를 역 추적할 때 사용되며 cfg벡터의 정보를 이용하여 생성된다. 표 4.2를 이용해 생성된 cfg_rev_벡터는 표 4.3과 같다.

표 4.3 rev_cfg_벡터

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
cfg_rev			1	2	3	3	2	4, 5, 6	7

다. vars_

본 자료구조는 분기에 사용된 변수의 정보를 가지며 표 4.1을 대상으로 생성된 vars_벡터는 표 4.4와 같다.

표 4.4 vars_벡터

cfg_rev	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
		z	x	x, z					

라. paired_branch

알고리즘을 수행할 때 목적 분기로 프로그램 경로를 유도하기 위해서 유도 분기를 선정, 유도 분기의 경로 제약 조건을 부정연산 하게 된다. 이때 유도 분기의 짝(paired) 분기가 존재한다고 가정하면 본 자료구조는 짝(paired) 분기의 branch_id를 추출하는데 사용된다. 표 4.5는 표 4.1을 대상으로 생성된 paired_branch벡터이다.

표 4.5 paired_branch벡터

paired_ branch	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
				6	5				

마. branches_

본 자료구조는 프로그램이 실행되었을 때 해당 테스트에 대한 프로그램의 실행 경로를 가지고 있다. 그림 2.4와 같은 CFG를 가지는 프로그램을 테스트한다고 가정하면 목표 분기가 4일 경우, branches_벡터에 저장된 내용은 표 4.6과 같다. 이때 벡터에 저장되는 branch_id는 짝(paired) 분기를 가지는 분기에 대해서만 저장되고 단일 분기인 경우는 저장되지 않는다.

표 4.6 branches_벡터

branches_	[0]	[1]	[2]	[3]
	-1	3	4	-2

바. constraints_

생성된 테스트 데이터를 시용해 테스트를 수행하며 생성된 경로 제약 조건식을 저장하는 자료구조로 알고리즘을 수행할 때, 예를 들어 어떤 분기의 경로 제약 조건을 부정 연산할 때 이용된다. 그림 2.4와 같은 CFG를 가지는 프로그램을 대상으로 생성된 constraints_벡터는 아래와 같다.

표 4.7 constraints_벡터

constraints_	[3]	[5]
	분기 3의 경로 제약 조건식	분기 5의 경로 제약 조건식

2.3 변수 및 함수

아래에서는 GCT 알고리즘에 사용되는 주요 변수 및 함수에 대해 기술한다.

가. 변수

GCT 알고리즘에서 사용되는 변수들은 아래와 같다.

표 4.8 GCT에 사용된 변수

자료형	변수명	설 명
vector<value_t>	input	생성된 테스트 데이터를 저장
string	var	현재 노드에서 사용된 변수를 저장
string	cause	내부 변수가 문제가 발생했을 때 해당 변수 저장
branch_id_t	p_branch	현재 노드의 짝(paired) 분기를 저장
branch_id_t	branch_idx	현재 노드의 branch_idx를 저장
branch_id_t	cur_node	현재 노드의 branch_id를 저장
vector<string>	cause_vector	하나 이상의 내부 변수가 존재할 때 변수들을 저장
SymbolicPath&	path	최근에 수행된 테스트의 프로그램 경로를 저장

나. 함수

아래에서는 GCT 알고리즘에서 구현된 함수들에 대해서 기술한다.

표 4.9 GCT에 사용된 함수

함수명	설 명
CheckNode	노드가 프로그램 경로에 존재하는지 체크
CheckConstraint	노드의 경로 제약 조건이 존재하는지 체크
isBranch	노드의 짝(paired) 분기가 존재하는지 체크
getPairedBranch	노드의 짝(paired) 분기의 branch_id를 가져옴
readCfg	cfg파일을 입력으로 받아 필요한 정보를 자료구조에 저장
getParentVars	부모 노드에 사용된 변수를 가져옴
getVars	노드의 변수를 가져옴
getBranch_idx	노드의 Branch_idx를 가져옴
getConstraint_idx	노드의 경로 제약 조건의 index를 가져옴
getParent	부모 노드의 branch_id를 가져옴
SolveAtBranch_Modify	경로 제약 조건식에 부정(negation)연산을 수행
GCT	알고리즘에 따라 실제 목적 지향 콘콜릭 테스트 수행

제 5 장 실험 결과

이 장에서는 여러 가지 예제 프로그램을 대상으로 CREST에서 기본적으로 제공되는 strategy의 동작 원리를 설명하고 본 논문에서 제안한 목적지향 콘콜릭 테스트 생성 방법의 효용성을 보이기 위해 기존에 제공되는 알고리즘과 비교하여 실험을 수행한 결과를 기술한다. 2절에서 확인할 수 있는 것처럼 입력으로 사용되는 프로그램의 경로 제약 조건식의 유형에 따라 테스트의 수행여부가 결정되므로 비교 대상으로 DFS 알고리즘을 모델로 삼는다.

제 1 절 실험환경

본 논문의 실험환경은 개발 초기에는 Cygwin이 설치된 윈도우 환경에서 실험을 수행하였으나 표 3.1과 같이 배열을 사용하는 예제를 실험할 때 배열이 일정크기 이상 증가하게 될 경우 스택 오버플로우 문제가 발생하게 되어 이를 해결하고자 Linux Ubuntu 9.04를 탑재한 Pentium 4(4GB의 메인 메모리, 2.5GHz) PC에서 수행하였다.

제 2 절 CREST에서 제공되는 탐색 알고리즘

CREST에서 기본적으로 제공되는 탐색 알고리즘은 모두 7가지이며 아래와 같다.

2.1 random strategy

랜덤으로 입력 데이터를 생성하여 테스트를 수행한 후에 새로운 분기를 만날 때마다 해당 분기를 테스트할 수 있는 테스트 데이터를 생성하고 테스트를 수행한다. 이 방법은 내부에서 default로 지정한 횟수를 초과하면 이전까지의 테스트 결과를 리셋하고 처음부터 테스트를 다시 수행한다. 그러므로 입력으로

들어오는 프로그램 분기의 개수가 일정 크기를 초과하면 즉, 표 3.1과 같이 제약 조건에 배열이 사용되는 경우를 테스트하기에 부적합하다.

그림 5.1은 표 5.1의 예제프로그램을 테스트한 결과이다. 표 5.1에서 사용된 예제 프로그램의 배열의 크기는 10이고 목적 분기를 실행하기 위해서는 배열 a[]의 요소가 모두 10을 만족해야 한다.

표 5.1 배열을 사용하는 예제 프로 그램

```
void test(int a[]){

        int a[10];
        int flag = 1;

        for(int i = 0; i< 10; i++){
                if(a[i] != 10){
                        flag = 0;
                }
        }

        if(flag){
                        fprintf(stderr,"GOAL!\n");
        }

}
```

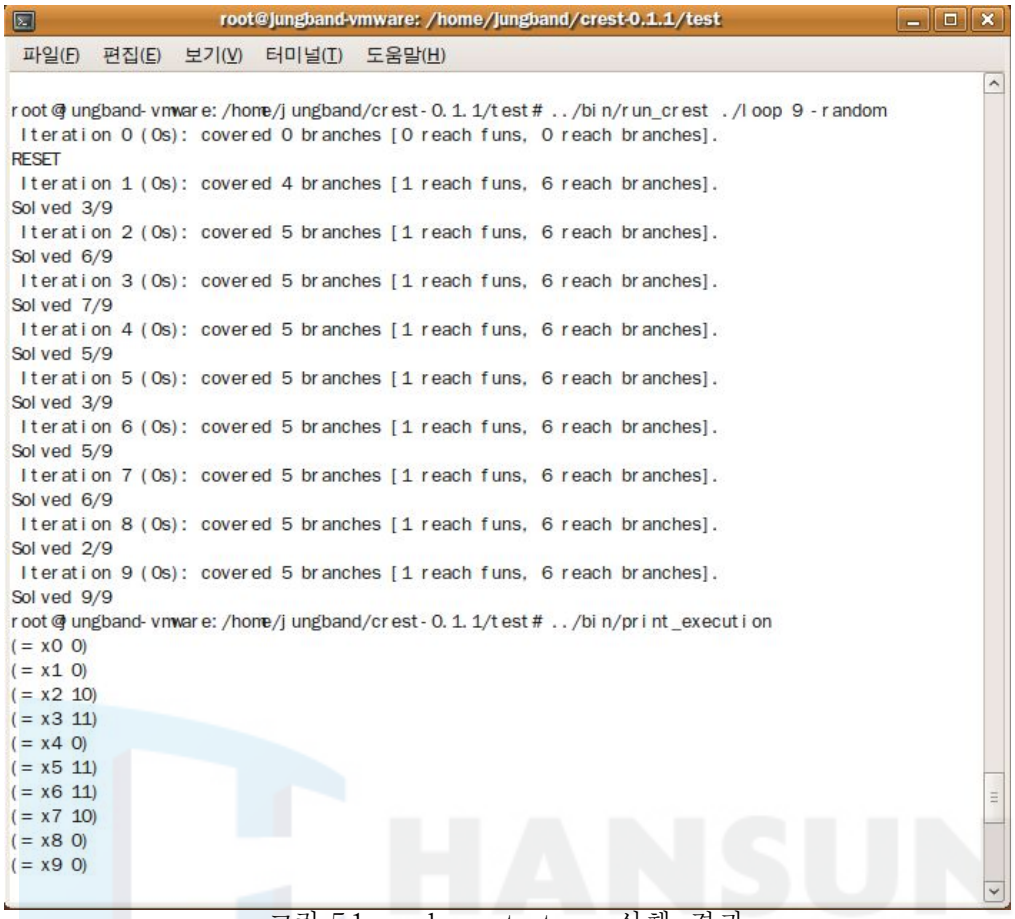

그림 5.1 random strategy 실행 결과

위 그림을 살펴보면 테스트를 모두 9번을 수행하였다. 테스트 결과를 살펴보면 제약 조건을 만족하기 위해서 반복문 내부에 존재하는 분기를 실행할 때 동적으로 생성되는 프로그램 경로를 따르기 때문에 배열 a[]의 같은 요소, 예를 들어 a[4]번째 배열의 요소를 처음에 테스트를 수행할 때 10으로 설정하고 다음번 테스트를 수행하면 a[4]를 다른 분기로 인식해서 다시 다른 값을 설정하는 과정을 반복하므로 테스트 횟수를 증가시켜도 결국 목적 분기를 실행할 수 없게 된다.

2.2 random_input strategy

랜덤으로 입력 데이터를 생성하여 테스트를 수행한 후 랜덤으로 입력 데이터를 선정하므로 표 5.1과 같이 제약 조건에 배열을 사용하는 경우 그 크기가 클수록 테스트 데이터를 생성하는데 많은 비용이 든다. 그림 5.2는 표 5.1의 예제 프로그램을 대상으로 테스트 실행 횟수를 10000번 반복한 결과이다. 테스트 결과를 살펴보면 테스트 데이터가 무작위로 생성되기 때문에 테스트를 보장할 수가 없다.

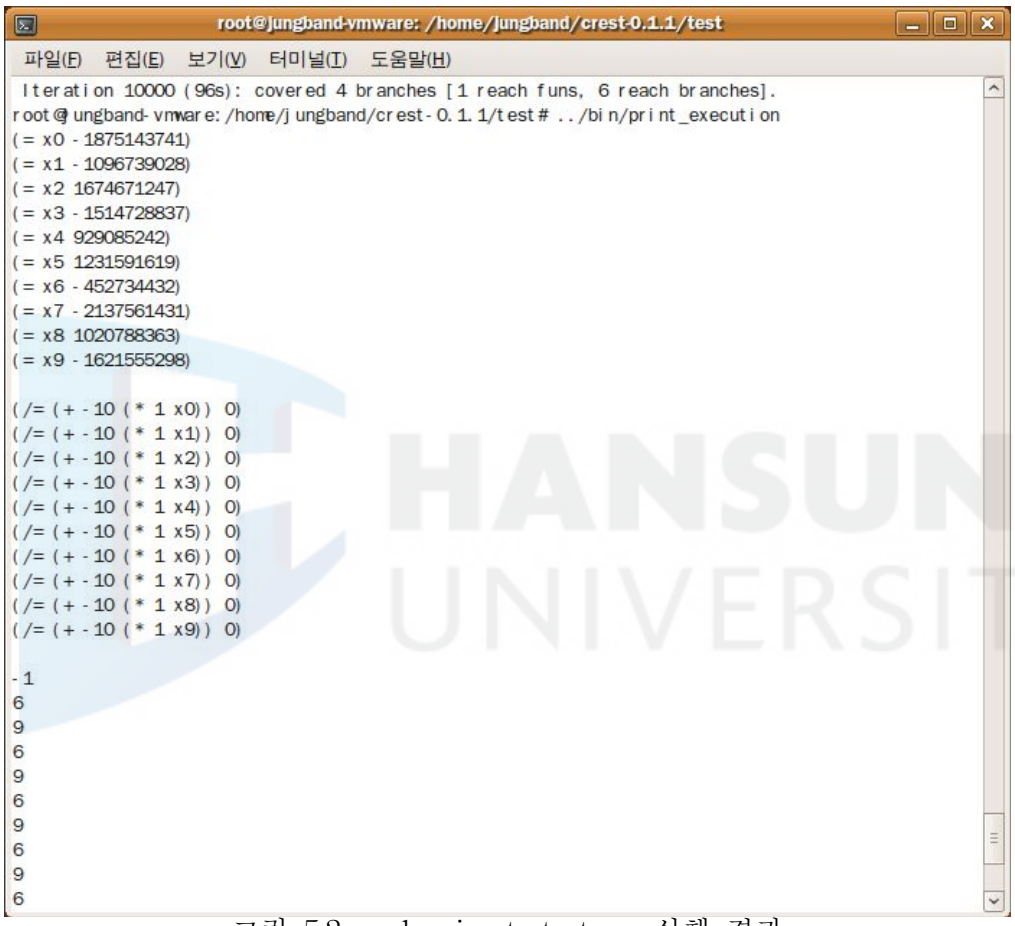

그림 5.2 random_input strategy 실행 결과

2.3 cfg strategy

테스트를 수행하기 전 입력 프로그램에 대한 정보를 추출하는 과정에서 생성된 cfg_branches 파일로부터 입력 프로그램의 제어 흐름 그래프(CFG) 정보를 가져와 테스트를 수행한다. 이 방법은 각 분기들의 조합으로 테스트를 수행하며 default로 지정한 횟수를 초과하면 이전까지의 테스트 결과를 리셋하고 처음부터 테스트를 다시 수행한다. 그러므로 분기의 개수가 일정 크기 이상이 되면 테스트를 보장할 수가 없다. 그림 5.3은 표 5.1의 예제 프로그램을 대상으로 테스트 실행 횟수를 1000번 반복한 결과이다.

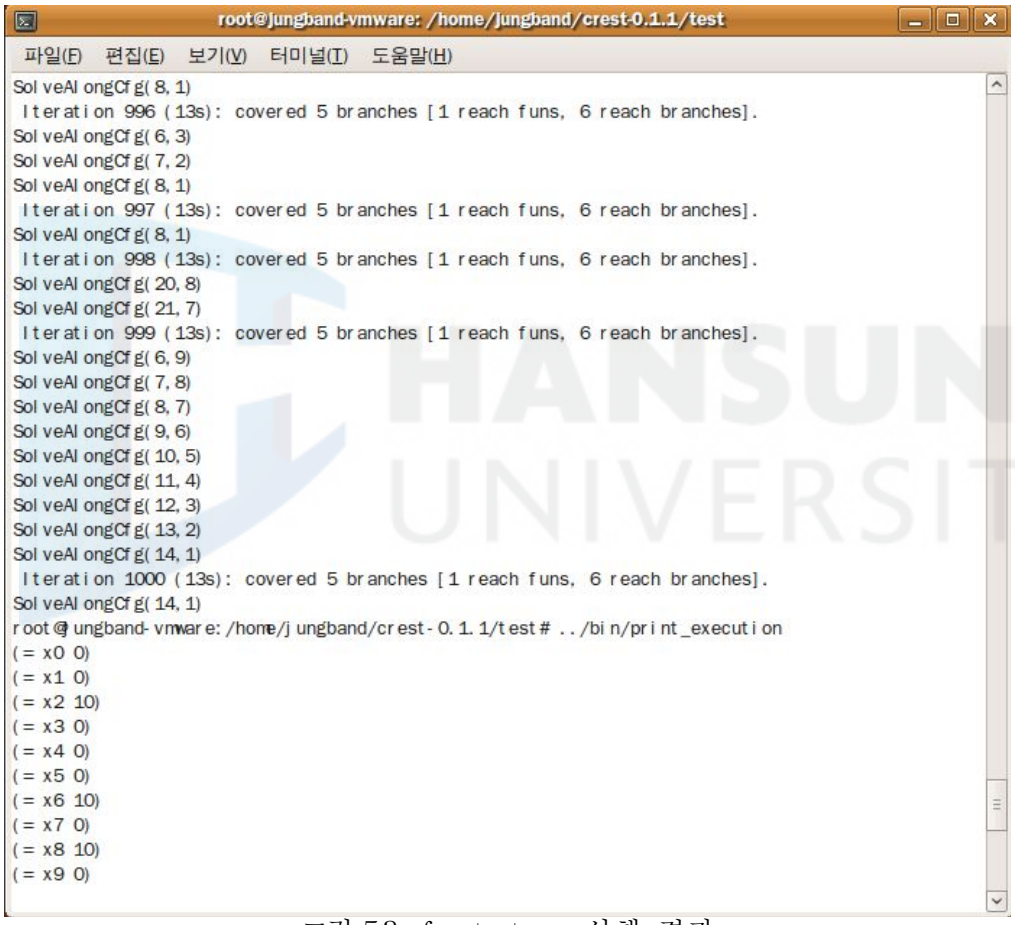

그림 5.3 cfg strategy 실행 결과

2.4 cfg_baseline strategy

cfg strategy를 기반으로 하고 테스트를 수행하는 방법으로 각 분기에 score값을 설정하며 default로 지정한 score 값을 초과하면 이전까지의 테스트 결과를 리셋하고 처음부터 테스트를 다시 수행한다. 그러므로 분기의 개수가 일정 크기 이상이 되면 테스트를 보장할 수가 없다. 그림 5.4은 표 5.1의 예제 프로그램을 대상으로 테스트 실행 횟수를 1000번 반복한 결과이다.

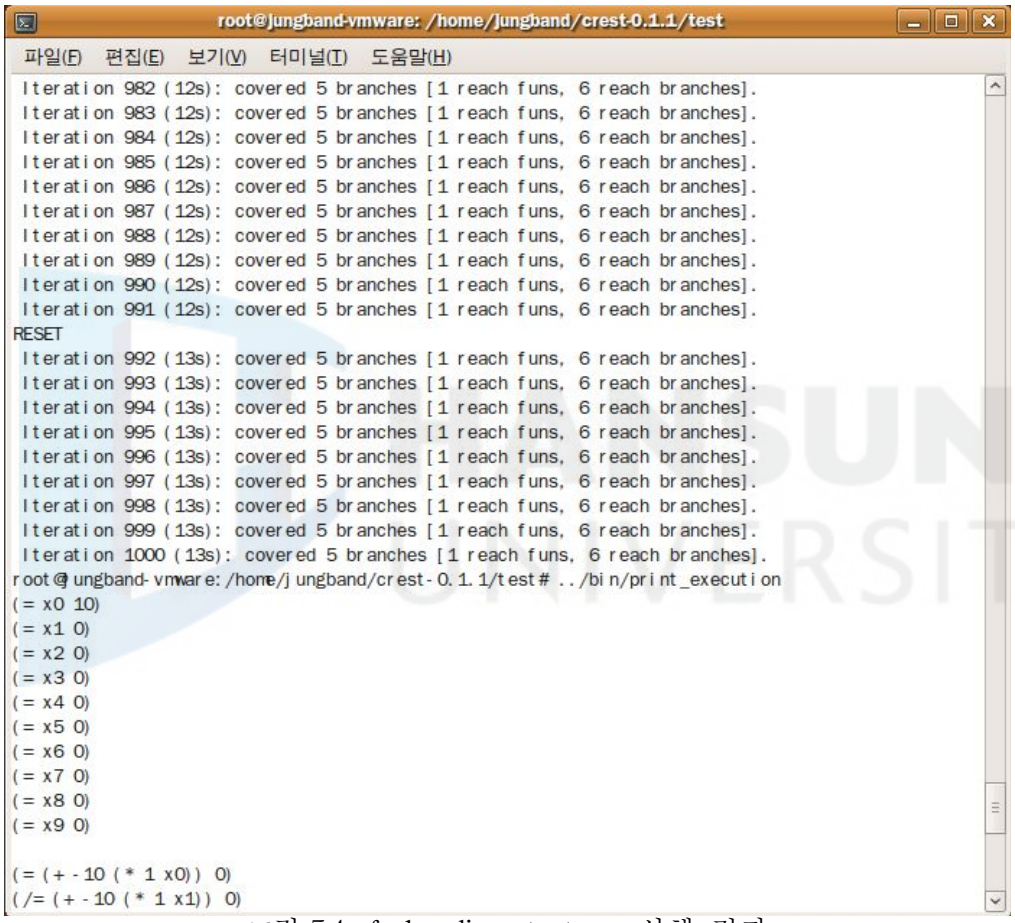

그림 5.4 cfg_baseline strategy 실행 결과

2.5 hybrid strategy

uniform_random strategy를 기반으로 하고 테스트를 수행하는 방법으로 반복문 내부에 존재하는 분기를 실행할 때 random strategy와 같이 배열의 같은 요소 값을 계속 재설정하게 되어 목적 분기를 실행할 수 없게 된다. 그림 5.5는 표 5.1의 예제 프로그램을 대상으로 테스트 실행 횟수를 1000번 반복한 결과이다.

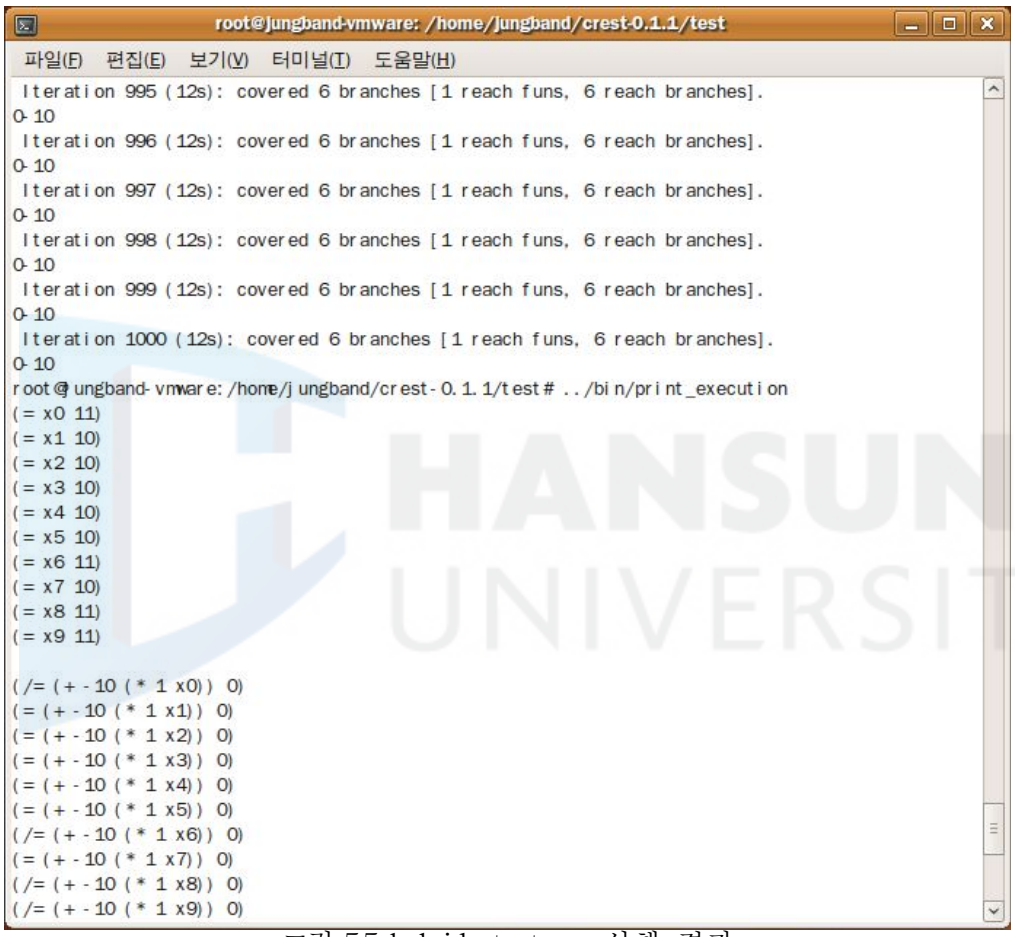

그림 5.5 hybrid strategy 실행 결과

2.6 uniform_random strategy

최초 생성된 테스트 데이터로 생성된 경로 제약 조건식을 이용하여 테스트를 수행하는 방법으로 반복문 내부에 존재하는 분기를 실행할 때 random strategy와 같이 배열의 같은 요소 값을 계속 재설정하게 되어 목적 분기를 실행할 수 없게 된다. 그림 5.6은 표 5.1의 예제 프로그램을 대상으로 테스트 실행 횟수를 1000번 반복한 결과이다.

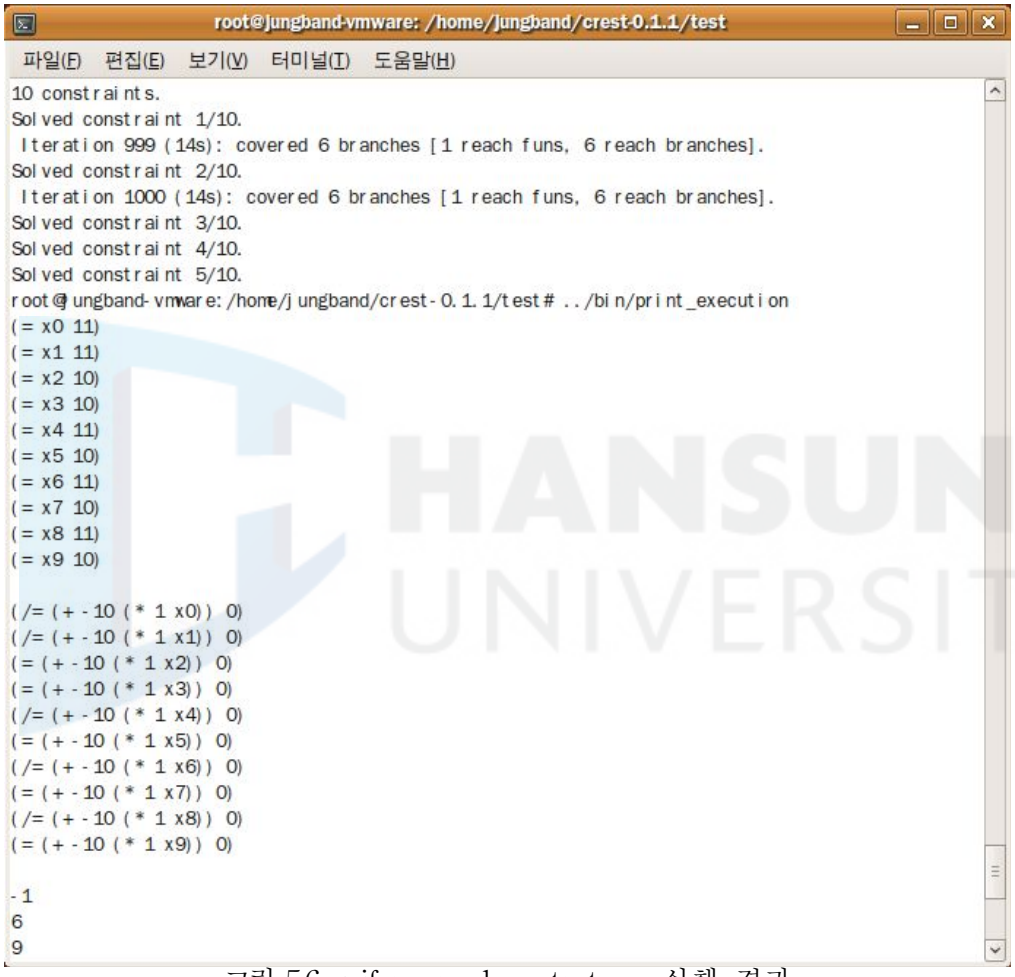

그림 5.6 uniform_random strategy 실행 결과

2.7 dfs strategy

랜덤으로 입력 데이터를 생성하여 테스트를 수행한 후에 생성된 경로 제약 조건식과 프로그램의 depth를 감소시키면서 테스트를 수행하는 방법으로 표 5.1과 같이 경로 제약 조건식에 하나의 배열을 사용하는 예제뿐만 아니라 표 3.1과 같이 하나 이상의 배열을 사용하는 예제도 모두 테스트를 수행할 수 있다. 그림 5.7은 표 5.1의 예제 프로그램을 대상으로 테스트를 수행한 결과이다.

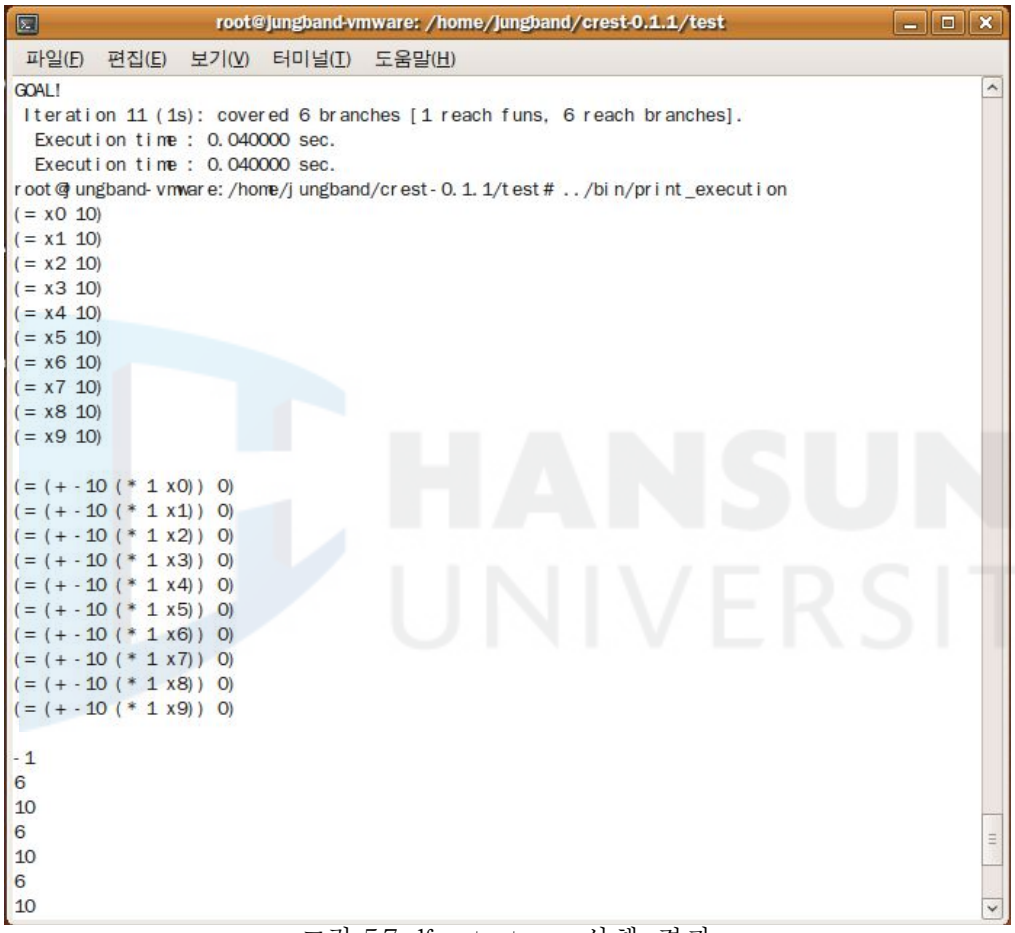

그림 5.7 dfs strategy 실행 결과

제 3 절 GCT 알고리즘의 성능 실험

본 절에서는 실험에 사용된 여러 가지의 예제 프로그램을 몇 가지 유형으로 분류 한다 그리고 이를 대상으로 GCT 알고리즘을 이용하여 테스트를 수행한 후 그 동작이 올바른지 여부를 실험하고 DFS 알고리즘과 비교해 GCT 알고리즘의 효용성을 증명한다.

3.1 예제 프로그램의 유형

본 논문에서 테스트에 사용되는 예제 프로그램은 모두 13가지이며 이 예제 프로그램들의 유형을 분류하면 아래와 같다.

내부 변수에 의해 프로그램의 실행 경로가 결정되는 프로그램
flag_assigment.c, flag_multiple.c, trityp.c 등

배열을 입력으로 받아 조건에 따라 실행 경로가 결정되는 프로그램
loop.c 등

구조체 변수 및 구조체 배열 의해 프로그램의 실행 경로가 결정되는 프로그램
handle_new_job.c, netflow.c 등

3.2 예제 프로그램을 이용한 실험

가. flag_multiple.c

두 개의 내부변수 initialized와 has_been_fired에 의해 프로그램의 실행 경로가 결정되는 프로그램이다. 프로그램 코드는 표 5.2와 같고 제어흐름 그래프는 그림 5.8과 같다. 본 예제에서는 하나 이상의 내부변수가 존재했을 때 GCT 알고리즘이 제대로 동작하는지를 실험한다.

표 5.2 flag_multiple.c

```
void test(int a, int b) {

        int a, b;
        int initialized = 0;
        int has_been_fired = 0;

         if(a != 0)
                initialized = 1;

         if(b != 0)
                has_been_fired = 1;

         if(initialized){
                if(b == 1)
                        has_been_fired = 1;

                if(has_been_fired)
                        fprintf(stderr, "GOAL!\n");
         }
}
```

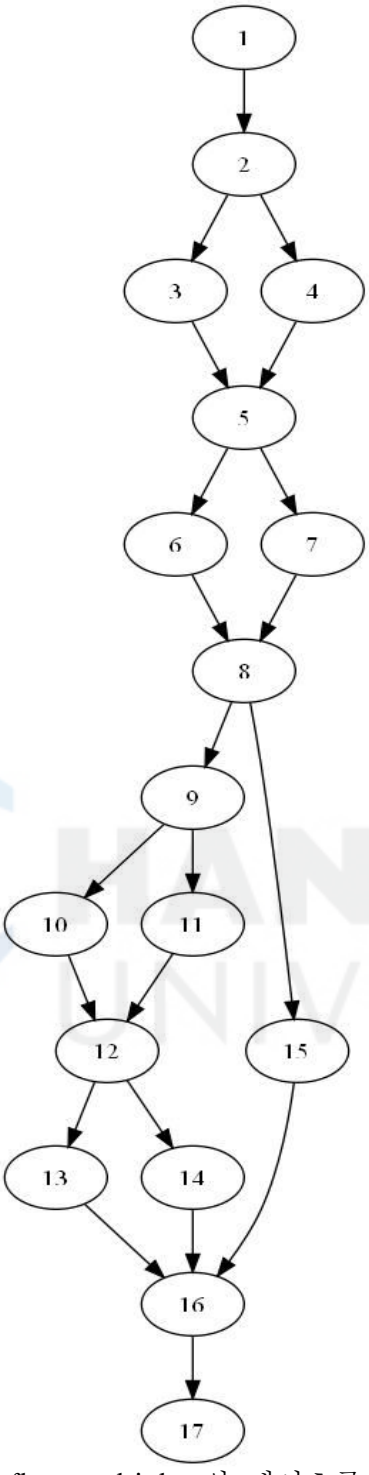

그림 5.8 flag_multiple.c의 제어흐름 그래프

그림 5.8에서 목적 블록은 13번 노드가 되고 이를 찾기 위한 테스트 시나리오는 아래와 같다.

1) 프로그램을 실행하여 테스트 데이터를 생성한다. 이때 a = 0, b = 0이 되고 실행 경로는 1→2→4→5→7→8→15→16→17이 된다.

2) 다음 실행에서 목적 분기를 찾기 위해서 유도분기를 선정(9)하고 유도분기의 제약 조건식(8)을 부정하여 테스트를 다시 수행한다.

2-1) 하지만 유도분기의 제약 조건이 존재하지 않아 이전과 프로그램 경로는 같게 된다.

2-2) 프로그램 경로에 영향을 미치는 요소, 즉 내부변수 initialized을 확인 하고 이 값에 영향을 미치는 노드를 찾기 위해 경로를 역 추적한다.

2-3) initialized값에 영향을 미치는 노드(4)를 찾아 해당 노드의 경로 제약 조건을 부정하여 테스트를 수행하면 a = 1, b = 0이 되고 프로그램 경로는 1→2→3→5→7→8→9→11→12→14→16→17이 된다.

3) 또 다른 내부변수 has_been_fired를 찾기 위해서 2)번 과정을 반복하여 결국 목적 분기를 실행하고 테스트를 종료한다.

아래 그림 5.9은 테스트의 수행 결과를 보여주며 세 번의 실행으로 목표 분기를 찾는데 성공하여 테스트가 종료되었음을 확인할 수 있다.

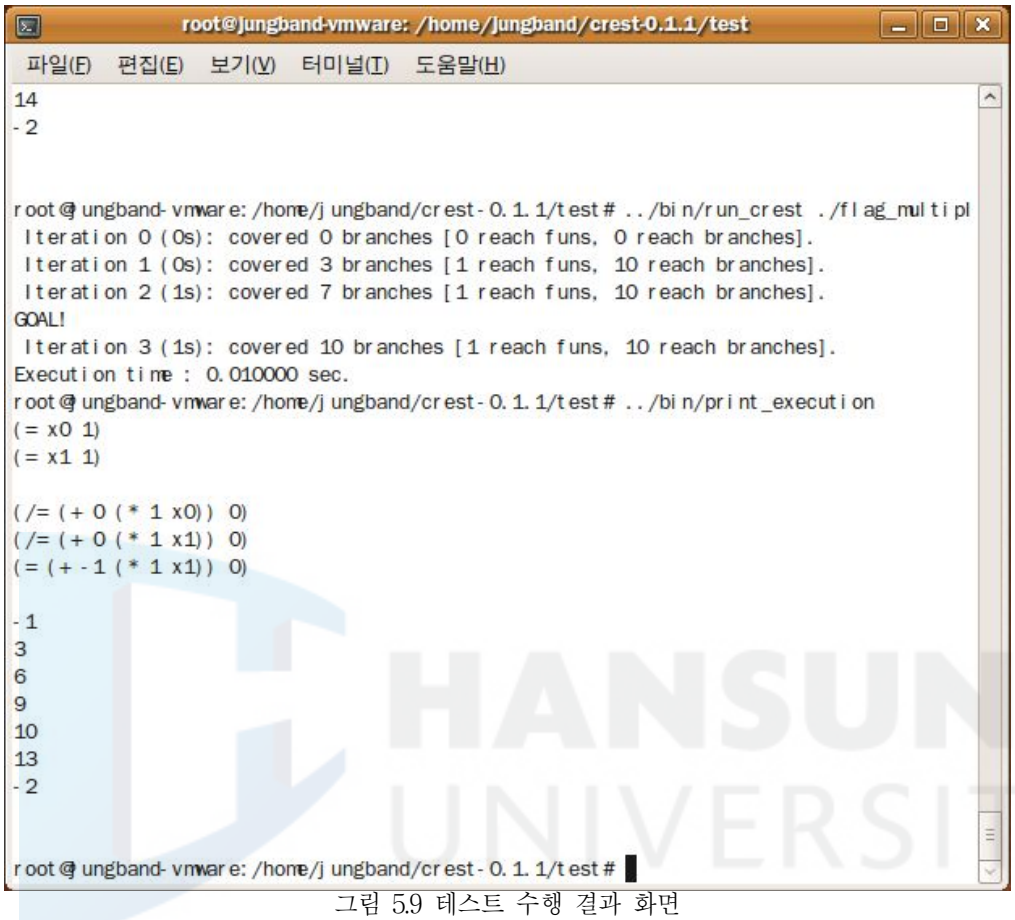

그림 5.9 테스트 수행 결과 화면

나. trityp.c

세변을 입력으로 받아 삼각형의 형태를 판단하는 프로그램이다. 본 예제에서는 입력으로 Side1, Side2, Side3를 받으며 내부변수 triOut가 사용되는 복잡한 프로그램을 대상으로 GCT 알고리즘이 제대로 동작하는지를 실험한다. 프로그램 코드는 표 5.3와 같고 제어 흐름 그래프는 그림 5.10과 같다.

표 5.3 trityp.c

```
int test(int Side1, int Side2, int Side3){

        int triOut = 0;
        int Side1, Side2, Side3;

         if(Side1 <= 0 || Side2<=0 || Side3<= 0){
                  fprintf(stderr,"Not Triangle1!\n"); //삼각형이 아님
                  return 0;
         }

         if(Side1 == Side2)
              triOut = triOut + 1;

         if(Side1 == Side3)
              triOut = triOut + 2;

         if(Side2 == Side3)
              triOut = triOut + 3;

         if(triOut == 0){
                   if(Side1+Side2<=Side3||Side2+Side3<=Side1||Side1+Side3<=Side2){
                            fprintf(stderr,"Not Triangle!\n"); //삼각형이 아님
                   }else{
                            fprintf(stderr,"Triangle!\n"); //삼각형
                   }
                   return 0;
```

```
        }

        if(triOut > 3){
            fprintf(stderr,"Equilateral Triangle!\n"); //정삼각형
        }else if(triOut==1 && Side1+Side2>Side3){
            fprintf(stderr,"isosceles Triangle1!\n"); //이등변삼각형
        }else if(triOut==2 && Side1+Side3>Side2){
            fprintf(stderr,"isosceles Triangle!\n"); //이등변삼각형
        }else if(triOut==3 && Side2+Side3>Side1){
            fprintf(stderr,"isosceles Triangle!\n"); //이등변삼각형
        }else{
                fprintf(stderr,"Not Triangle!\n"); //삼각형이 아님
        }

    return 0;
}
```

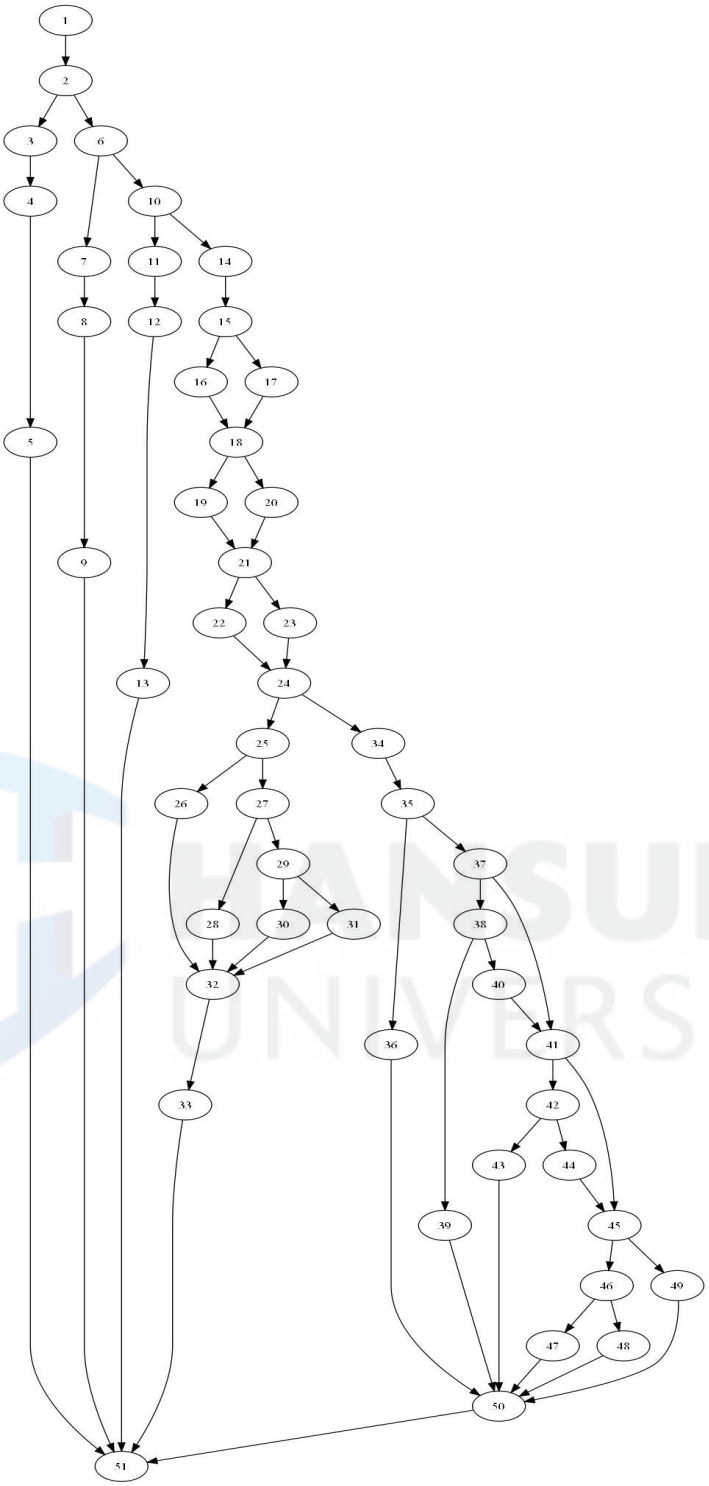

그림 5.10 trityp.c의 제어 흐름 그래프

그림 5.10에서 목적 블록은 삼각형이 아님을 나타내는 블록(3, 7, 11, 26, 28, 30), 이등변 삼각형을 나타내는 블록(31), 정삼각형을 나타내는 블록(36), 삼각형임을 나타내는 블록(31) 등 여러 개가 존재할 수 있다. 실험에서는 목표 분기를 각 유형마다 하나씩만 선정해 실험을 수행하였다.

삼각형이 아님을 나타내는 블록으로 11번을 선정하였으며 실험 결과는 그림 5.11와 같다.

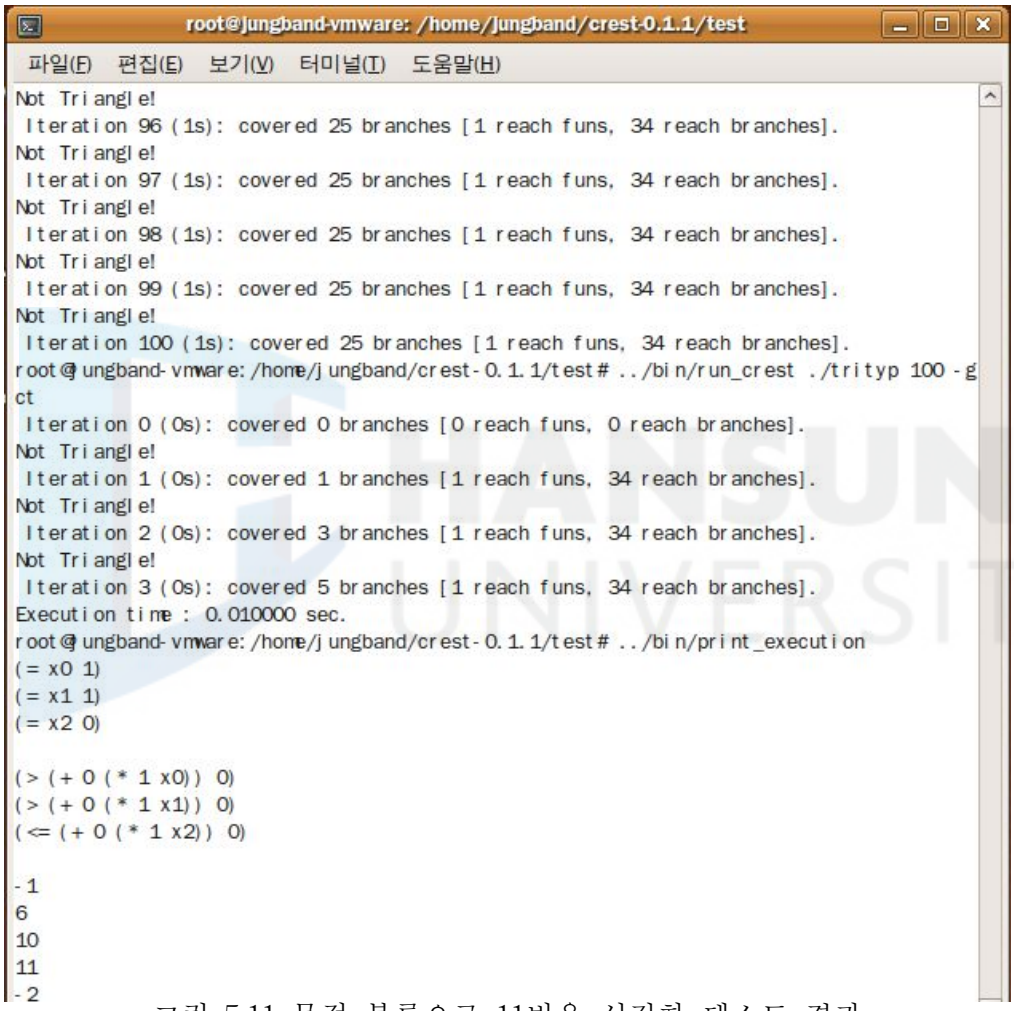

그림 5.11 목적 블록으로 11번을 선정한 테스트 결과

위 그림을 살펴보면 여덟 번의 테스트를 수행한 후 목표 분기를 실행하게 되어 테스트가 종료됨을 확인할 수 있다.

아래 결과는 이등변 삼각형을 나타내는 블록으로 43번 선정하였으며 실험 결과는 그림 5.12과 같다.

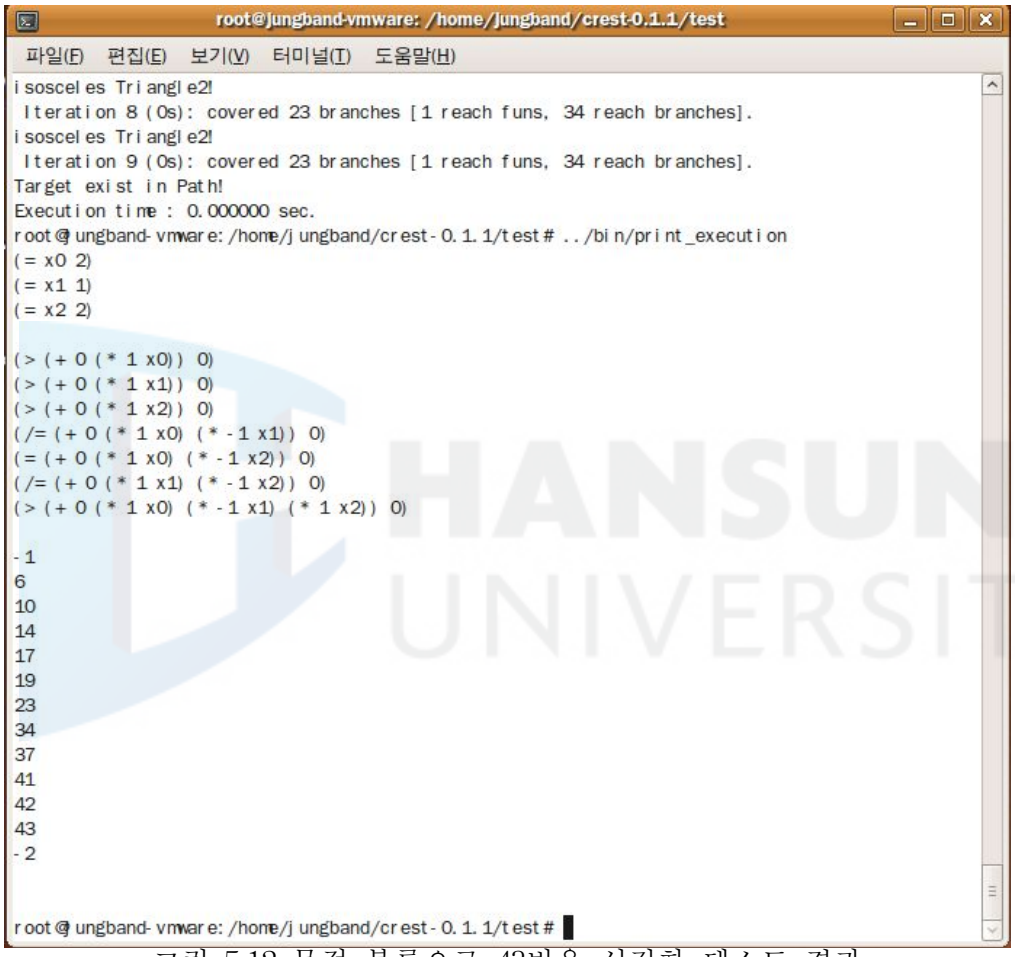

그림 5.12 목적 블록으로 43번을 선정한 테스트 결과

그림을 살펴보면 아홉 번의 테스트를 수행한 후 목표 분기를 실행하게 되어 테스트가 종료됨을 확인할 수 있다.

아래 결과는 정삼각형을 나타내는 블록으로 36번 선정하였으며 실험 결과는 그림 5.13과 같다.

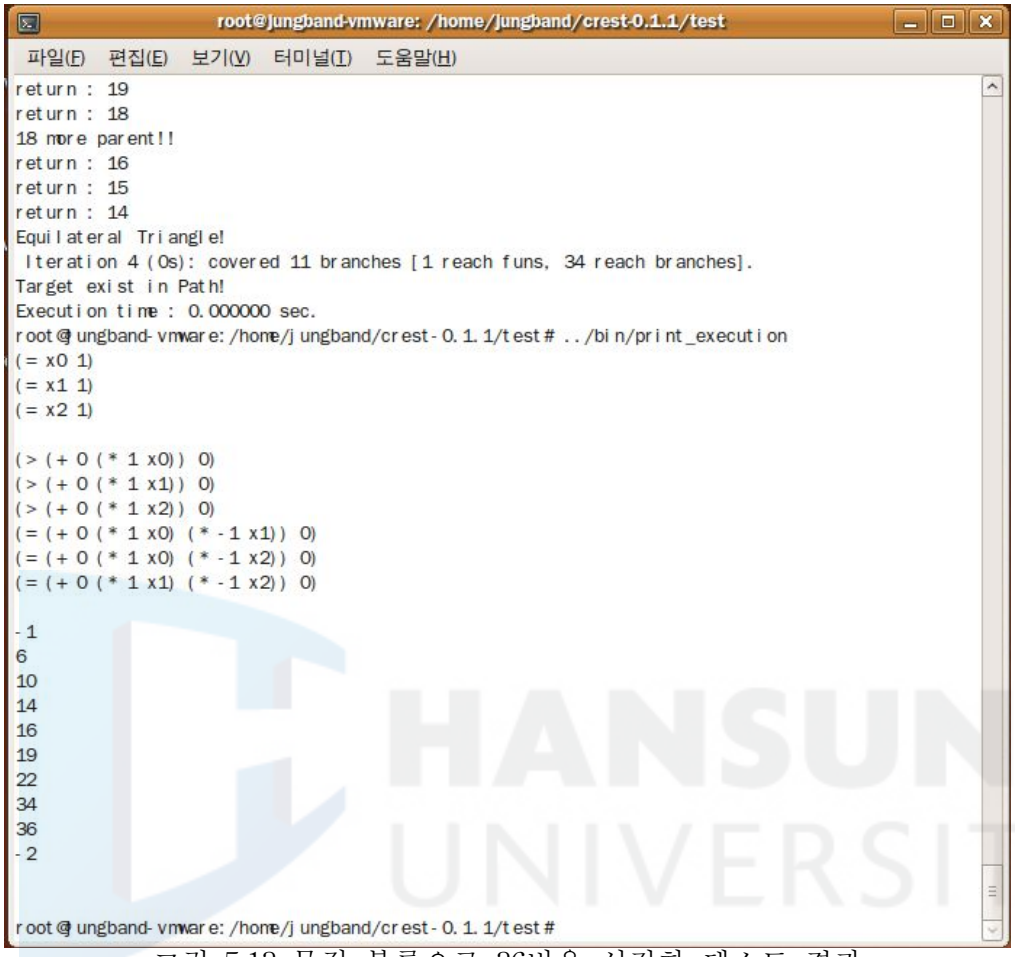

그림 5.13 목적 블록으로 36번을 선정한 테스트 결과

그림을 살펴보면 네 번의 테스트를 수행한 후 목표 분기를 실행하게 되어 테스트가 종료됨을 확인할 수 있다.

아래 결과는 삼각형을 나타내는 블록으로 31번 선정하였으며 실험 결과는 그림 5.14과 같다.

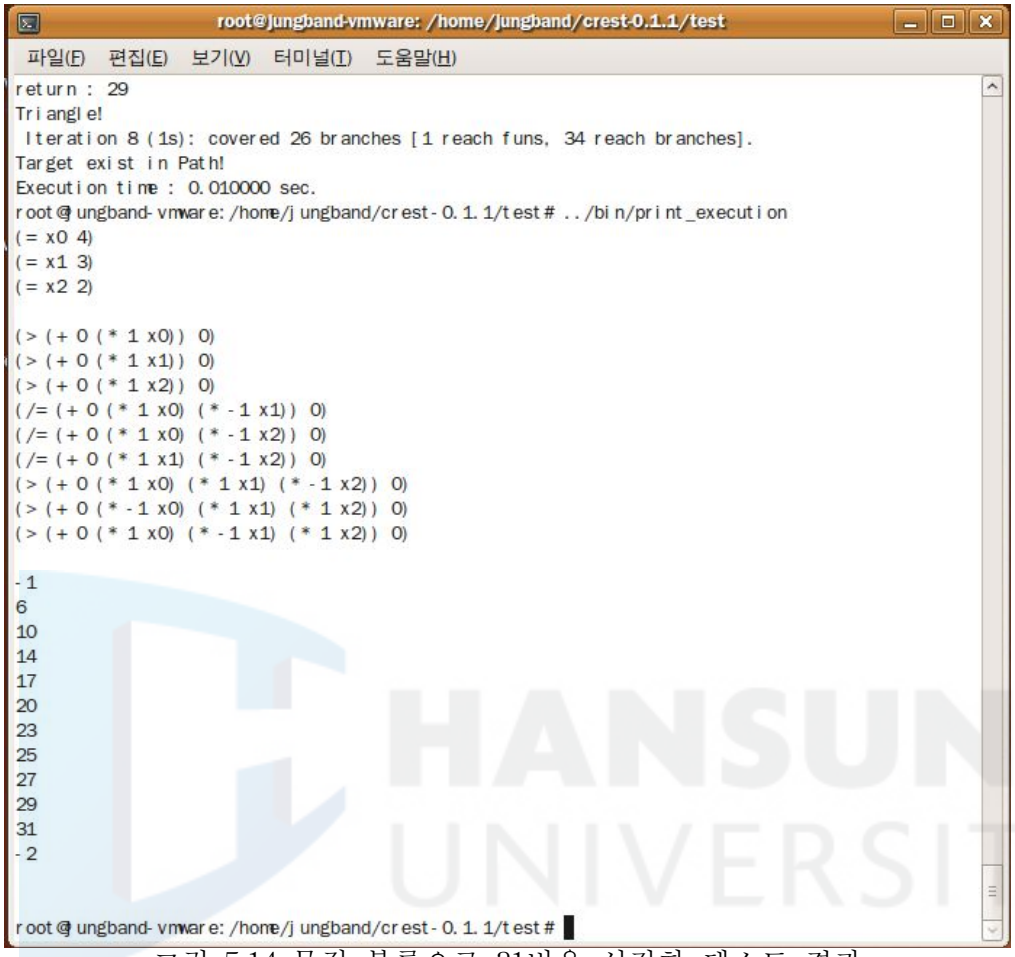

그림 5.14 목적 블록으로 31번을 선정한 테스트 결과

그림을 살펴보면 여덟 번의 테스트를 수행한 후 목표 분기를 실행하게 되어 테스트가 종료됨을 확인할 수 있다.

다. loop.c

두 배열을 입력으로 받아 특정 조건을 만족했을 때 목표 분기를 실행하는 프로그램이다. 본 예제에서는 DFS 알고리즘에 비해 GCT 알고리즘이 효율적임을 보이기 위해 배열의 크기와 조건을 변경하며 실험을 수행하였다. 예제 프로그램의 코드는 표 5.4와 같고 제어 흐름 그래프는 그림 5.15과 같다.

표 5.4 loop.c

```
int test(int a[], int b[]){

        int i;
        int a[10],b[10];
        int fa, fb;

        i = 0;
        fa = 0;
        fb = 0;

        while(i < 10){
                if(a[i] == 10)
                fa = 1;
                i = i + 1;
        }

        if(fa == 1){
                fb = 1;
                i = 0;
                while(i < 10){
                        if(b[i] != 10)
                                fb = 0;
                        i = i + 1;
```

```
                }
        }

        if(fb == 1)
                fprintf(stderr,"GOAL!!\n");
}
```

위 예제에서 배열의 크기는 모두 10으로 설정하였고 마지막 조건문에서 fb가 1, 즉, 참이 되는 경우가 목표 분기이며 이를 실행하기 위해서는 입력 배열 a[]에 원소 중의 하나가 10을 가지고 배열 b[]의 모든 원소들이 10이 되는 경우이다. 변수 fa나 fb와 같이 참이나 거짓을 가지는 부울리언 변수로 프로그램의 분기 조건을 사용하는 프로그램에서 기존의 많은 테스트 데이터 탐색 알고리즘의 성능은 랜덤 테스트를 수행하는 경우와 동일한 것으로 밝혀졌다.

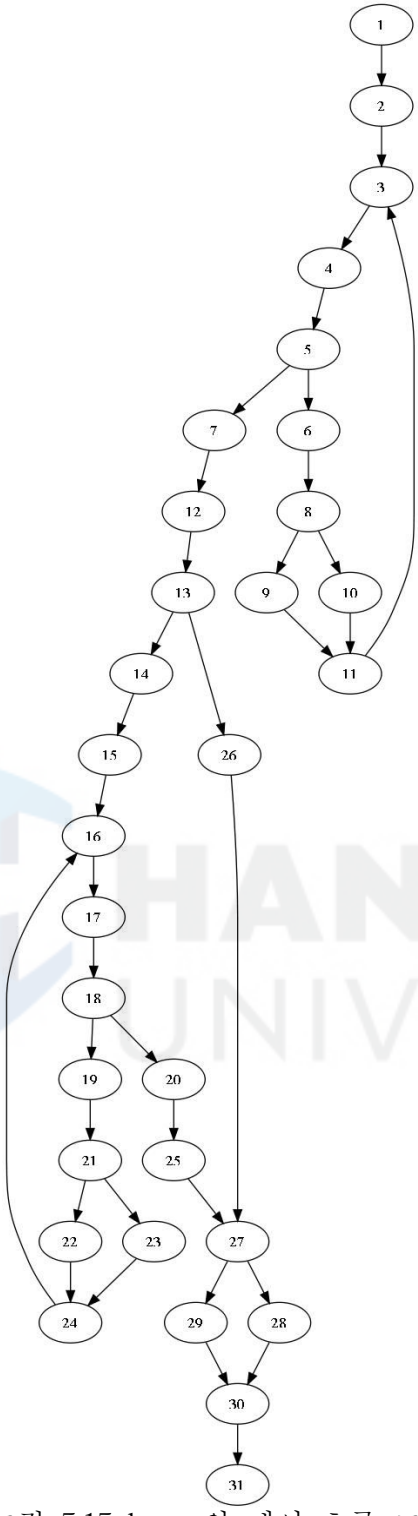

그림 5.15 loop.c의 제어 흐름 그래프

그림 5.16와 5.17은 각각 DFS 알고리즘과 GCT 알고리즘을 이용해 테스트를 수행한 결과이다.

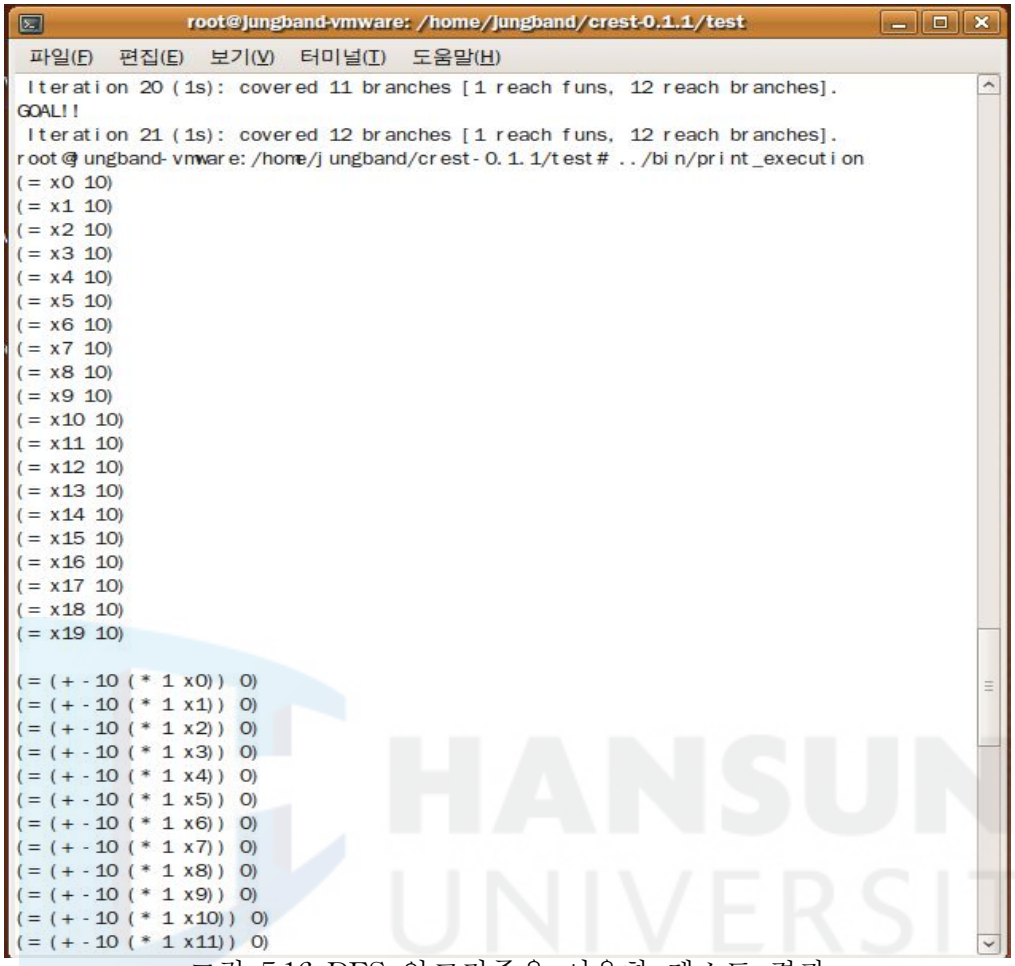

그림 5.16 DFS 알고리즘을 이용한 테스트 결과

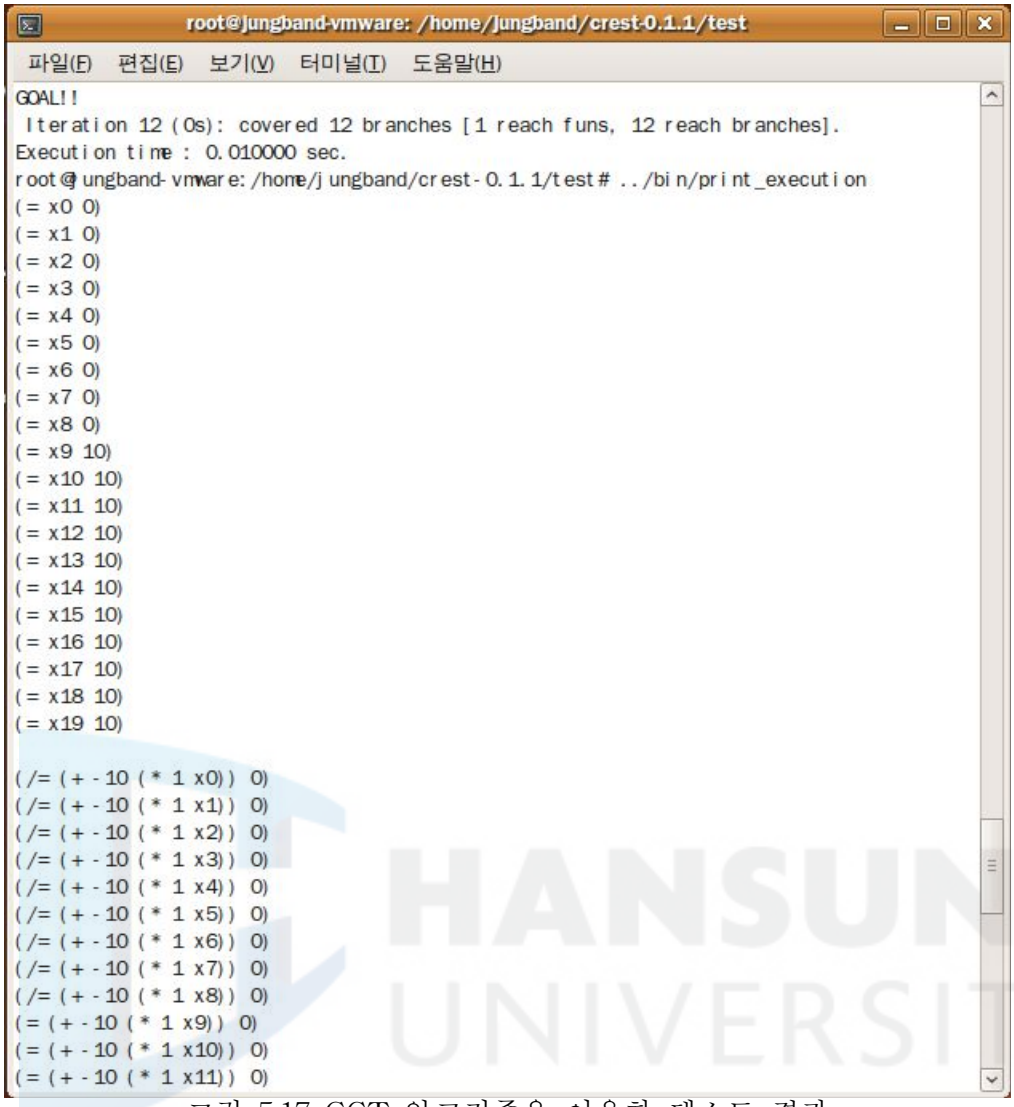

그림 5.17 GCT 알고리즘을 이용한 테스트 결과

위 그림을 비교해 보면 DFS 알고리즘은 스물 한 번의 테스트를 수행하고 나서 목표 분기를 찾아 테스트를 종료한 반면, GCT 알고리즘은 열 한 번의 테스트를 수행하고 목표 분기를 찾아 GCT 알고리즘이 효과적이라는 것을 확인할 수 있다. 이번에는 조건을 변경하여 실험을 수행하도록 한다. 변경된 조건은 표 5.5와 같으며 제어 흐름 그래프는 그림 5.15과 동일하다.

표 5.5 변경된 loop.c

```
int test(int a[], int b[]){

        int i;
        int a[10],b[10];
        int fa, fb;

        i = 0;
        fa = 0;
        fb = 0;

        while(i < 10){
                if(a[i] == 10)
                fa = 1;
                i = i + 1;
        }

        if(fa == 1){
                fb = 0;
                i = 0;
                while(i < 10){
                        if(b[i] == 10)
                                fb = 1;
                        i = i + 1;
                }
        }

        if(fb == 1)
                fprintf(stderr,"GOAL!!\n");
}
```

예제에서 배열의 크기는 모두 10으로 설정하였고 목표 분기를 실행하기 위해서는 입력 배열 a[]에 원소 중의 하나가 10을 가지고 배열 b[]의 원소 중의 하나가 10이 되는 경우이다. 그림 5.18와 5.19은 각각 DFS 알고리즘과 GCT 알고리즘을 이용해 테스트를 수행한 결과이다.

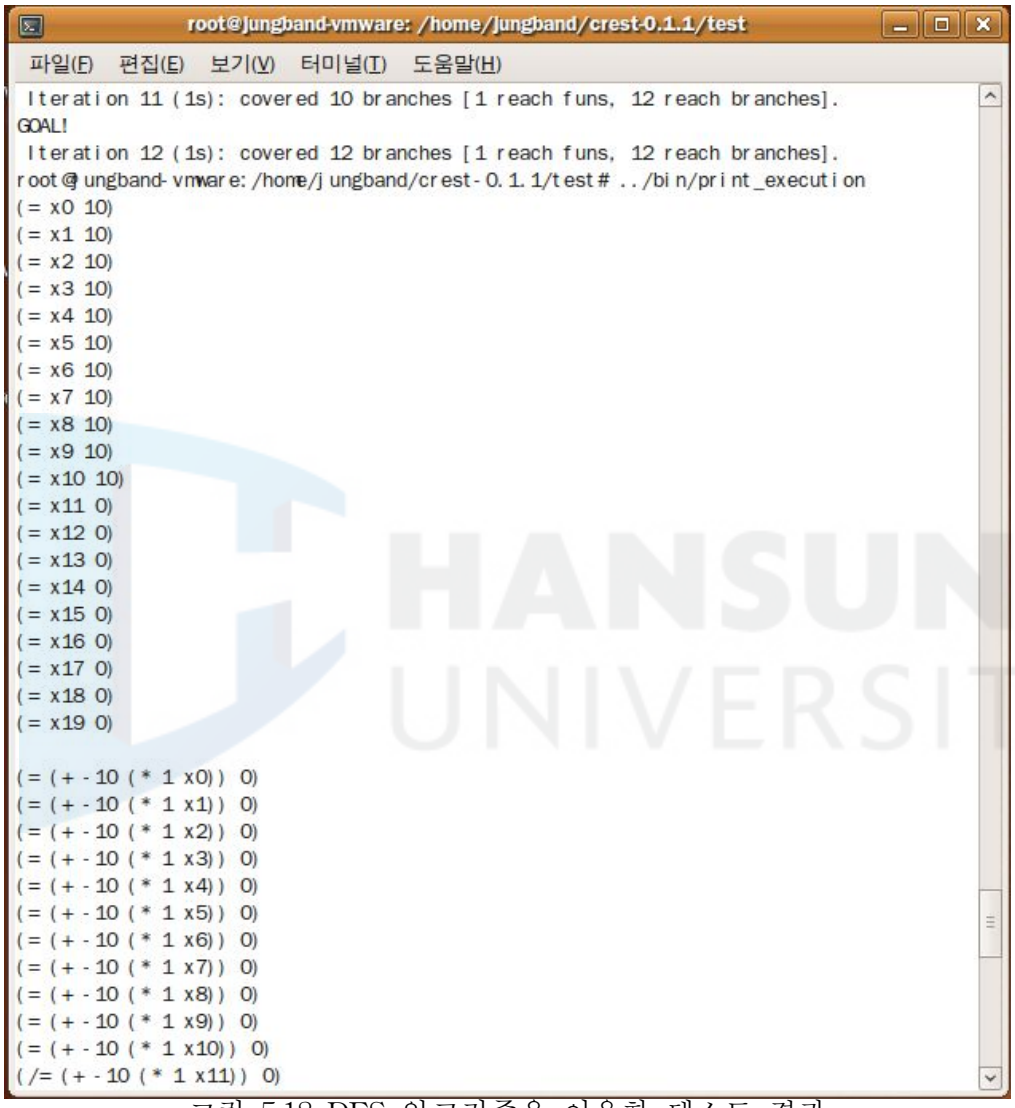

그림 5.18 DFS 알고리즘을 이용한 테스트 결과

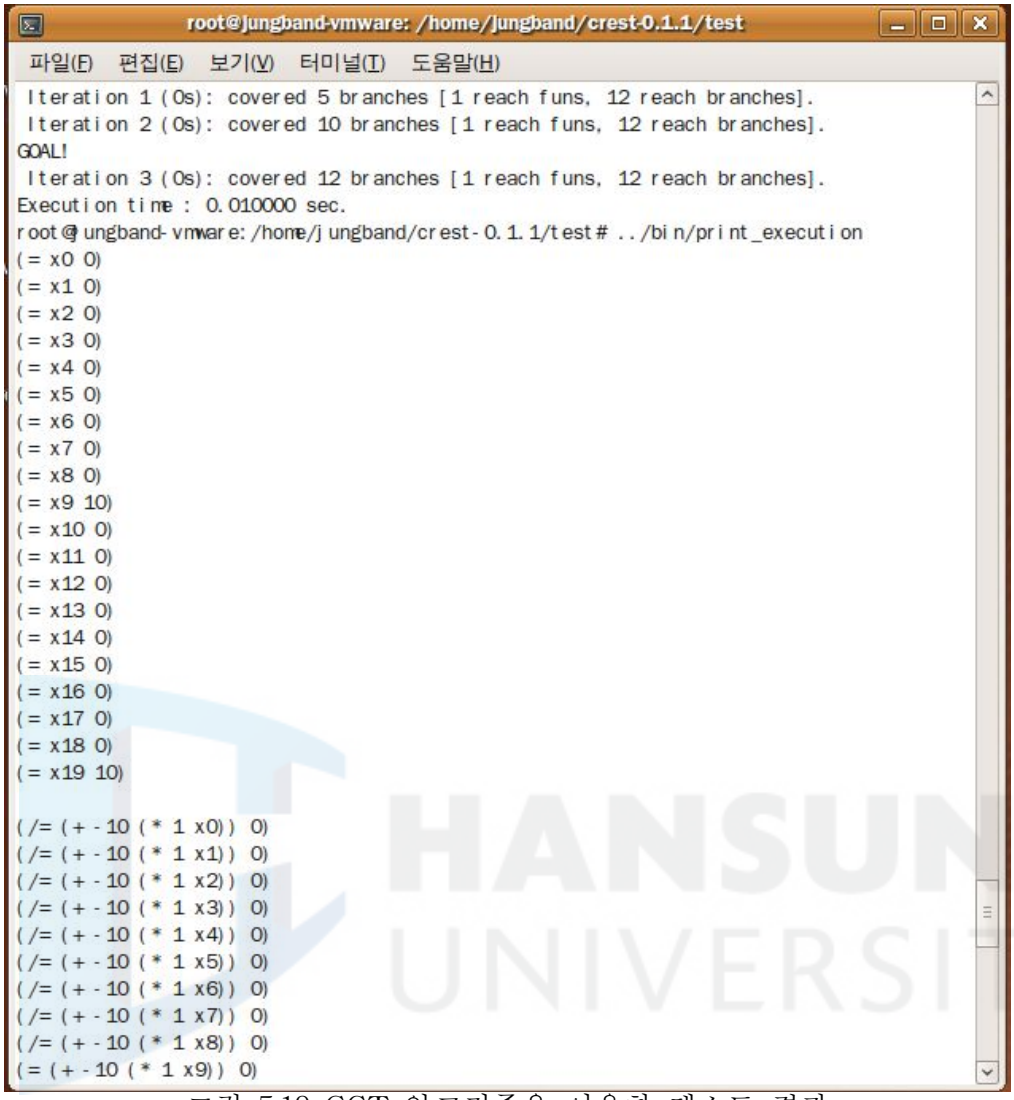

그림 5.19 GCT 알고리즘을 이용한 테스트 결과

위 그림을 비교해 보면 DFS 알고리즘은 열두 번의 테스트를 수행하고 나서 목표 분기를 찾아 테스트를 종료한 반면, GCT 알고리즘은 세 번의 테스트를 수행하고 목표 분기를 실행해 테스트가 종료된다. 그림 5.20과 5.21는 각각 표 5.4와 5.5에 대해 DFS 알고리즘과 GCT 알고리즘을 배열의 크기에 따라 목표 분기를 실행하는 테스트 데이터를 생성하는데 까지 걸리는 시간을 비교한 그래프이며 이때 배열의 크기는 30으로 한정하였다.

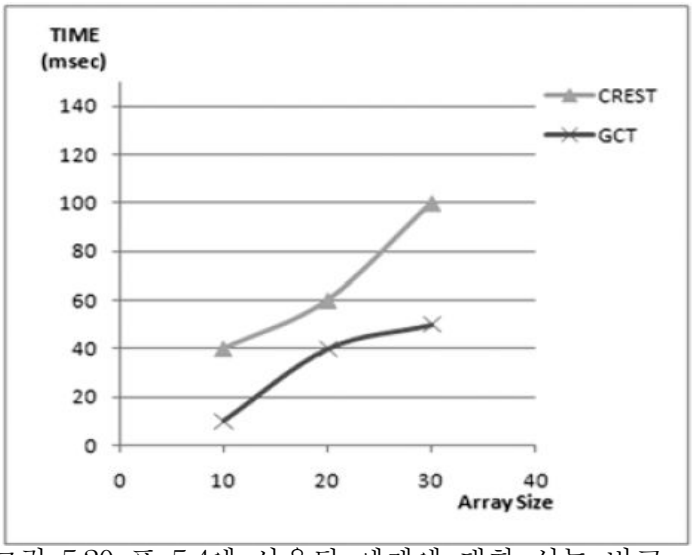

그림 5.20 표 5.4에 사용된 예제에 대한 성능 비교

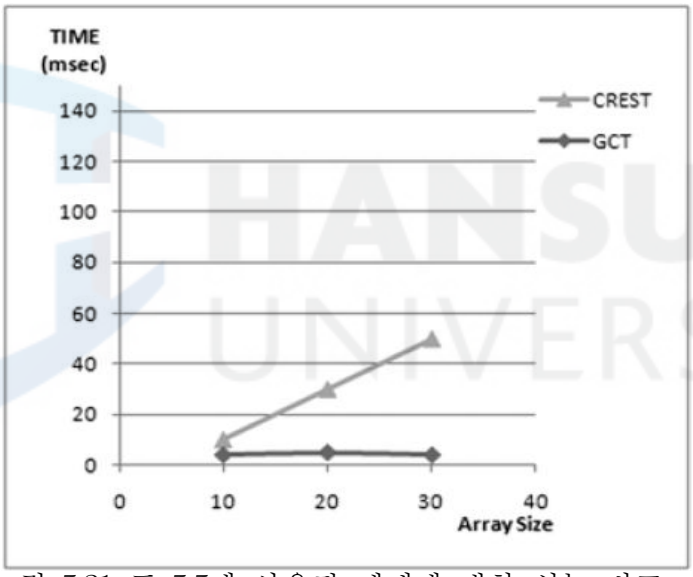

그림 5.21 표 5.5에 사용된 예제에 대한 성능 비교

그래프에서 볼 수 있듯이 GCT 알고리즘이 목표 분기를 실행하는 테스트 데이터를 찾는 시간에서 절반정도의 시간이 소요된 사실을 알 수 있다. 이러한 성능 차이는 목표 분기에 도달하기 전에 탐색해야 하는 경로가 많은 경우에 확연하게 드러난다.

이와 같은 결과는 DFS 알고리즘에서 목표 분기에 도달하기 전에 탐색해야 하는 경로들을 모두 한 후에 목표 분기에 도달한다. 즉 “조건 if (a[i]==10)...”에 의해 배열 크기만큼 실행 분기가 생성되며 (i.e. 'if(a[0]==10)...', 'if(a[1]==10)...',,,,,,'if[9]==10...') DFS 알고리즘은 이 분기들을 모두 탐색하기 때문이다. 반면 GCT 알고리즘은 생성된 실행 분기 중에서 하나만을 실행하는 테스트 데이터를 탐색하면 충분하다. 특히 이 경우에 GCT 알고리즘의 성능은 배열의 크기에 의존적이지 않음을 알 수 있다.

제 6 장 결론 및 향후 연구

본 논문에서는 콘콜릭 테스트를 기반으로 특정 분기를 실행할 수 있는 테스트 데이터 생성 방법인 목적 지향 콘콜릭 테스트에 대해 기술하였다.

제 1 절 결론

기본적으로 콘콜릭 테스트는 프로그램을 테스트할 때 실행 가능한 모든 경로를 심볼릭 수행기법을 사용하여 테스트함으로써 많은 비용을 초래할 수 있다. 랜덤 테스트와 같은 저비용의 테스트 기법을 사용하여 우선 테스트 데이터를 생성한 후에 여전히 실행이 되지 않은 프로그램이나 블록들만을 대상으로 테스트 데이터를 생성하는 방식이 보다 효과적일 수 있다.

목적 지향 콘콜릭 테스트는 이때 적용할 수 있는 효과적인 방법으로 간주된다. 실험 결과에서 목적 지향 콘콜릭 테스트 방법은 목표 분기를 실행하기에 앞서 탐색해야 하는 공간이 상당히 큰 경우에는 기존의 콘콜릭 테스트에 비해 매우 효과적임을 보였다.

제 2 절 향후 연구

향후 연구에서는 포인터를 사용하는 프로그램과 통합 테스팅을 지원할 수 있도록 한 개 이상의 프로시저에 걸친 프로그램을 처리할 수 있도록 확장하는 것과 논문에서 구현한 목적 지향 콘콜릭 테스트(GCT)를 서버에 올린 후, CREST와 같은 테스트 환경이 구축되어 있지 않은 사용자들도 웹을 통해 테스트를 수행하고 결과를 확인할 수 있도록 확장할 계획이다.

【참고문헌】

1. 국외문헌

Bottaci. L. , "Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithm", In *Proc. of the Genetic and Evolutionary Computation Conf.(GECCO'02)*, pp. 1337-1342, NY, USA, July 2002.

Burnim. J. , Sen. K. , "Heuristics for Dynamic Test Generation", In *the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

Dutertre. B. , and Moura. L. , "A Fast Lineararithmetic Solver for DPLL(T)," International Conference on Computer Aided Verification, 2006

Edvardsson. J. , "A Survey on Automatic Test Data Generation", In *Proc. the Second Conf. on Computer Science and Engineering,* pp. 21-28, 1999.

Ferrante. J , Ottenstein. J. , and Warren. J. , "The Program Dependence Graph and its Use in Optimization". *ACM Trans. Softw. Eng., Methodology*, Vol. 2, No. 9, pp. 319-349, 1987.

Godefroid. P. , Klarlund. N. , and Sen. K. , “DART: Directed automated random testing”, In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (PLDI), 2005.

Necula. G. C. , McPeak. S. , Rahul. S. P. , and Weimer. W. , “CIL: Intermediate Language and Tools for Analysis and transformation of C Programs”. In *Proc. of Conference on compiler Construction*, pp. 213-228, 2002.

SMT-LIB: The Satisfiability Modulo Theories Library
http://combination.cs.uiowa.edu/smtlib/.

Xu. Z. , and Rothermel. G. , "Directed Test Suite Augmentation", In *Proc. of 16th APSEC*, pp. 406-413, 2009.

【부 록】

실험에 사용된
예제 프로그램 목록

flag_assigntmen.c

```
int main(int a, int b){

        int flag = 0;

        if(a == 10)
                flag = 1;

        if(b == 10){
                if(flag)
                        fprintf(stderr, "GOAL!!!!!\n");
        }
}
```

flag_multiple.c

```
int main(int a, int b) {

        int a, b;
        int initialized = 0;
        int has_been_fired = 0;

         if(a != 0)
                initialized = 1;

         if(b != 0)
                has_been_fired = 1;

         if(initialized){
                 if(b == 1)
```

```
                                    has_been_fired = 1;


                        if(has_been_fired)
                                    fprintf(stderr, "GOAL!\n");
            }
}
```

loop.c

```
int main(void){

        int a[10];
        int flag = 1;

        for(int i = 0; i< 10; i++){
                if(a[i] != 10){
                        flag = 0;
                }
        }

        if(flag){
                        fprintf(stderr,"GOAL!\n");
        }

}
```

loop2.c

```
int main(int a[], int b[]){

        int i;
        int fa, fb;

        i = 0;
        fa = 0;
        fb = 0;

        while(i < 10){
                if(a[i] == 10)
                fa = 1;
                i = i + 1;
        }

        if(fa == 1){
                fb = 1;
                i = 0;
                while(i < 10){
                        if(b[i] != 10)
                                fb = 0;
                        i = i + 1;
                }
        }

        if(fb == 1)
                fprintf(stderr,"GOAL!!\n");
}
```

loop3.c

```
int main(int a[], int b[]){

        int i;
        int fa, fb;

        i = 0;
        fa = 1;
        fb = 0;

        while(i < 10){
                if(a[i] != 10)
                fa = 0;
                i = i + 1;
        }

        if(fa == 1){
                fb = 1;
                i = 0;
                while(i < 10){
                        if(b[i] != 10)
                                fb = 0;
                        i = i + 1;
                }
        }

        if(fb == 1)
                fprintf(stderr,"GOAL!\n");
}
```

loop4.c

```
int main(int a[], int b[]){

        int i;
        int fa, fb;

        i = 0;
        fa = 1;
        fb = 0;

        while(i < 10){
                if(a[i] != 10)
                fa = 0;
                i = i + 1;
        }

        if(fa == 1){
                fb = 0;
                i = 0;
                while(i < 10){
                        if(b[i] == 10)
                                fb = 1;
                        i = i + 1;
                }
        }

        if(fb == 1)
                fprintf(stderr,"GOAL!\n");
}
```

loop5.c

```
int main(int a[], int b[]){

        int i;
        int fa, fb;

        i = 0;
        fa = 0;
        fb = 0;

        while(i < 10){
                if(a[i] == 10)
                fa = 1;
                i = i + 1;
        }

        if(fa == 1){
                fb = 0;
                i = 0;
                while(i < 10){
                        if(b[i] == 10)
                                fb = 1;
                        i = i + 1;
                }
        }

        if(fb == 1)
                fprintf(stderr,"GOAL!\n");
}
```

trityp.c

```
int main(int Side1, int Side2, int Side3;){

        int triOut = 0;
        int Side1, Side2, Side3;

         if(Side1 <= 0 || Side2<=0 || Side3<= 0){
                  fprintf(stderr,"Not Triangle1!\n"); //삼각형이 아님
                  return 0;
        }

         if(Side1 == Side2)
              triOut = triOut + 1;

         if(Side1 == Side3)
              triOut = triOut + 2;

         if(Side2 == Side3)
              triOut = triOut + 3;

         if(triOut == 0){
                   if(Side1+Side2<=Side3||Side2+Side3<=Side1||Side1+Side3<=Side2){
                            fprintf(stderr,"Not Triangle2!\n"); //삼각형이 아님
                   }else{
                            fprintf(stderr,"Triangle!\n"); //삼각형
                   }
                    return 0;
         }

         if(triOut > 3){
               fprintf(stderr,"Equilateral Triangle!\n"); //정삼각형
         }else if(triOut==1 && Side1+Side2>Side3){
```

```
                fprintf(stderr,"isosceles Triangle1!\n"); //이등변삼각형
        }else if(triOut==2 && Side1+Side3>Side2){
                fprintf(stderr,"isosceles Triangle2!\n"); //이등변삼각형
        }else if(triOut==3 && Side2+Side3>Side1){
                fprintf(stderr,"isosceles Triangle3!\n"); //이등변삼각형
        }else{
                fprintf(stderr,"Not Triangle3!\n"); //삼각형이 아님
        }

    return 0;
}
```

handle_new_job.c

```
struct Job {
        int state;
        int priority;
};

int main(Job joblist[]){
        struct Job joblist[10];
        int check_work = 0;

        for(int i = 0 ; i < 10 ; i++){
                CREST_int(joblist[i].state);
        }

        for(int i = 0 ; i < 10 ; i++){
                if(joblist[i].state != 7)
                        check_work = 1;
        }
```

```
        if(!check_work){
                fprintf(stderr,"GOAL!\n");
        }

        return 0;
}
```

simpleloop.c

```
int main(char c[], char s[]){

        int state, i;

        for(i = 0; i < 30; i ++){
                state = 0;
                if(c[0] == '[' && state == 0)   state = 1;
                if(c[1] == '(' && state == 1)   state = 2;
                if(c[2] == '{' && state == 2)   state = 3;
                if(c[3] == '~' && state == 3)   state = 4;
                if(c[4] == 'a' && state == 4)   state = 5;
                if(c[5] == 'x' && state == 5)   state = 6;
                if(c[6] == '}' && state == 6)   state = 7;
                if(c[7] == ')' && state == 7)   state = 8;
                if(c[8] == ']' && state == 8)   state = 9;

                if ( s[0] == 'r' &&
                        s[1] == 'e' &&
                        s[2] == 's' &&
                        s[3] == 'e' &&
                        s[4] == 't' &&
                        s[5] == 0 &&
```

```
                                state == 9 ){
                                printf("error");
                                scanf("%d",&state);
                        }
                }

                return 0;
}
```

netflow.c

```
int main(int low[], int high[]){

                int i = 0;
                int violation = 1;

                while(i<10){
                        if(low[i] >= high[i]){
                                violation = 0;

                          }
                        i++;
                }

                if(violation){
                        fprintf(stderr,"GOAL!\n");
                }

                return 0;
}
```

alltrue.c

```
int main(int a[]){

        int alltrue = 1

        for(int i = 0; i< 10; i++){
                alltrue = alltrue && a[i];
        }

        if(alltrue)
                fprintf(stderr,"GOAL!!!\n");

        return 0;
}
```

ABSTRACT

Goal-oriented Concolic Testing.

Park Jung kyu
Major in Computer Engineering
Dept. of Computer Engineering
Graduate School
Hansung University

Concolic testing generates test data by combining concrete program execution and symbolic execution to achieve high test coverage. CREST is a representative open-source test tool implementing concolic testing. Currently, however, CREST aims at exploring all possible execution paths. In case of testing a specific branch or block, thus, it can be ineffective. This paper suggests a goal-oriented concolic testing that generates test data to execute a given branch or block.