코드 분석 기술을 활용한 화이트박스 테스팅

슈어소프트테크㈜
배현섭

목차

화이트박스 테스팅 기법

테스트 설계
테스트 수행 및 모니터링
모니터링 기술

테스팅에 필요한 정적 분석 기술

테스트 데이터 생성 기술
코드 변환 및 모니터링 기술

동적 테스팅

테스팅의 목적 및 효과

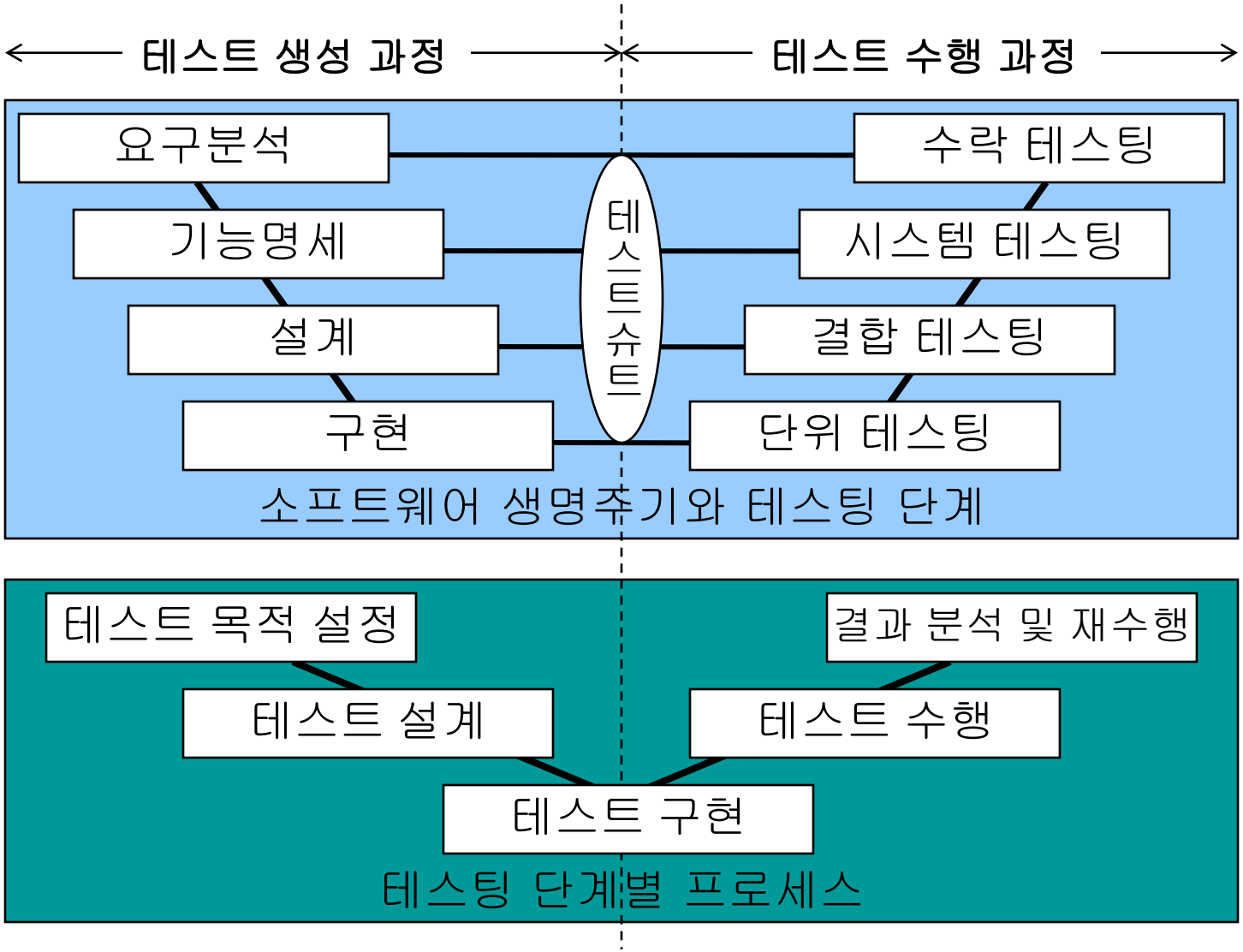
- 오류의 존재를 밝힘
- 오류의 부재를 입증하지는 못함

테스팅 기법 분류

- 정보의 출처에 따라
  - 구조적 테스팅 (structural testing, white-box testing)
  - 기능적 테스팅 (functional testing, black-box testing)
- 테스팅 목적 및 적용 시점에 따라
  - 컴포넌트 테스팅 : 단위 테스팅, 모듈 테스팅
  - 결합 테스팅 : 서브시스템 테스팅, 시스템 테스팅
  - 사용자 테스팅 : 시스템 테스팅, 수용성 테스팅

테스팅 이론

- Dual V Model



Testing Issues

Is an exhaustive input test possible?

Can We find a few input values that represent the entire input domain?

Is an exhaustive path test possible?

How can we handle loops?

How can we generate the test data?

KAIST 권용래 교수님 강의 노트에서 인용

테스트 케이스 설계 기법

Exhaustive Testing

- 가능한 모든 테스트 케이스를 모두 시험하는 기법
- 이론적으로/현실적으로 불가능

Reduce the number of test cases, while still achieving a good coverage!

Divide the program's input space into equivalent domains.
Select one test case from each equivalence class.

- All inputs from the same equivalence class have an equal chance of finding a defect
- Testing with more inputs from the same class hardly increases the chance of finding defects

- *Partition Testing*
- *Domain Testing*

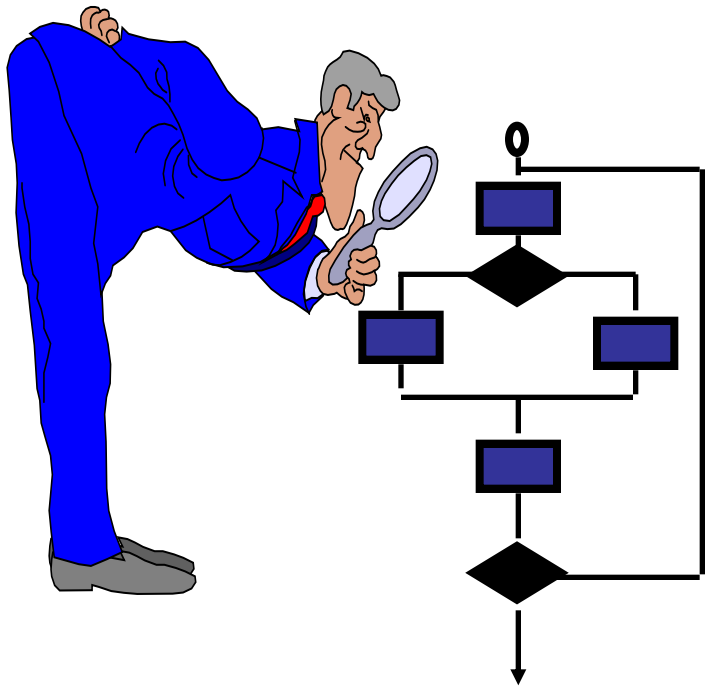
Selective Testing

- Black-box testing: 명세로부터 테스트 케이스를 선택
- White-box testing: 코드로부터 테스트 케이스를 선택
- User-driven testing: 사용자 행동으로부터 테스트 케이스 녹화
- Random testing

White-box Testing

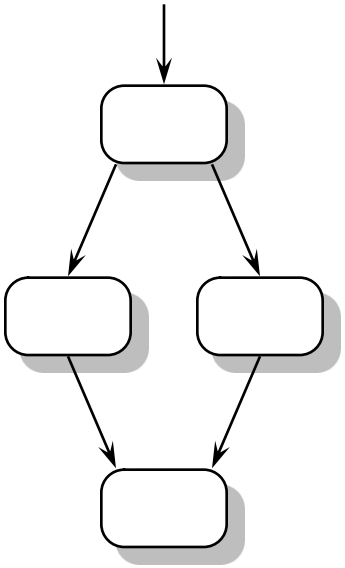
특징

- 모듈 안의 작동을 자세히 관찰하는 시험
- 프로그램의 내부 구조에 대한 지식을 이용하여 *테스트 케이스 생성*
- 테스트 수행 과정에서 프로그램 내부에 대한 *모니터링* 시도
- cf) 구조적 시험(structural testing), 코드 기반 시험(code-based testing)

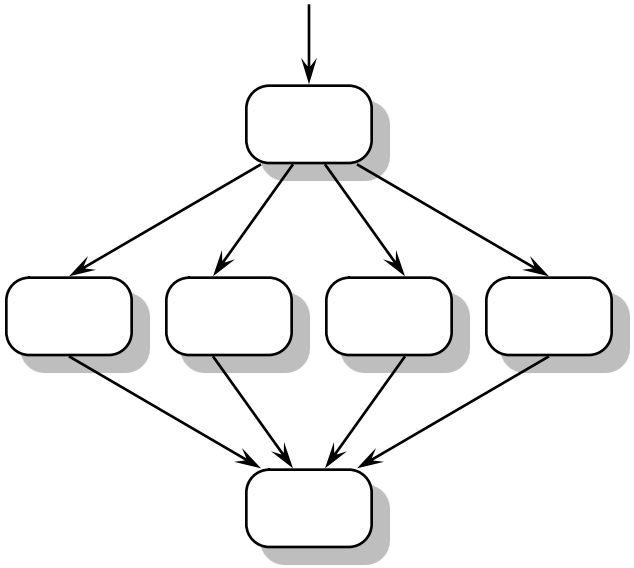


제어 흐름 그래프 (Control Flow Graph)

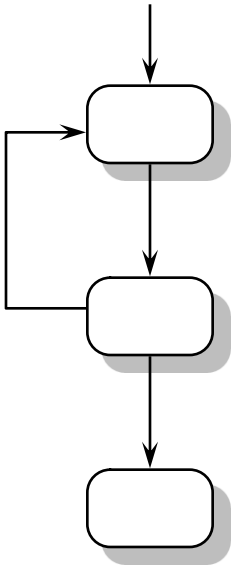
- 프로그램의 제어 구조를 그래프 형태로 나타냄
- 그래프 구성 요소: 블록(block), 분기(decision)
- 프로그램 제어 구조에 대한 표현 방법



if-then-else



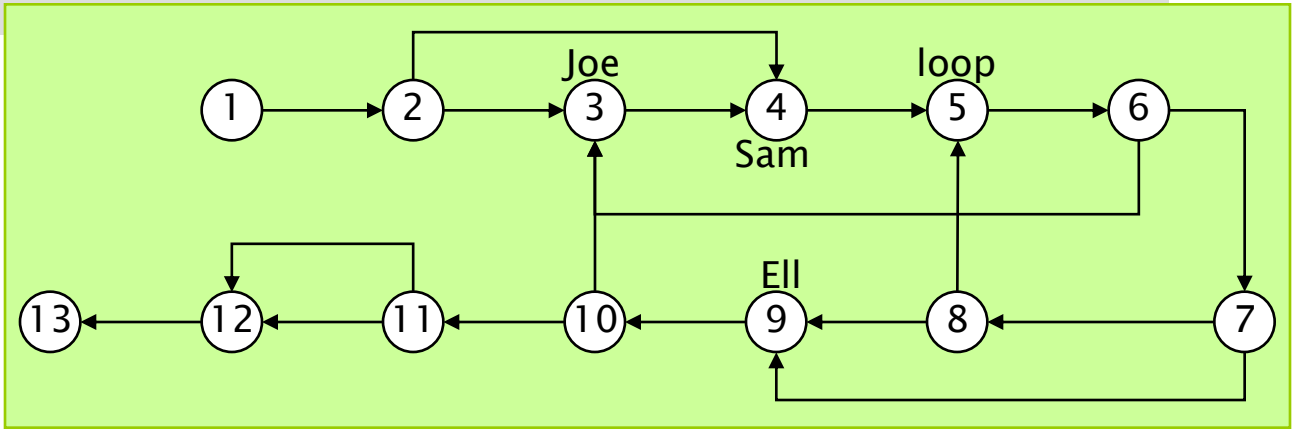
case-of



loop

제어 흐름 그래프 예제

```
        input x, y
        z := x + y
        v := x - y
        if z >= 0 goto Sam
Joe:    z := z - 1
Sam:    z := z + v
        for u = 0 to z
        v(u),u(v) := (z+v)*u
        if v(u) = 0 goto Joe
        z := z - 1
        if z = 0 goto Ell
        u := u + 1
        next u
        v(u-1) := v(u+1) + u(v-1)
Ell:    v(u+u(v)) := u + v
        if u = v goto Joe
        if u > v then u := z
        z := u
        end
```

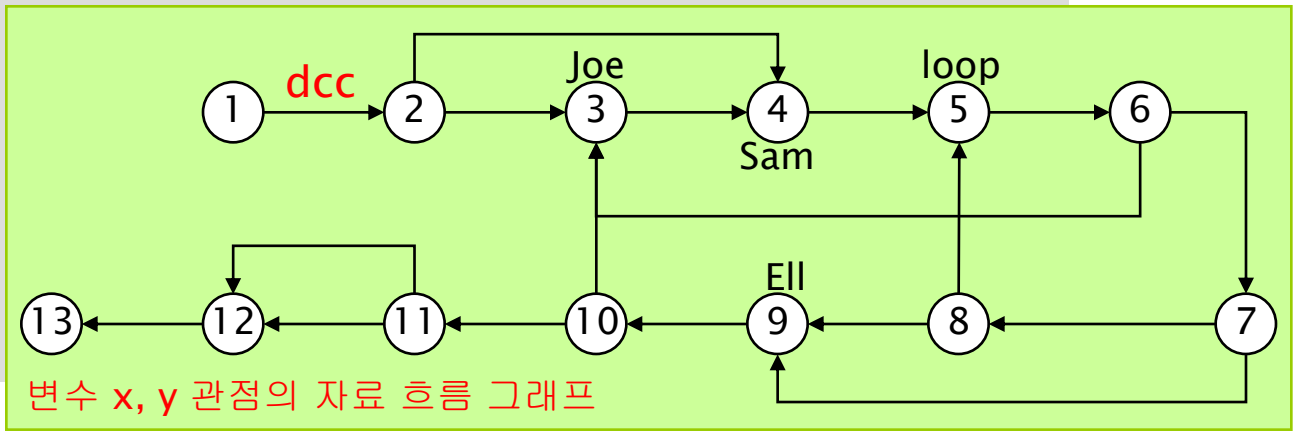


자료 흐름 그래프 (Data Flow Graph)

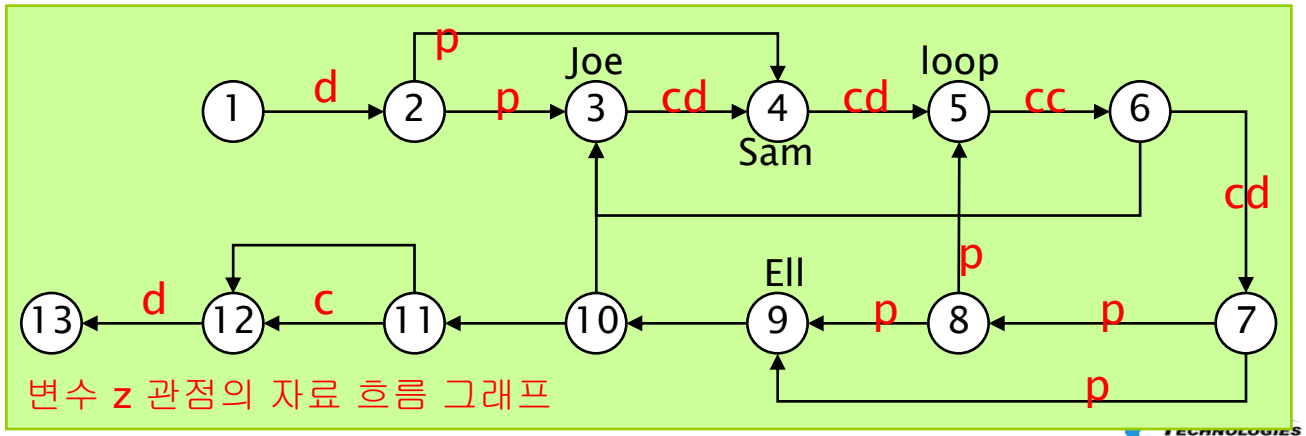
- 제어 흐름 그래프에 데이터 사용 현황을 추가한 그래프
- 데이터 사용 유형
  - 정의(d, defined, created, initialized)
  - 소멸(k, killed, undefined, released)
  - 사용(u, used)
    - 계산에 사용(c, used in a calculation)
    - 결정에 사용(p, used in a predicate)

자료 흐름 그래프 예제

```
        input x, y
        z := x + y
        v := x - y
        if z >= 0 goto Sam
Joe:    z := z - 1
Sam:    z := z + v
        for u = 0 to z
        v(u),u(v) := (z+v)*u
        if v(u) = 0 goto Joe
        z := z - 1
        if z = 0 goto Ell
        u := u + 1
        next u
        v(u-1) := v(u+1) + u(v-1)
Ell:    v(u+u(v)) := u + v
        if u = v goto Joe
        if u > v then u := z
        z := u
        end
```



변수 x, y 관점의 자료 흐름 그래프



변수 z 관점의 자료 흐름 그래프

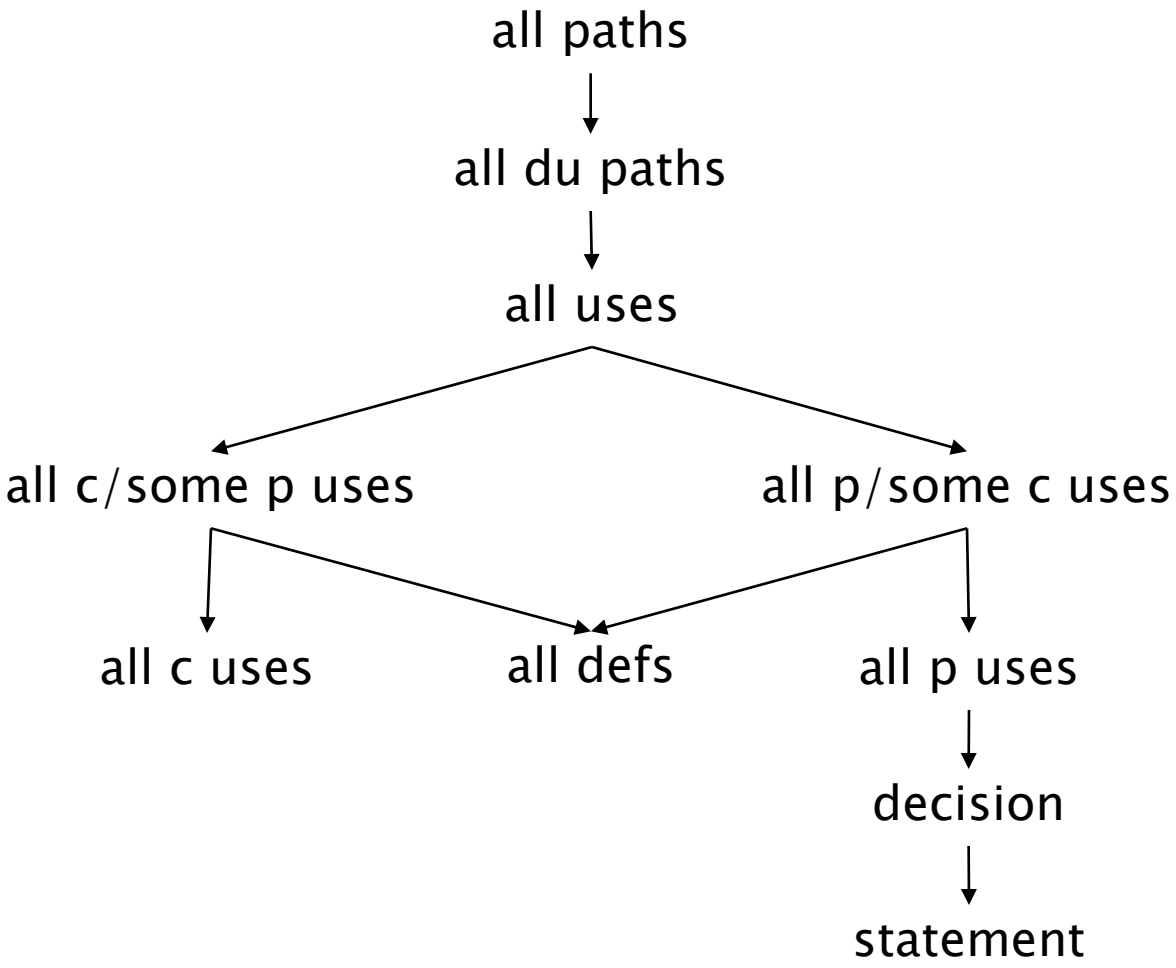
테스트 커버리지 (Test Coverage)

의미: 그래프에 대한 테스팅 기준

커버리지 종류

1) 제어 흐름 기반 커버리지 (Control Flow Coverage)
   - 문장 커버리지 (statement coverage)
   - 분기 커버리지 (branch/decision coverage)
   - 조건 커버리지 (condition coverage)
   - 기본 경로 커버리지 (simple path coverage)
2) data flow를 기반으로 하는 커버리지
   - All defs: every definition should be tested at least one use of that definition
   - All uses: test at least one path segment from every definition to every use which is reachable from that definition
   - All du paths: test every du-path from every definition to every use

테스트 커버리지 (Test Coverage)



경로 선택 (Path Selection)

의미

- 목적하는 테스트 커버리지를 달성하기 위한 **entry/exit path**를 선택
- 가능한 적은 수의 **entry/exit path** 만으로 커버리지를 달성하고자 함

테스트 케이스 생성 (Path Sensitization)

정의

- 선택된 경로를 수행하기 위한 입력 데이터를 결정하는 것

방법

- **path predicate**: 선택된 경로를 수행하기 위해서 만족해야 하는 조건들
- **path predicate**을 만족하는 입력 데이터를 구함으로서 테스트 케이스 생성

문제점

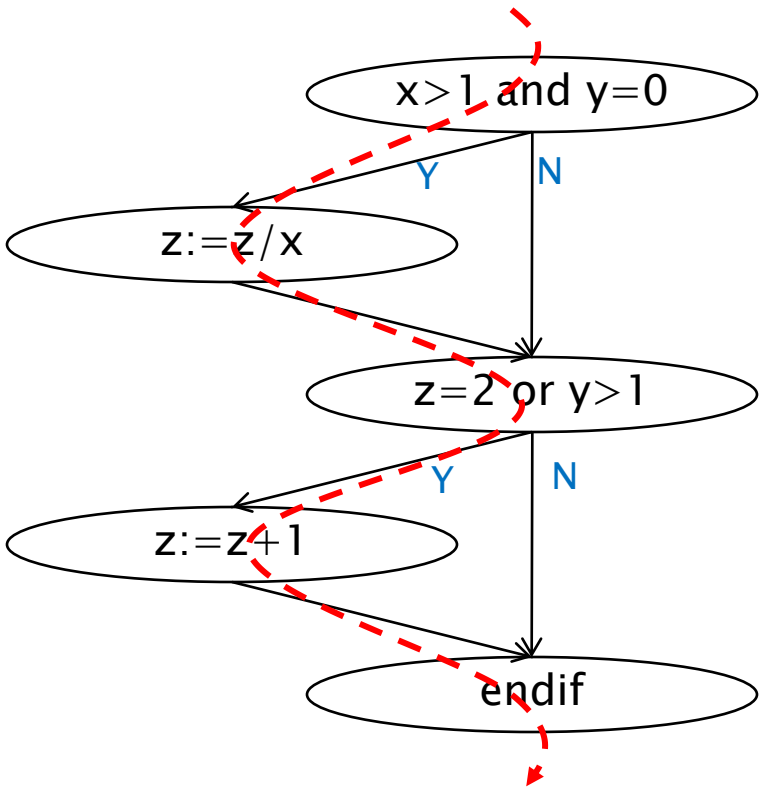
- **path predicate**에 대한 해가 존재하지 않는 경우: **unachievable path** 이므로 다른 경로를 선택
- **path predicate**을 풀지 못하는 경우: 임의의 방정식/부등식에 대한 일반해를 구하는 방법이 없음

문장 커버리지 (Statement Coverage)

- **Definition.** Every executable statement in the program is invoked at least once during software testing
- shows that all code statements are reachable
- a weak criterion because it is insensitive to control structures
- example

  if (x>1) and (y=0) then z:=z/x; endif;

  if (z=2) or (y>1) then z:=z+1; endif;



x=2, y=0, z=4

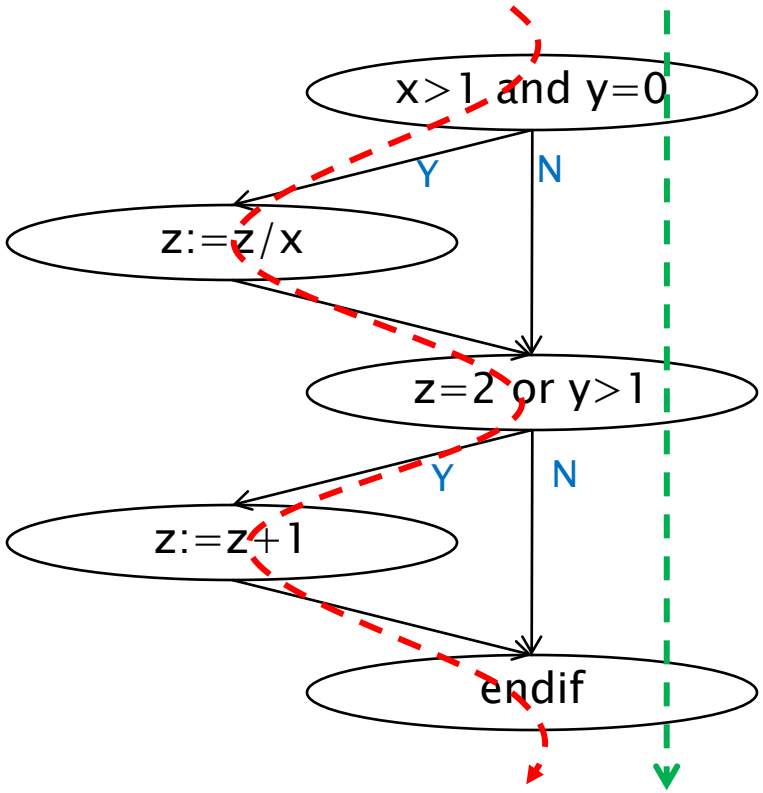
cannot detect mistake in decision statement

분기 커버리지 (Decision Coverage)

- **Definition.** Every decision statement in the program should take on all possible outcomes at least once during software testing
- ensures complete testing of control constructs for simple decision
- how about the complex decisions ?
- example

  if (x>1) and (y=0) then z:=z/x; endif;

  if (z=2) or (y>1) then z:=z+1; endif;



x=2, y=0, z=4 (TT-TF)

x=1, y=0, z=4 (FT-FF)

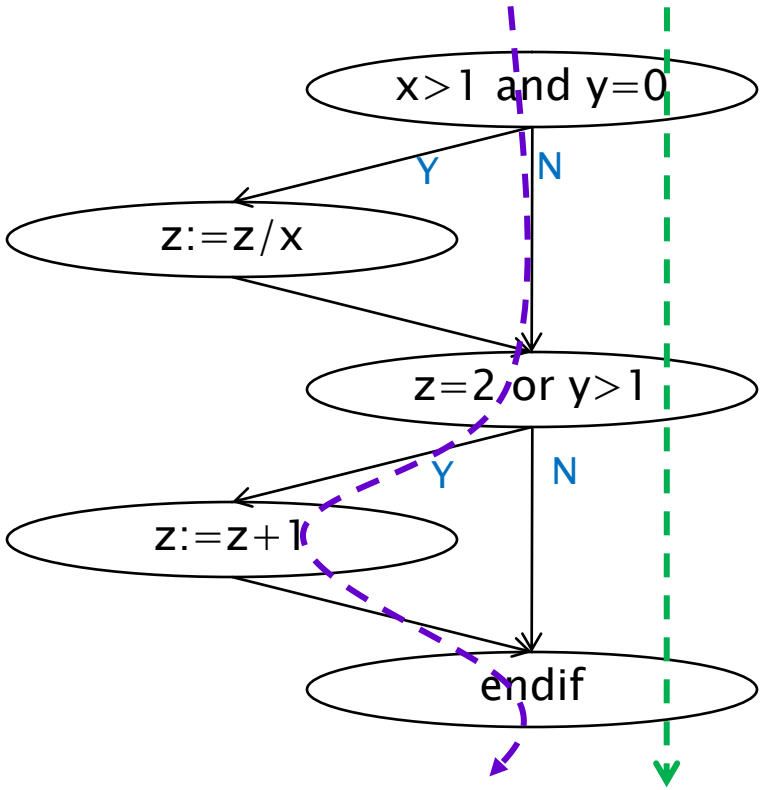
cannot distinguish simple decision from complex decision

조건 커버리지 (Condition Coverage)

- **Definition.** Each condition in a decision take on all possible outcomes at least once during software testing
- do not cause the decision to take on all possible outcomes
- example

  if (x>1) and (y=0) then z:=z/x; endif;

  if (z=2) or (y>1) then z:=z+1; endif;



x=2, y=2, z=2 (TF-TT)
x=1, y=0, z=4 (FT-FF)

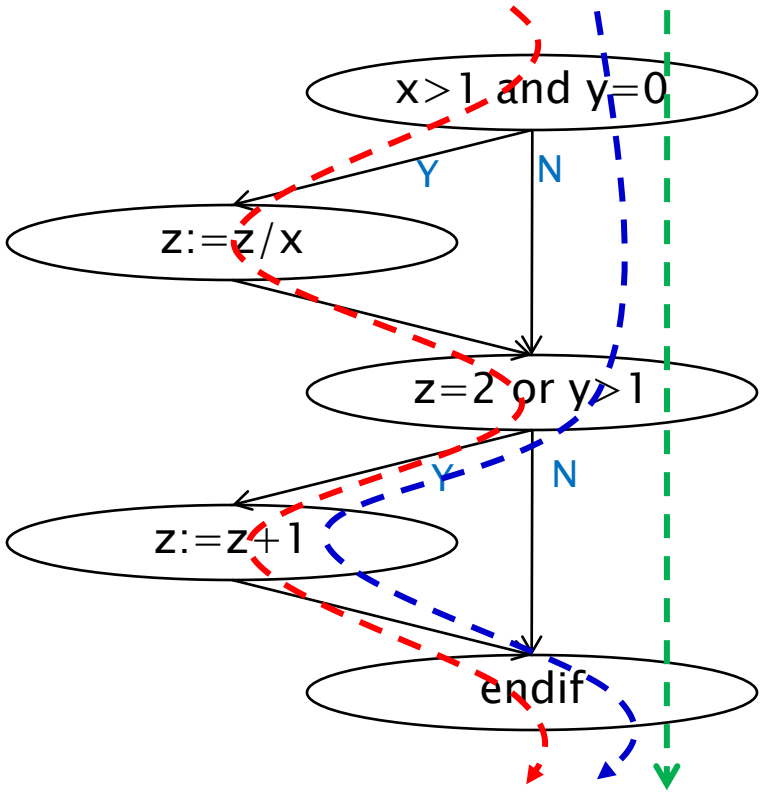
cannot satisfy even statement coverage

변형 조건/분기 커버리지 (Modified Condition/Decision Coverage)

- **Definition.** MC/DC criterion enhances the condition/decision coverage by requiring that each condition be shown to independently affect the outcome of the decision
- ensures that the effect of each condition is tested relative to the other conditions
- example

  if (x>1) and (y=0) then z:=z/x; endif;

  if (z=2) or (y>1) then z:=z+1; endif;



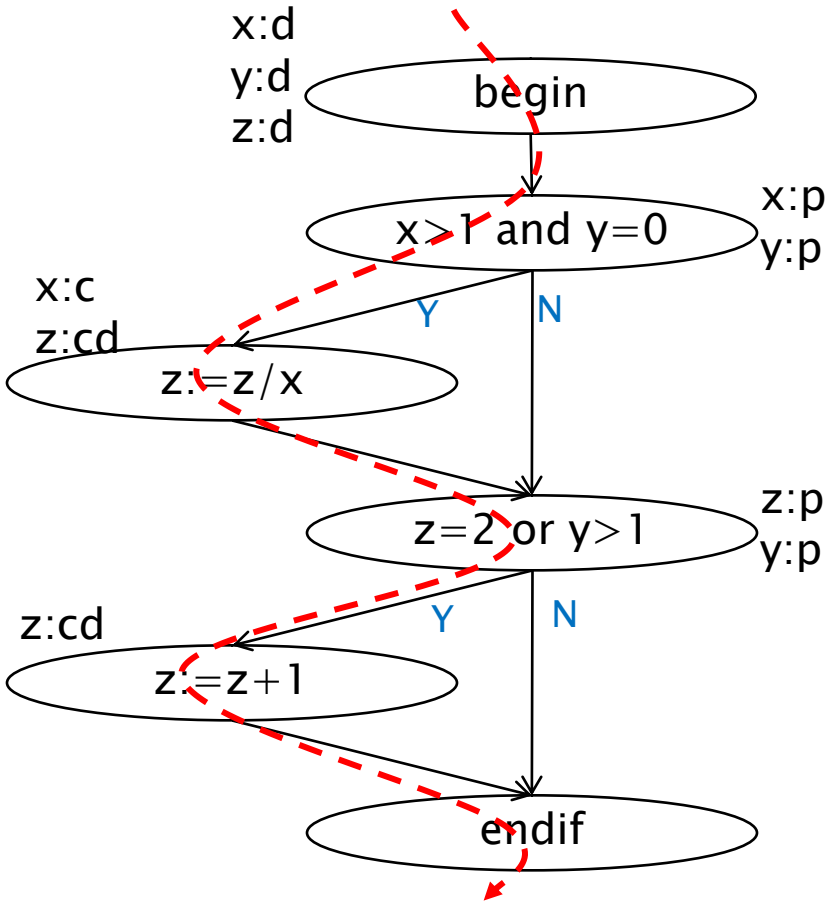
x=2, y=0, z=4 (TT-TF)
x=2, y=2, z=4 (TF-FT)
x=1, y=0, z=4 (FT-FF)

for decisions with a large number of inputs, MC/DC requires considerably more test cases than C/DC

All defs 커버리지

- **Definition.** Every definition of every variable should be tested at least one use of that definition
- example

```
if (x>1) and (y=0) then z:=z/x; endif;
if (z=2) or (y>1) then z:=z+1; endif;
```



x=2, y=0, z=4

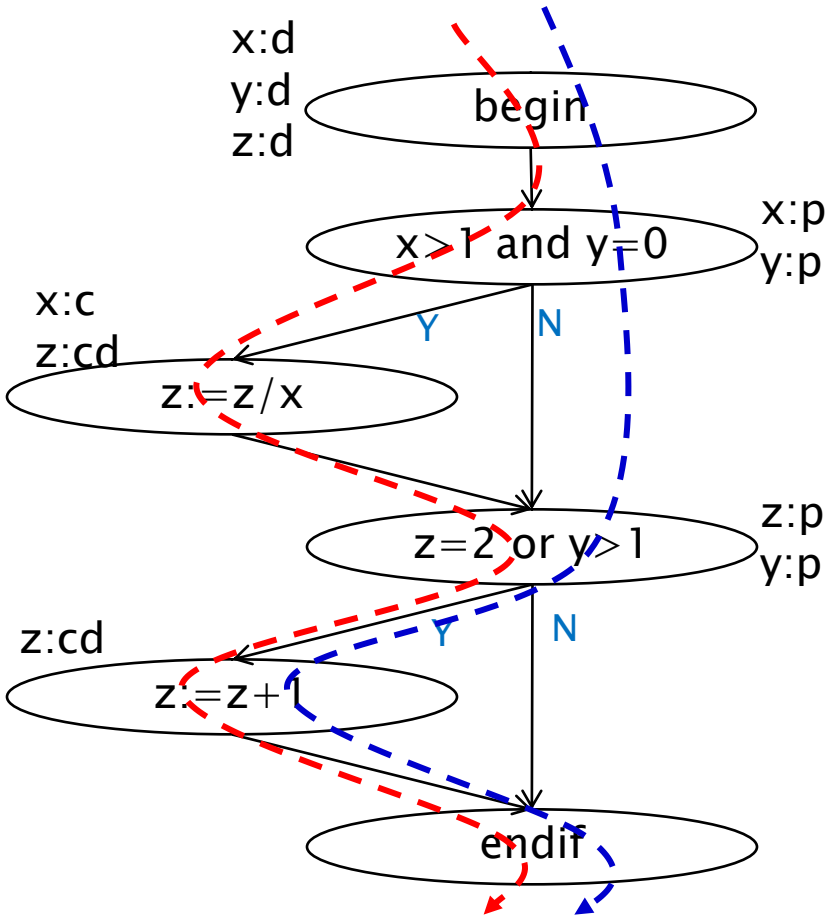
4 definitions are covered by
3 p-uses and 1 c-uses

All uses 커버리지

- **Definition.** At least one path segment from every definition of every variable to every reachable use should be tested
- example

  if (x>1) and (y=0) then z:=z/x; endif;

  if (z=2) or (y>1) then z:=z+1; endif;



There are 2+2+3+2+0=9 du-pairs in this program

x=2, y=0, z=4
covers 2+2+1+2=7 du-pairs

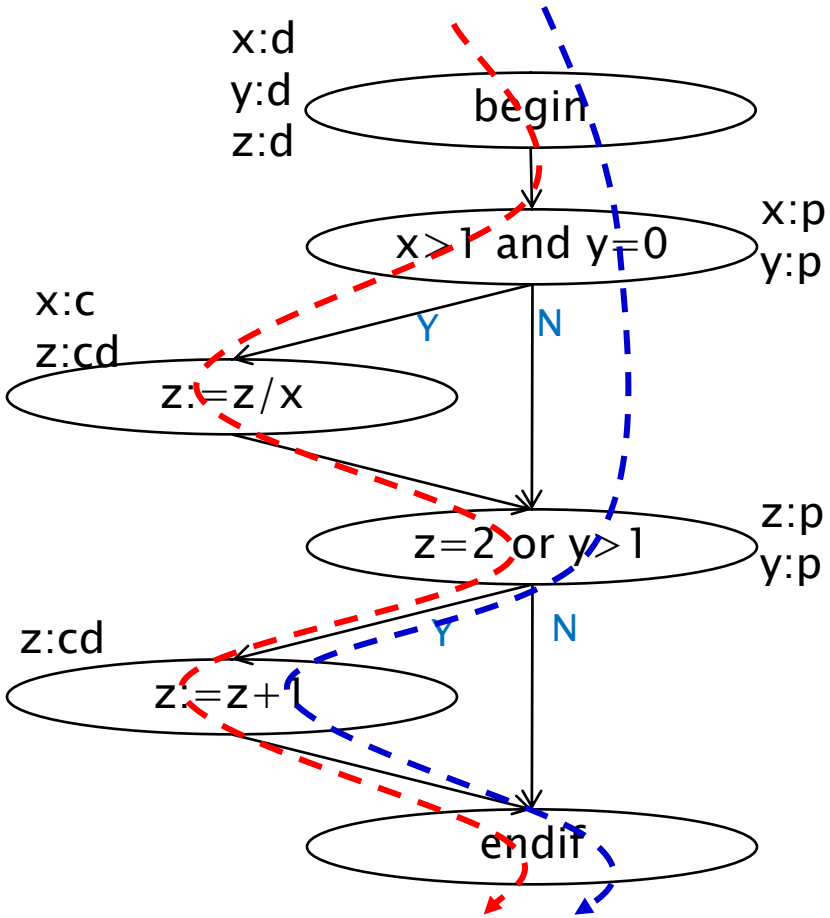
x=2, y=2, z=4 (TF-FT)
covers another 2 du-pairs

All du path 커버리지

- **Definition.** Every du-path from every definition of every variable to every reachable use should be tested
- example

  if (x>1) and (y=0) then z:=z/x; endif;

  if (z=2) or (y>1) then z:=z+1; endif;



There are 10 du-paths for 9 du-pairs in this program

x=2, y=0, z=4
covers 2+2+1+2=7 du-paths

x=2, y=2, z=4 (TF-FT)
covers another 3 du-paths

커버리지 모니터링

정의

- 테스트 수행 경로를 파악하기 위해서 수행 산출물을 만들고 관찰하는 것

방법

- 현재 커버리지 모니터링은 모두 탐침 삽입 (probe instrumentation)에 기반을 두고 있음
- link marker: CFG 내의 각 에지에 대해서 유일한 식별자를 붙임
- link counter: CFG 내의 각 에지에 대해서 카운터를 붙임

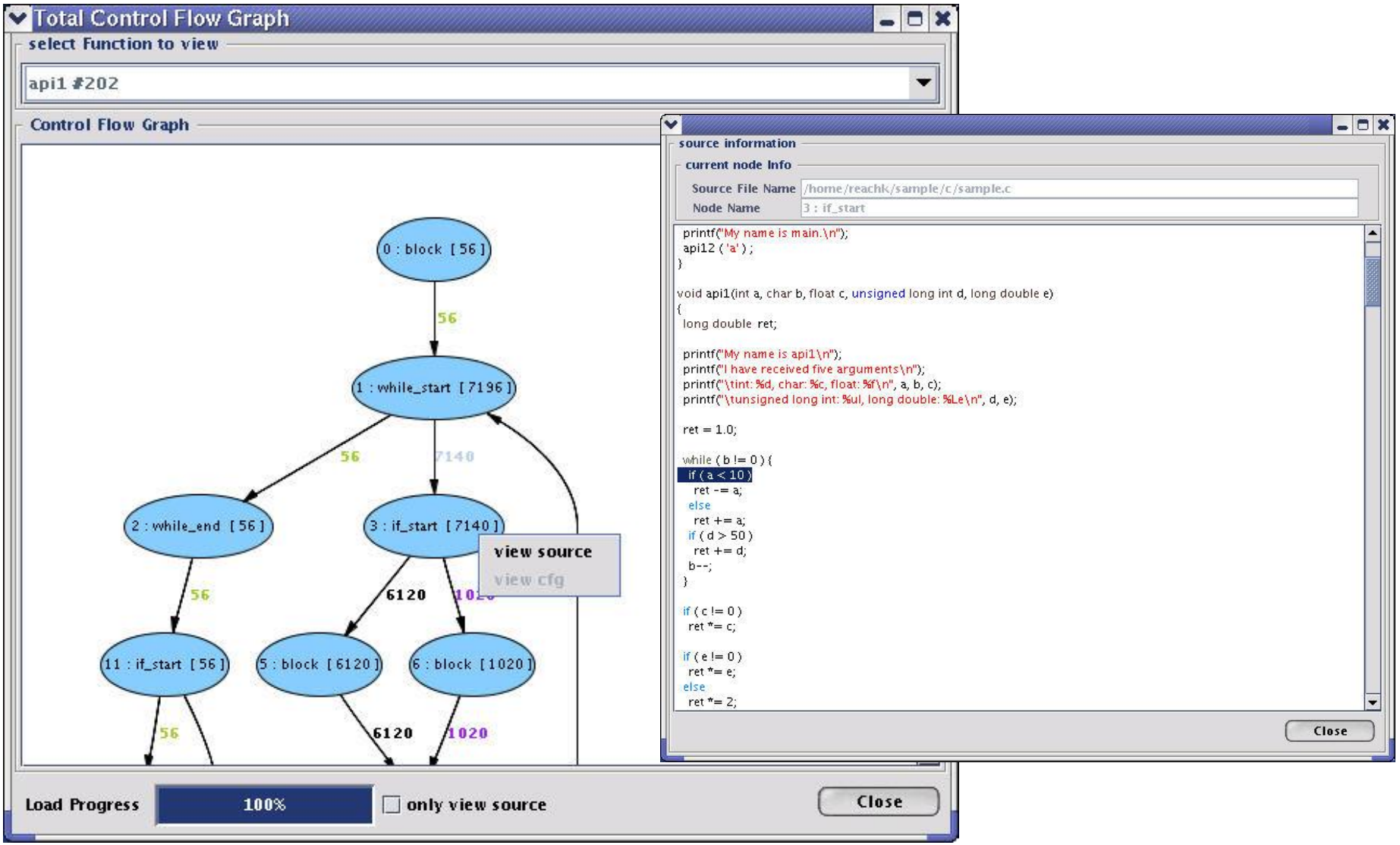
문제점

- exit edge에 instrumentation 불가능한 경우가 많음
- 함수 호출을 edge로 모델링하지 않기 때문에 부정확한 경우가 있음

```
probe("before return");
return f(x);
probe("after return");  // 의미 없음
// 즉, return f(x) 문장의 성공 종료 여부를 알 수 없음
```

```
probe("before block");  // 진짜 오류의 위치는?
...      // p1
x = g() + h();
...      // p2
probe("after block");
```

커버리지 모니터링 결과



테스팅을 위한 정적 분석 기술 (1/2)

- **테스트 데이터 생성기술**
  - 주어진 경로 혹은 주어진 지점을 지나는 테스트 입력 데이터를 생성하는 기술
  - 경로에 해당하는 path predicates를 푸는 문제로 귀결됨
  - 현재 수준
    - with very simple analysis: achieve 40~60% coverage
    - with complex analysis (model checking & constraint solving)

소요시간	평균 1.5초 최악의 경우 20초
효과 (데이터 생성 성공률)	25% 내외의 성공률 (함수 호출 무시하는 경우) 8% 내외의 성공률 (함수 호출을 고려한 경우)

테스팅을 위한 정적 분석 기술 (2/2)

- **프로그램 모니터링 기술**
  - 프로그램 수행 중에 (on-the-fly) 프로그램 내부 상황을 모니터링 하기 위한 기술
  - 프로그램 자동 변환 기술 + 로그 데이터 수집 및 분석 기술
  - 이슈: 프로그램의 원래 행동에 영향을 끼치지 않아야 함
  - 모니터링 대상
    - 수행 경로
    - 입,출력 데이터
    - 메모리 사용 현황
  - 기술 현황
    - 수행 경로 모니터링: 오버헤드 30% 내외
    - 입,출력 데이터 모니터링: ???
    - 메모리 사용현황 모니터링: 오버헤드 200% 내외