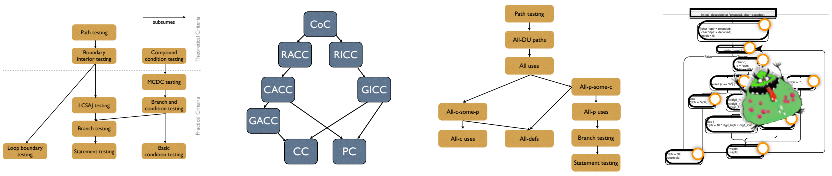
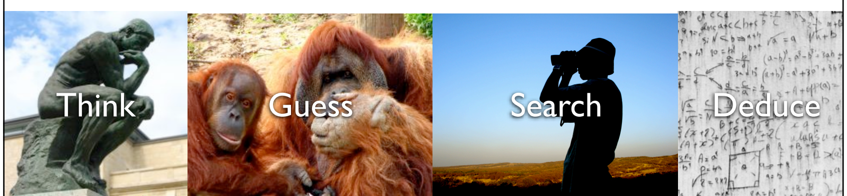
So many different test objectives...



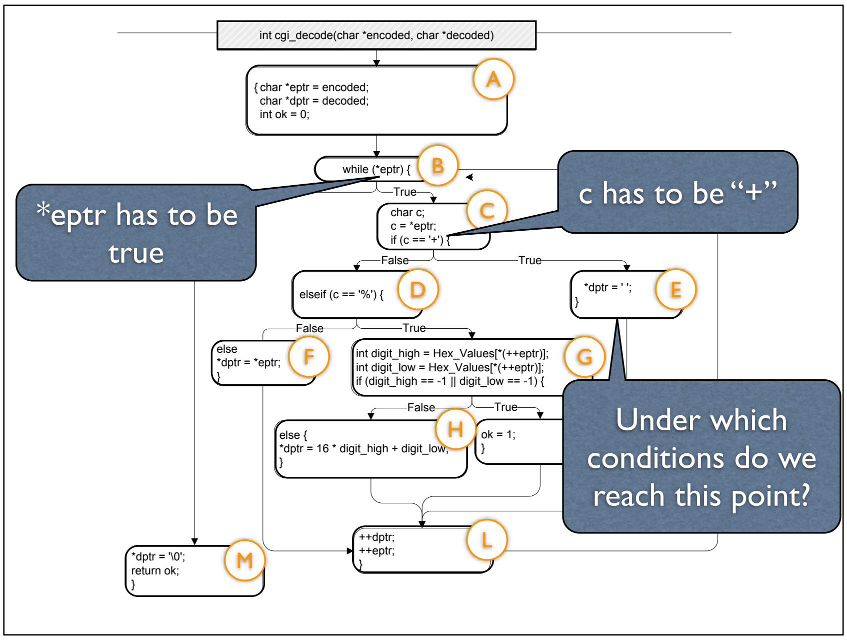
But how to create the tests?

Test Data Generation

Given a function and a location we want to reach, how do we derive inputs to the function that lead the control flow to the desired statement?



We are still looking at the problem of deriving input data that will lead execution to some particular point in the control flow that is interesting for testing reasons. Today, we will consider constraint based testing, which allows us to reason about the precise conditions under which a test goal is satisfied, and allows us to deduce test data satisfying the test goals.



Looking at the control flow graph of the cgi_decode example, we see that in order to reach node E several conditions have to hold - conditions on variables that are altered during the program execution.

Constraint-based Testing

- Constraint generation
  Extract a constraint system from the program and a testing objective
- Constraints on inputs
  If inputs satisfy constraints, then testing objective will be satisfied
- Constraint solving:
  Solve the constraint system to generate test data

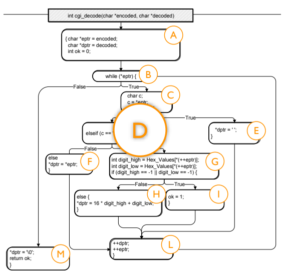
- Static analysis aims at finding runtime errors (e.g. division-by-zero, overflows, ...) at compile time
- CBT aims at finding functional faults (e.g. P returns 3 while 2 was expected)

- Model-checking tools explore paths of software models for proving properties
- CBT looks only for counter-examples

- Dynamic analysis approaches extract likely invariants
- CBT exploits symbolic reasoning to find counter-examples to given properties

Overview

Program Test Goal Constraint system



$(a>b \wedge c = 2) \vee (a>5 \wedge b>a) \vee (a<b \wedge b<5 \wedge a > 0)$

The ida of constraint based testing is to transform a program and a test goal for that program to a constraint system...

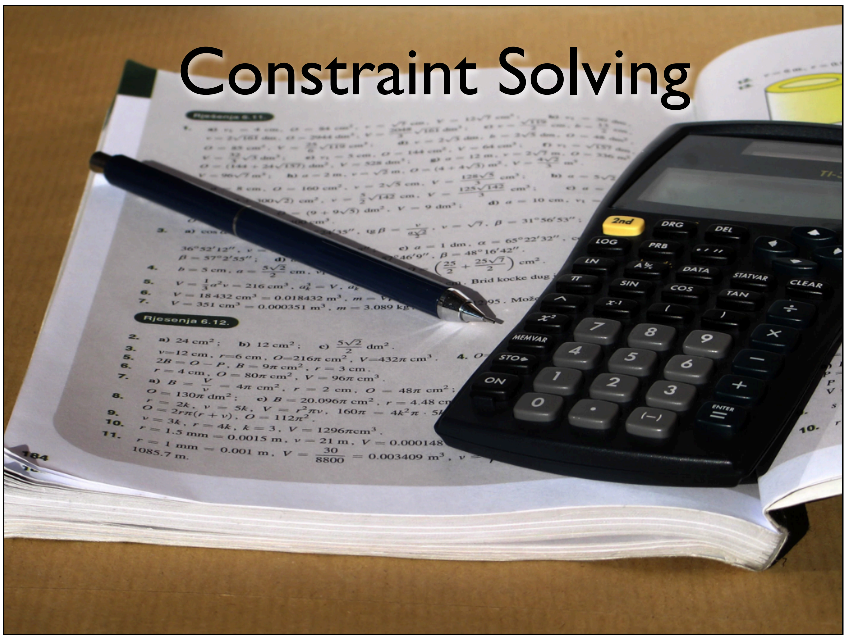
Overview

Constraint system → Constraint solver → Test data

$(a>b \wedge c = 2) \vee (a>5 \wedge b>a) \vee (a<b \wedge b<5 \wedge a > 0)$

a = 1
b = 2
c = 3

...and then use a constraint solver to automatically derive a solution to the constraint system - which is our test data.



What is a constraint?

- A constraint is a condition that a solution to an optimization problem must satisfy
- $x > 5, y < 10$
- $a > b \wedge b > 10$
- Constraint satisfaction:
  Finding value assignments to variables such that constraints are satisfied

Relevant questions: Does the constraint system (CS) have a solution? Can we generate a solution to CS? Can we generate the best solution to CS?

Relevant Questions

- Does the constraint system (CS) have a solution?
  To decide whether the testing objective is reachable or not
- Can we generate a solution to CS?
  Test data generation
- Can we generate the best solution to CS?
  Test data generation that optimizes a cost function

Constraint solving

- Computational domain, constraint language results from the choice of programs and properties to be considered
- Booleans - Boolean formula (A&&B&&!C)||(...)
- Integers
- Bounded Integers
- Rationals
- Reals
- Floating-point numbers
-

Decidability and complexities

	Boolean Formula	Linear constraints	Polynomial constraints	Non-linear constraints
Booleans	2-SAT in P 3-SAT is NP-complete	0-1 programming is NP-complete	?	?
Bounded integers	-	NP-complete	NP-complete	NP-complete
Integers	-	Integer programming is NP complete	Undecidable	Undecidable
Rationals and reals	-	Linear programming in P	Nonlinear programming is NP-complete	Undecidable

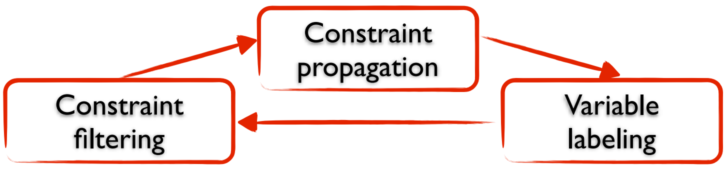
The landscape of complexities for different domains (rows) and types of constraints (columns) looks scary - in practice, however, we can handle anything involving non-linear constraints quite well using heuristics.

Decision procedures (best practices)

	Boolean Formula	Linear constraints	Polynomial constraints	Non-linear constraints
Booleans	Davis & Putnam (DPLL)			
Bounded integers		Cooper's procedure fir Oresburger algorith		
Integers		Constraint satisfaction	Constraint satisfaction	Constraint satisfaction
Rationals and reals		Simplex Fourier Elimination	Groebner basis (Buchberger alg)	Interval propagation

Constraint Satisfaction

- A constraint system involves a set of variables V, a set of finite domains D, and a set of constraints C
- A solution is an assignment of V to values in D that satisfies C
- A constraint system is *unsatisfiable* when it has no solutions
- Constraint satisfaction involves 3 interleaved processes:

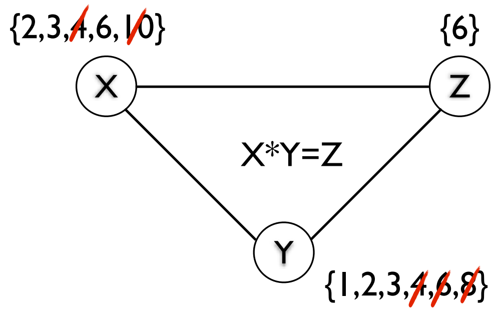


Constraint filtering

- Given a single constraint, filter the domains of variables by removing inconsistent values
- Depends on a level of consistency to be achieved
  Domain consistency - bound consistency - and many more
- Example:
  X in {2,3,4,6,10}, Y in {1,2,3,4,6,8}, Z in {6}, X*Y=Z

Domain Consistency

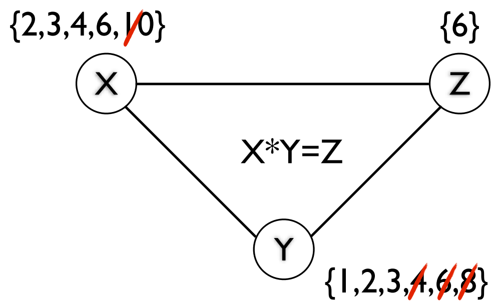
For each value in Dx, find a support in Dz and Dy



Domain consistency checks for each variable of a constraint for which of its values there is support in the values of the other variables.

Bound Consistency

For each bound in Dx, find a support in Dz and Dy



Bound consistency only reduces the domain bounds, and is therefore cheaper.

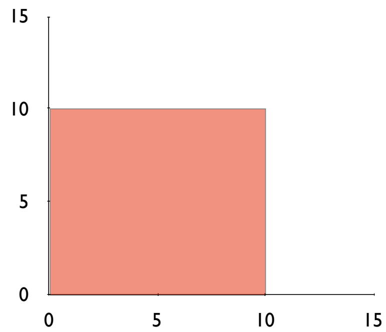
Constraint Propagation

- Propagates prunings throughout the constraint system
- Implemented as a fixpoint algorithm:

```
Agenda := C;
while(!Agenda.isEmpty()) {
  c := POP(Agenda);
  D’ := narrow(c,D);
  if(D’ != D)
    Agenda := Agenda ∪
    {c’ in C | vars(c’) ∩ vars(c)!=∅}
  D := D’
}
return D’;
```

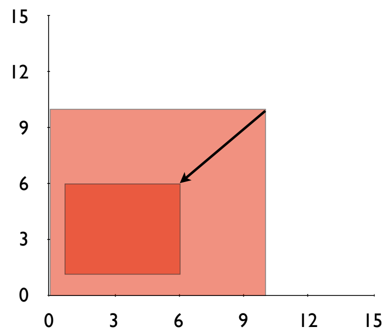
After constraint filtering, the reduced value domains (result of function narrow in above source code) are propagated to other constraints that involve the same variables. This process is implemented as a fixpoint algorithm that iterates until no more changes can be propagated.

Constraint Propagation



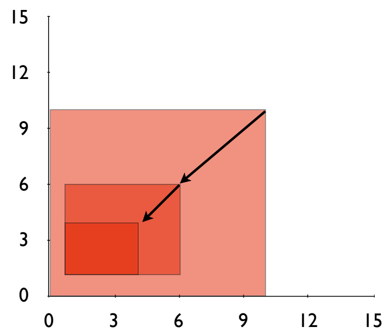
X, Y in 0..10, X*Y=6, X+Y=5

Constraint Propagation



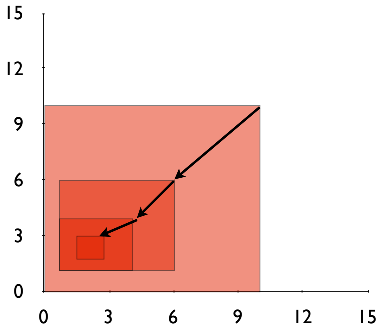
X,Y in 0..10, **X*Y=6**, X+Y=5

Constraint Propagation



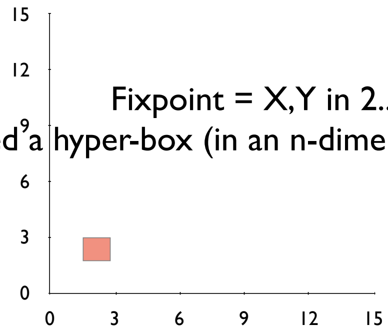
X,Y in 0..10, X*Y=6, **X+Y=5**

Constraint Propagation



X,Y in 0..10, **X*Y=6**, X+Y=5

Constraint Propagation

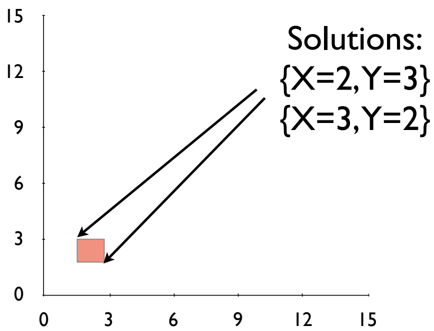


Fixpoint = X,Y in 2..3
called a hyper-box (in an n-dimensional space)

X,Y in 0..10, X*Y=6, X+Y=5

Variable Labeling

Select a value v from the domain of X and propagate X = v



X,Y in 0..10, X*Y=6, X+Y=5

When heuristics for selecting values and variables are complete, labeling is a decision procedure for constraint satisfaction. But, it is also the costly part of it (NP-complete) while constraint filtering and propagation are polynomial in the number of constraints (and values in domains). Routinely in applications, constraint satisfaction handles thousands of constraints and variables

Selection Heuristics

- Leftmost
  Select the leftmost variable in the list
- First-fail
  Select the variable with the smallest domain
- Most-constrained
  Select the var that has the most constraints suspended on it
- And many more

Variable Labeling

- When heuristics for selecting values and variables are complete, labeling is a decision procedure for constraint satisfaction
- But, it is also the costly part of it (NP-complete) while constraint filtering and propagation are polynomial in the number of constraints (and values in domains)
- Routinely in applications, constraint satisfaction handles thousands of constraints and variables

Satisfiability Modulo Theory (SMT)

- To decide the satisfiability of formulas with respect to de[…] ground theories .
  $\Phi ::= A \mid \neg\Phi$ […] Φ
- Numerous ap[…]cluding test data generation
- Used in PEX, through Z3 the SMT-solver of Microsoft

Drop this slide

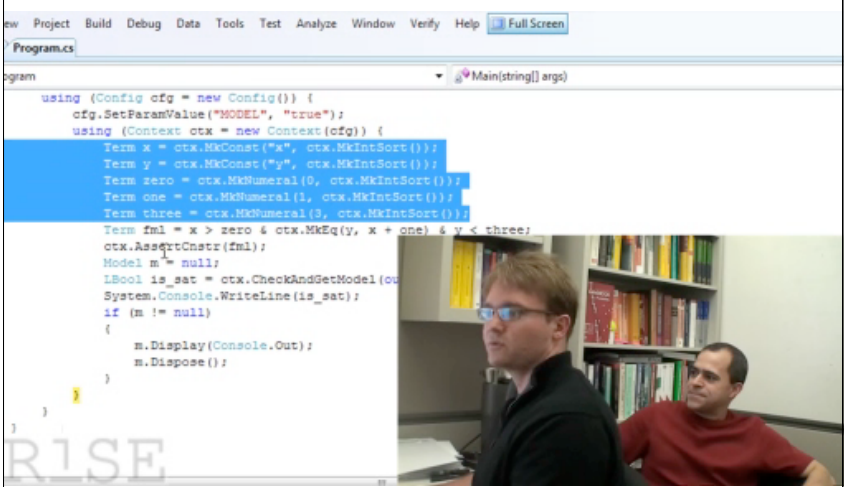
Satisfiability Modulo Theories (SMT)

- Example theories:
  - R: theory of rationals
    SR = {≤, +, -, 0, 1}
  - L: theory of lists
    SL = {=, hd, tl, nil, cons}
  - E: theory of equality
    SE: uninterpreted functions and predicate symbols
- Problem:
  Is $x \le y \wedge y \le x + hd(cons(0, nil)) \wedge P(h(x) - h(y)) \wedge \neg P(0)$
  satisfiable in R.L.E ?

SMT solving builds on the success of SAT solvers, and generalizes boolean satisfiability to include theories.

$$x \le y \wedge y \le x + \underbrace{hd(cons(0, nil))}_{v1} \wedge P(\underbrace{\underbrace{h(x)}_{v3} - \underbrace{h(y)}_{v4}}_{v2}) \wedge \neg P(0)$$

R	L	E
x <= y		!P(0)
y <= x+v1	v1=hd(cons(0,nil))	P(v2)
v2 = v3-v4	v1 = 0	v3 = h(x)
x = y		v4 = h(y)
v2 = 0		v3 = v4
		⊥



Z3 is one of the most powerful SMT solvers currently available. In this video, the authors of Z3 briefly describe constraint solving and demonstrate how to use a constraint solver via API calls. The video is available online at http://channel9.msdn.com/posts/Peli/The-Z3-Constraint-Solver/

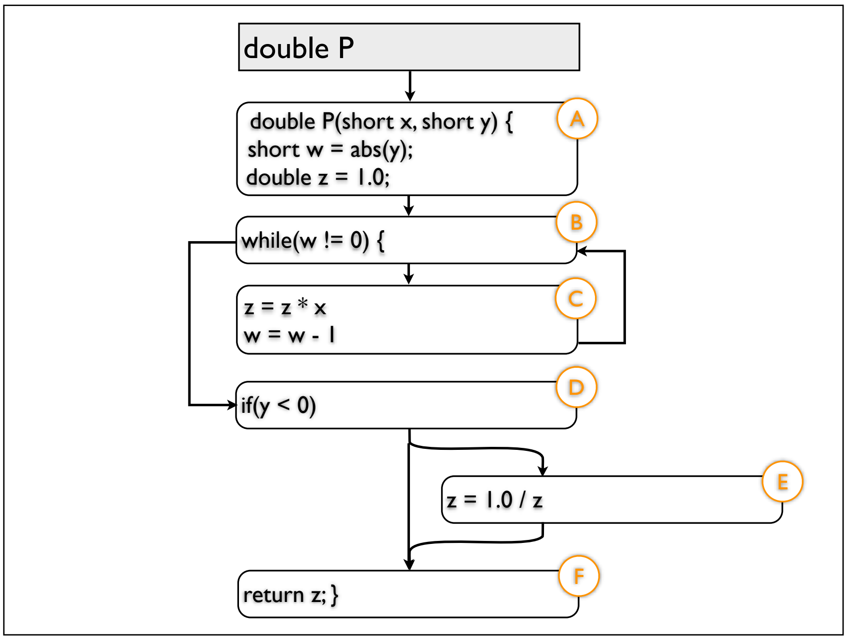
Path-Oriented Generation

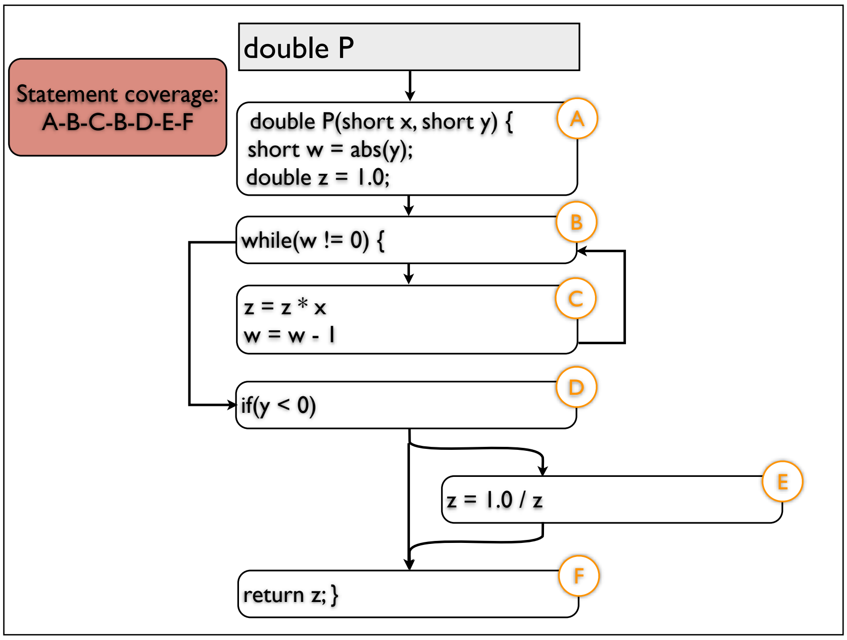
- Select one or several paths - Path selection
- Generate the path conditions - symbolic evaluation techniques
- Solve the path conditions to generate test data that activate the selected paths
- Useful for generating a test suite that covers a given test criterion (all statements, all branches, all defs, all uses, ...)

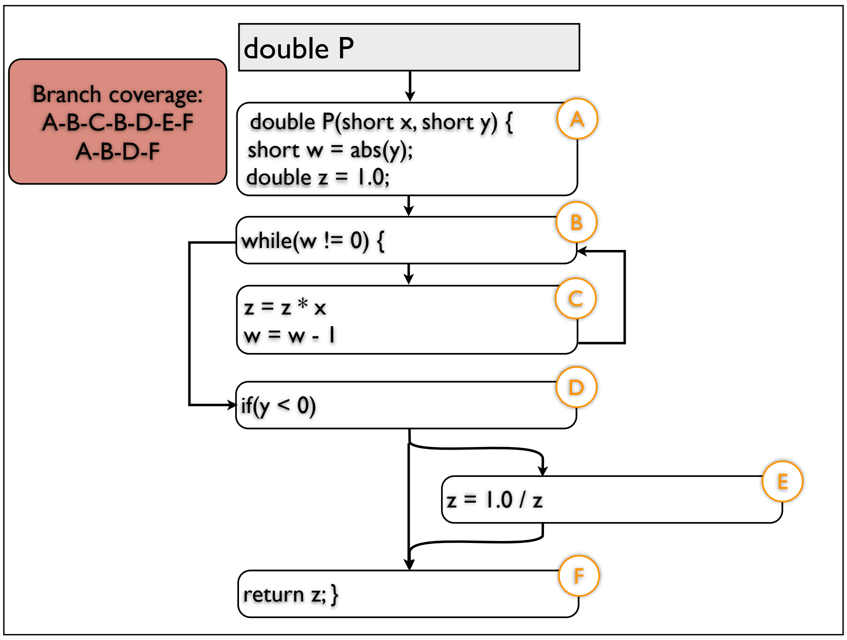
Path Selection

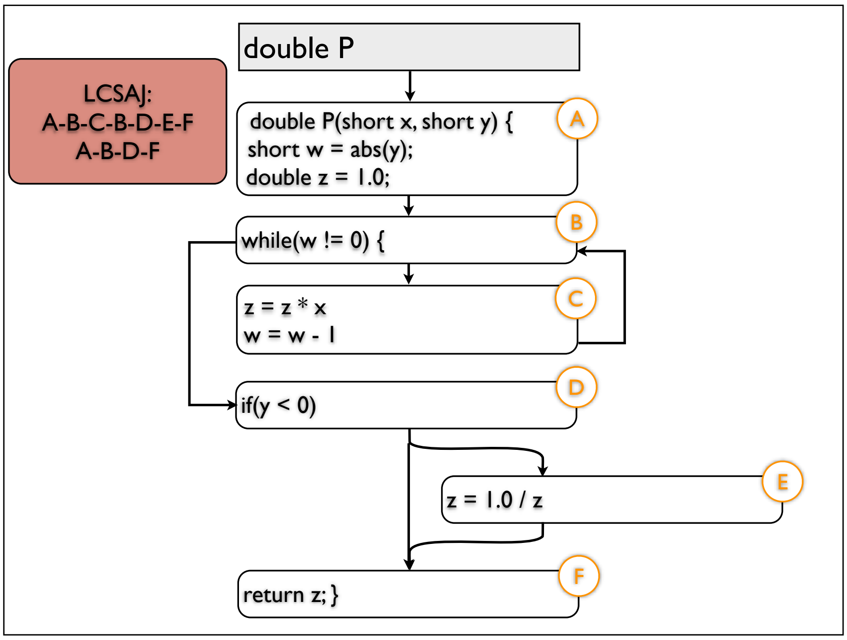
```
double P(short x, short y) {
  short w = abs(y);
  double z = 1.0;
  while(w != 0) {
    z = z * x;
    w = w - 1;
  }
  if(y < 0)
    z = 1.0 / z;

  return z;
}
```

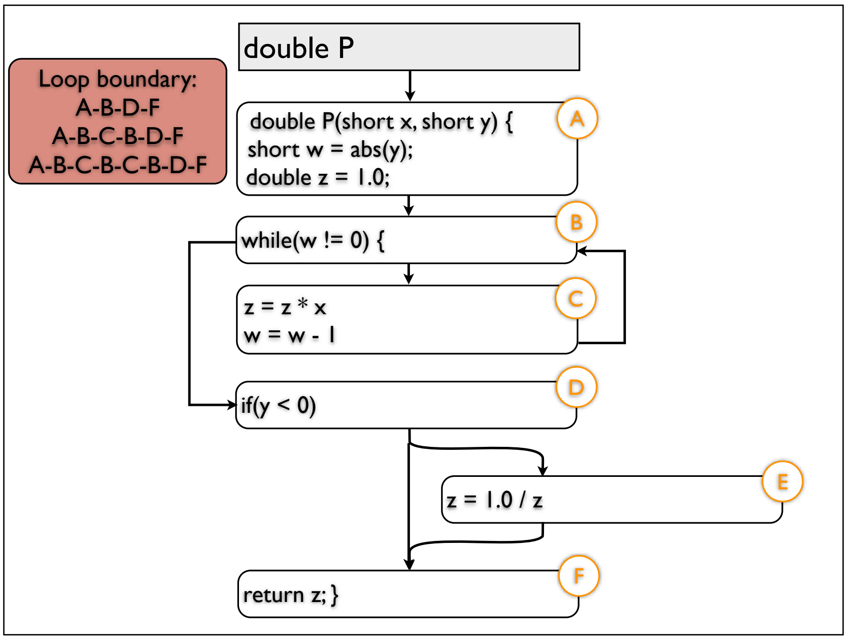
The first step in a path oriented test generation technique is to select which path we want to execute in our test case.

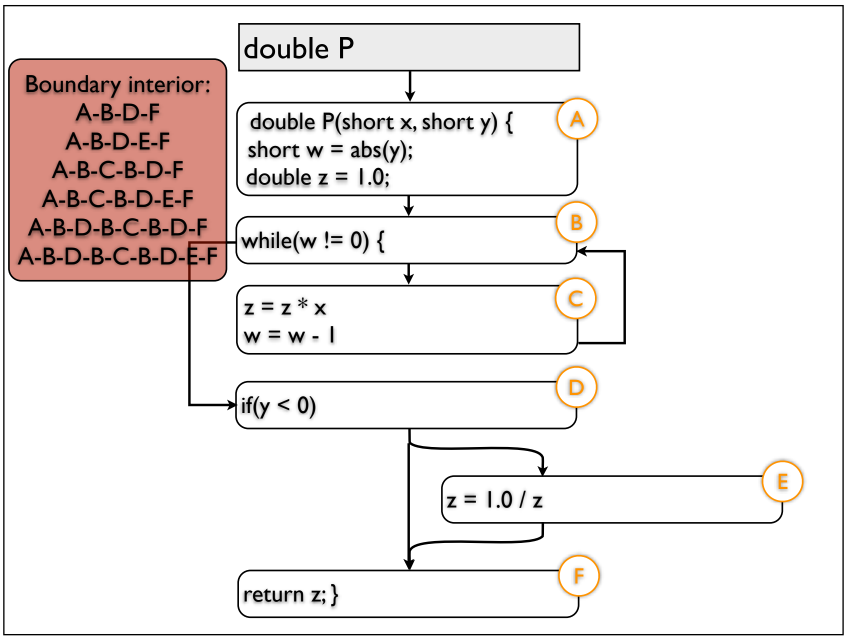

double P
double P(short x, short y) {
short w = abs(y);
double z = 1.0;
while(w != 0) {
z = z * x
w = w - 1
if(y < 0)
z = 1.0 / z
return z; }
A
B
C
D
E
F

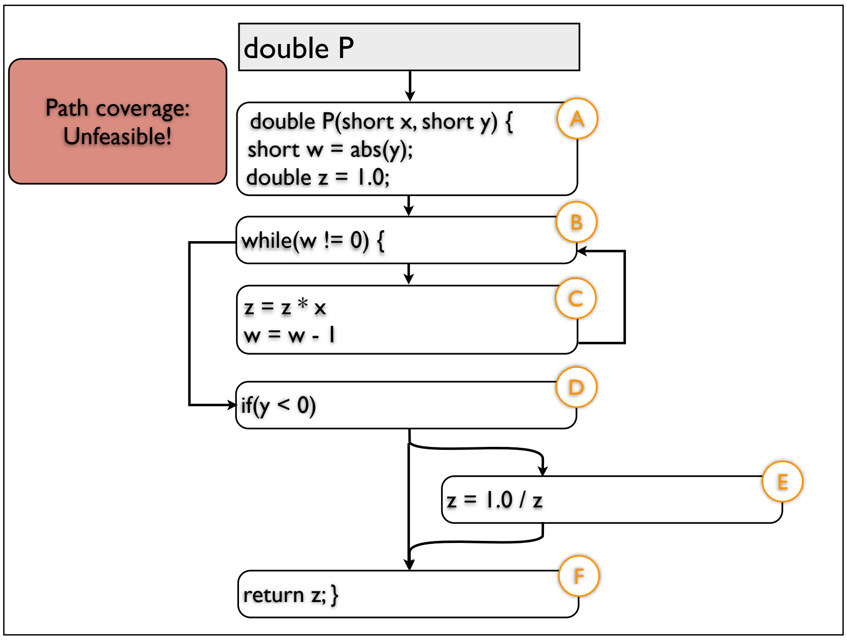

Statement coverage:
A-B-C-B-D-E-F
double P
double P(short x, short y) {
short w = abs(y);
double z = 1.0;
while(w != 0) {
z = z * x
w = w - 1
if(y < 0)
z = 1.0 / z
return z; }
A
B
C
D
E
F

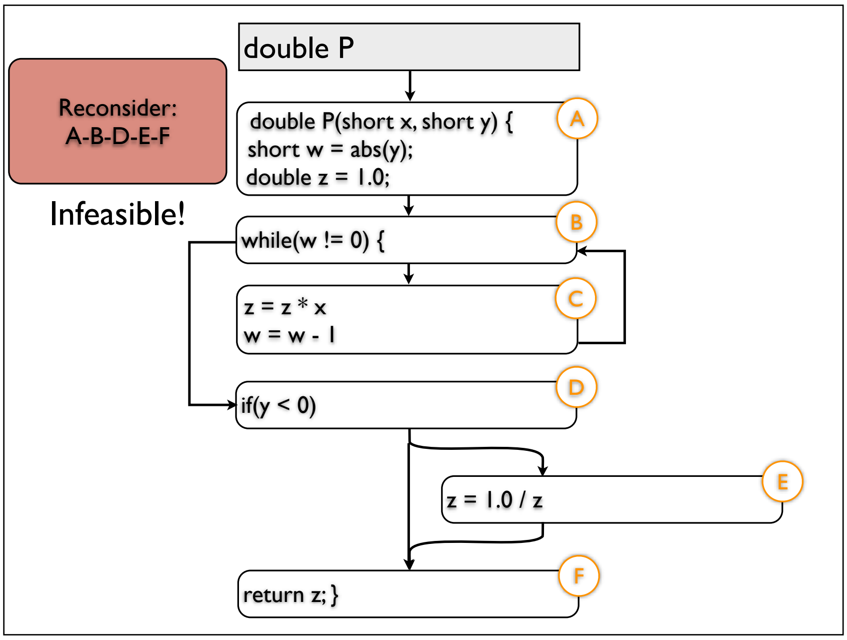

Branch coverage:
A-B-C-B-D-E-F
A-B-D-F
double P
double P(short x, short y) {
short w = abs(y);
double z = 1.0;
while(w != 0) {
z = z * x
w = w - 1
if(y < 0)
z = 1.0 / z
return z; }
A
B
C
D
E
F


LCSAJ:
A-B-C-B-D-E-F
A-B-D-F
double P
double P(short x, short y) {
short w = abs(y);
double z = 1.0;
while(w != 0) {
z = z * x
w = w - 1
if(y < 0)
z = 1.0 / z
return z; }
A
B
C
D
E
F

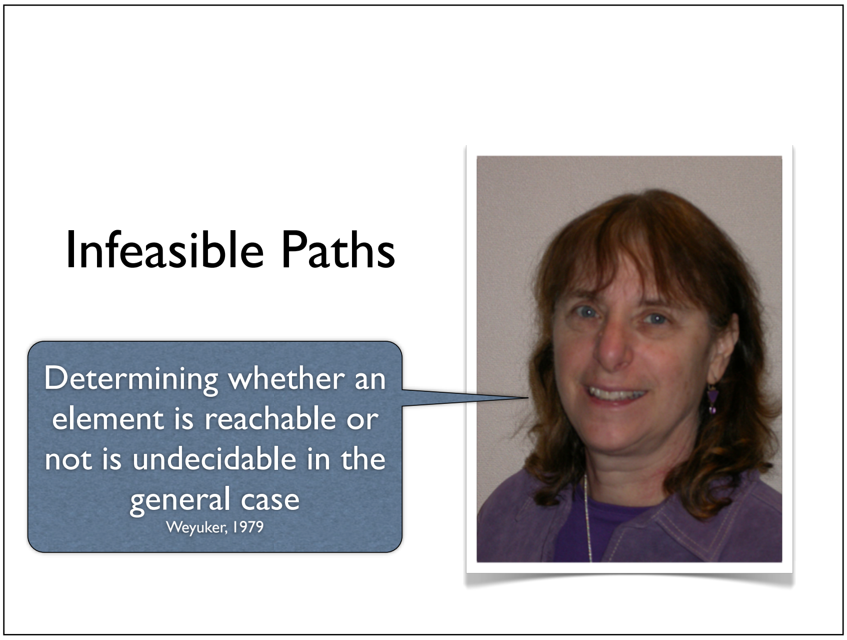

Loop boundary:
A-B-D-F
A-B-C-B-D-F
A-B-C-B-C-B-D-F
double P
double P(short x, short y) {
short w = abs(y);
double z = 1.0;
while(w != 0) {
z = z * x
w = w - 1
if(y < 0)
z = 1.0 / z
return z; }
A
B
C
D
E
F


Boundary interior:
A-B-D-F
A-B-D-E-F
A-B-C-B-D-F
A-B-C-B-D-E-F
A-B-D-B-C-B-D-F
A-B-D-B-C-B-D-E-F
double P
double P(short x, short y) {
short w = abs(y);
double z = 1.0;
while(w != 0) {
z = z * x
w = w - 1
if(y < 0)
z = 1.0 / z
return z; }
A
B
C
D
E
F



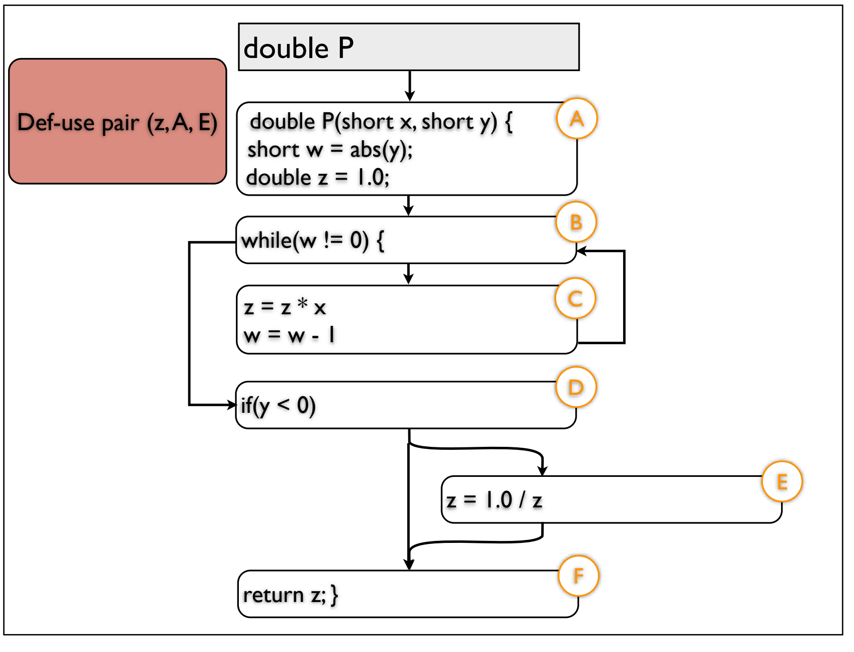


There is no guarantee that a path selected from the CFG is actually executable - paths can be infeasible, in which there is no input that would drive the execution through the chosen path.

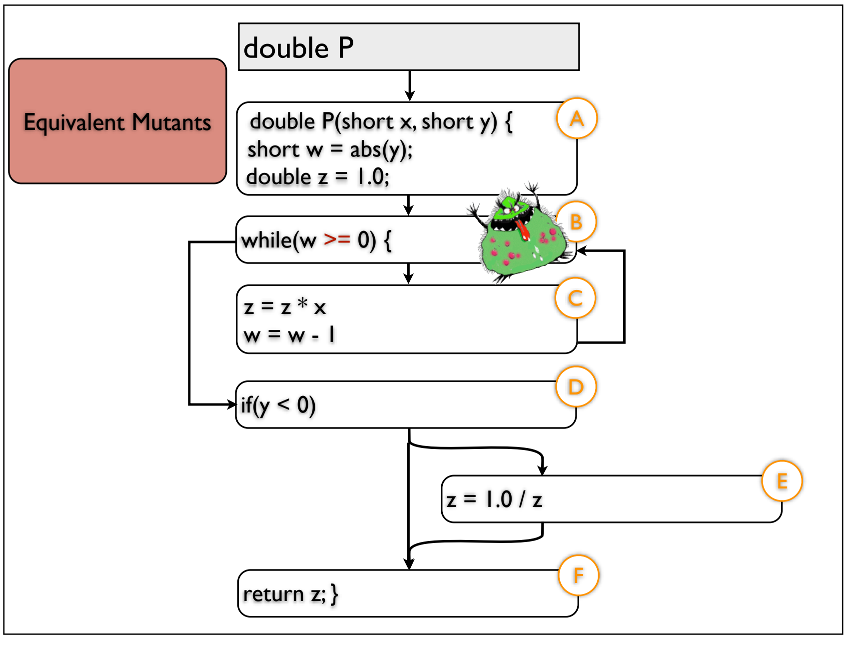


Infeasible Paths

- Determining whether an element is reachable or not is undecidable in the general case
- Infeasible paths are ubiquitous in imperative programs
- Infeasible paths can be selected during the path selection process



We have encountered the infeasible path problem before - infeasible DU pairs are another example instance of the same problem.



The equivalent mutant problem can also be reduced to the infeasible path problem.

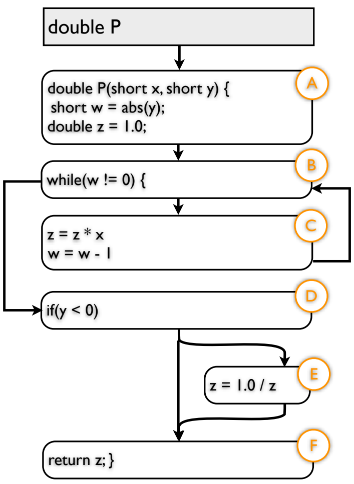
Symbolic Evaluation

- Three path-oriented techniques:
  1. Simple symbolic execution (forward and backward)
  2. Dynamic symbolic execution
  3. Global symbolic execution
- Exploits algebraic expressions over symbolic inputs to represent internal states of variables
- Application in software testing, compiler optimization, specialization, parallel computing, model-checking, program proving, etc.

Simple forward symbolic execution

A-B-C-B-C-B-D-F with X,Y

- A: w := abs(Y); z := 1.0;
- B: *abs(Y) != 0*
- C: z := X; w := abs(Y) - 1;
- B: *abs(Y) - 1 != 0*
- C: z := X * X; w := abs(Y) - 2;
- B: *abs(Y) - 2 = 0*
- D: *Y >= 0*
- F: return (X*X);



Given a path, we can derive constraints by symbolically executing the path either in a forward or backward fashion.

Symbolic State

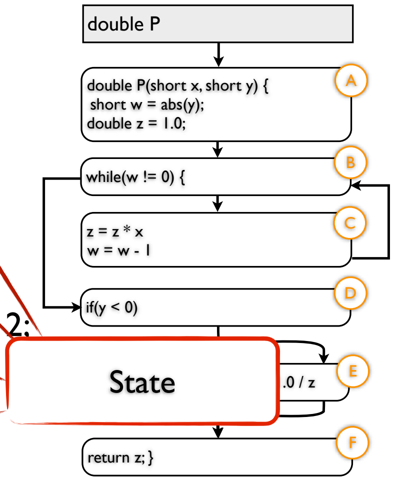
- <Path, State, Path Conditions>
- Path = n_i - ... - n_j is a path of a CFG
- State = $\{<v,\varphi>\}_{v \in Var(P)}$ where φ is an algebraic expression over x
- Path Condition = $c_i \wedge .. \wedge c_n$ where c_k is a condition over x
- x denotes symbolic variables associated to the inputs of program P and Var(P) denotes internal variables

During symbolic execution we maintain a symbolic state of the execution. If we encounter a condition along the execution then the path conditions are updated, if we encounter assignments, then the state expressions are updated.

Simple forward symbolic execution

A-B-C-B-C-B-D-F with X,Y

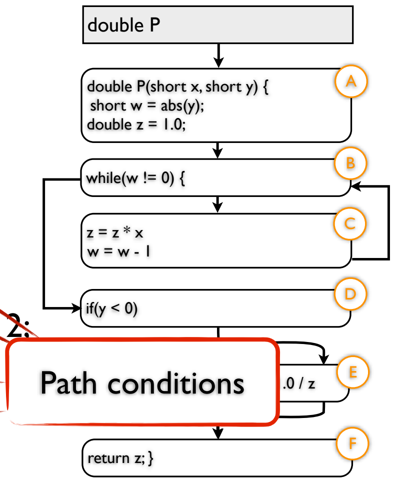
- A: w := abs(Y); z := 1.0;
- B: *abs(Y) != 0*
- C: z := X; w := abs(Y) - 1;
- B: *abs(Y) - 1 != 0*
- C: z := X * X; w := abs(Y) - 2;
- B: *abs(Y) - 2 = 0*
- D: *Y >= 0*
- F: return (X*X);



Simple forward symbolic execution

A-B-C-B-C-B-D-F with X,Y

- A: w := abs(Y); z := 1.0;
- B: *abs(Y) != 0*
- C: z := X; w := abs(Y) - 1;
- B: *abs(Y) - 1 != 0*
- C: z := X * X; w := abs(Y) - 2;
- B: *abs(Y) - 2 = 0*
- D: *Y >= 0*
- F: return (X*X);

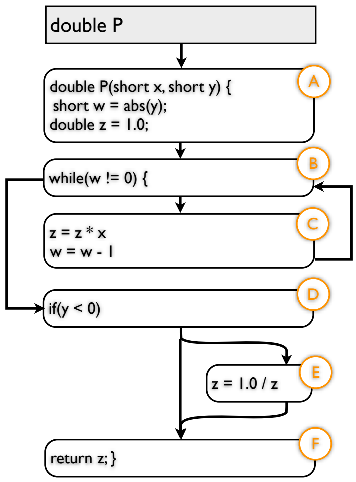


<Path, State, Path Conditions>

<A, {<z,⊥>, <w,⊥>}, true>

<A-B-C-B,
{<z,X>, <w,abs(Y)-1>},
abs(Y) != 0>

<A-B-C-B-C-B-D-F,
{<z,X^2>, <w,abs(Y)-2>},
(abs(Y)!=0) ∧(abs(Y)!=1) ∧(abs(Y)
=2) ∧(Y>=0)>

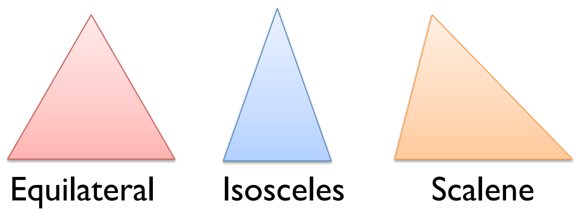


Computing Symbolic States

- <Path,State,PC> is computed by induction over each statement of Path
- When the path conditions are unsatisfiable then Path is infeasible
- Example: <A-B-D-E-F, {...}, abs(Y)=0 && Y<0 >
- Forward vs backward analysis:
  - Forward: Interesting when states are needed
  - Backward: Saves memory space as states remain implicit

Example

Classify triangle by the length of the sides

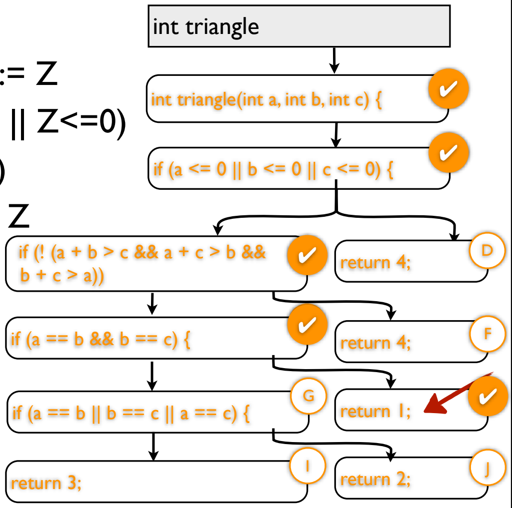



```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

A-B-C-E-H

- A: a :=X; b := Y; c := Z
- B: !(X<=0 || Y<=0 || Z<=0)
- C: (X+Y>Z && ...)
- E: X==Y && Y == Z

Symbolic state at end:
<A-B-C-E-H,
{<a,X>, <b,Y>, <c,Z>},
¬(X<=0 || Y<=0 || Z<=0)∧
(X+Y>Z ∧ ...) ∧ (X==Y ∧
Y == Z)>



The triangle example does not change the state along the execution, so symbolic execution reduces to collecting path conditions.

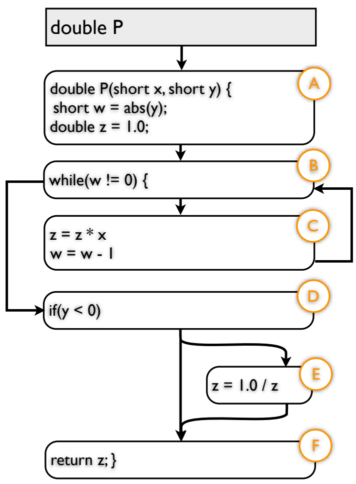
- X>0 && Y>0 && Z>0 && X+Y>Z && X +Z>Y && Y+Z>X && X==Y && Y == Z
- Example, (1,1,1)

Backward Analysis

A-B-C-B-C-B-D-F with X,Y

- F,D: Y >= 0
- B: Y>=0, w=0
- C: Y>=0, w-1=0
- B: Y>=0, w-1=0, w!=0
- C: Y>=0, w-2=0, w-1!=0
- B: Y>=0, w-2=0, w-1!=0, w!=0
- A: Y>=0, abs(Y)-2=0, abs(Y)-1=0, abs(Y)!=0

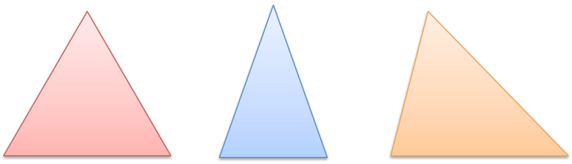
Constraint system



In backward analysis, the state is not explicit as on forward execution (which means this needs less memory). We simply maintain the set of path conditions, and whenever we encounter a state update, we apply this update to the path conditions collected so far.

Example

Classify triangle by the length of the sides

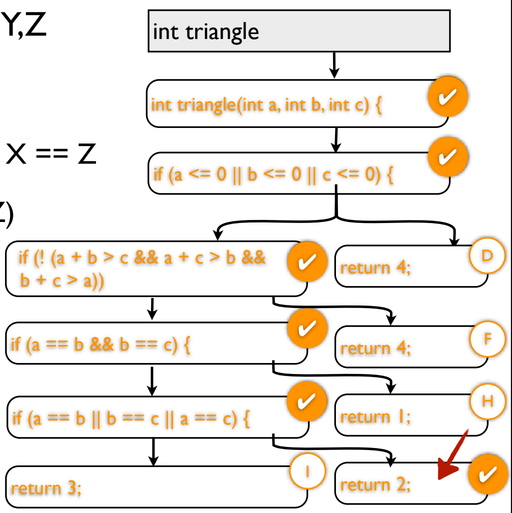


Equilateral Isosceles Scalene

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

A-B-C-E-G-J with X,Y,Z

- J,G: X == Y || Y == Z || X == Z
- E: ... && (X != Y || Y != Z)
- C: ... && (X+Y>Z && X+Z>Y && Y+Z>X)
- B: ... && (X>0 && Y>0 && Z>0)



Implementing Symbolic Execution

- Transformation approach
  Transform to program that operates only on symbolic values
- Instrumentation approach
- Customized runtime approach

Transformation Approach

- Transform the program to another program that operates on symbolic values such that execution of the transformed program is equivalent to symbolic execution of the original program

Instrumentation Approach

- callback hooks are inserted in the program such that symbolic execution is done in background during normal execution of program
- easy to implement for C

Customized runtime approach

- Customize the runtime (e.g. JVM) to support symbolic execution
- Java PathFinder (NASA)
- Applicable to Java, .NET

Limitations

- Limited by the power of constraint solver No non-linear or complex constraints
- Does not scale when number of paths is large
- Source code or equivalent (Bytecode) is required for precise symbolic execution
- Infeasible path problem

Goal-oriented testing

- A three step process:
  1. Generate a constraint model of the whole program
  2. Choose a goal: point to be reached or property to be refuted
  3. Generate test data that respects the model and satisfies the goal
- Useful for generating test data that reaches a single testing objective (reach a statement or a branch, find a counter-example to a property, etc.)

A constraint model of imperative programs

- Viewing an assignment statement as a relation requires to rename the variables
- $i := i + 1 \rightarrow i_2 = i_1 + 1$
- Static Single Assignment (SSA) form or single assignment language

Destructive assignment to variables makes it necessary to rename variables for representation in a logic system.

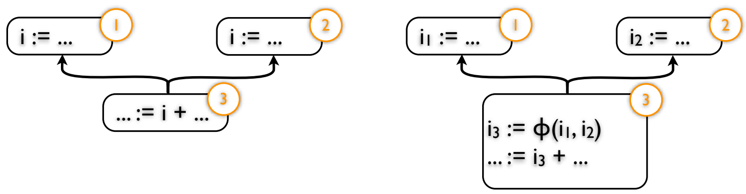
SSA form

- Each use of a variable refers to a single definition

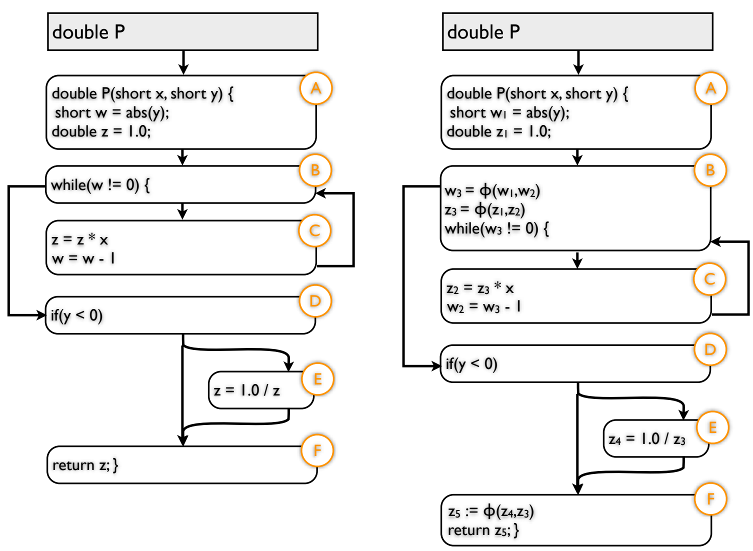
```
x := x + y;
y := x - y;
x := x - y;
```

$x_1 := x_0 + y_0;$
$y_1 := x_1 - y_0;$
$x_2 := x_1 - y_1;$

ϕ Functions



At join points in the control flow, we need to add phi functions that represent a choice of the values of the two branches. In an IF condition the phi function is simply added after the if and else branches, but for loops we need to add the phi function before loop condition.



From SSA to a Constraint System

Variable declaration	Domain constraint
`unsigned int i`	$i \in 0 \ldots 2^{32}-1$

Assignment, decision	Arithmetical constraints {=, <, ...}
$j_2 = j_1 * i$	$j_2 = j_1 * i$
`i == j`	`i = j`
`i < j`	`i < j`

Assignment and comparison have equal operations due to SSA.

From SSA to a Constraint System

Conditional (SSA)

if D then C1 else C2
$v_3 := \phi(v_1, v_2)$

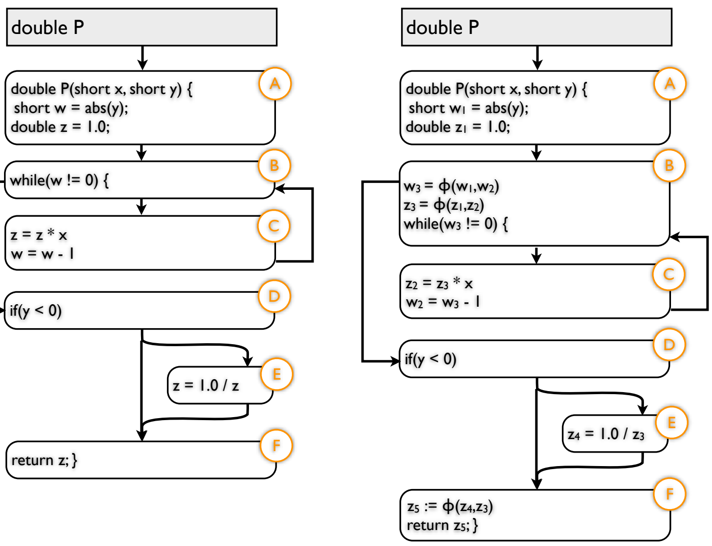
$(D \wedge C1 \wedge v3 = v1) \vee$
$(!D \wedge C2 \wedge v3 = v2)$

Iteration (SSA)

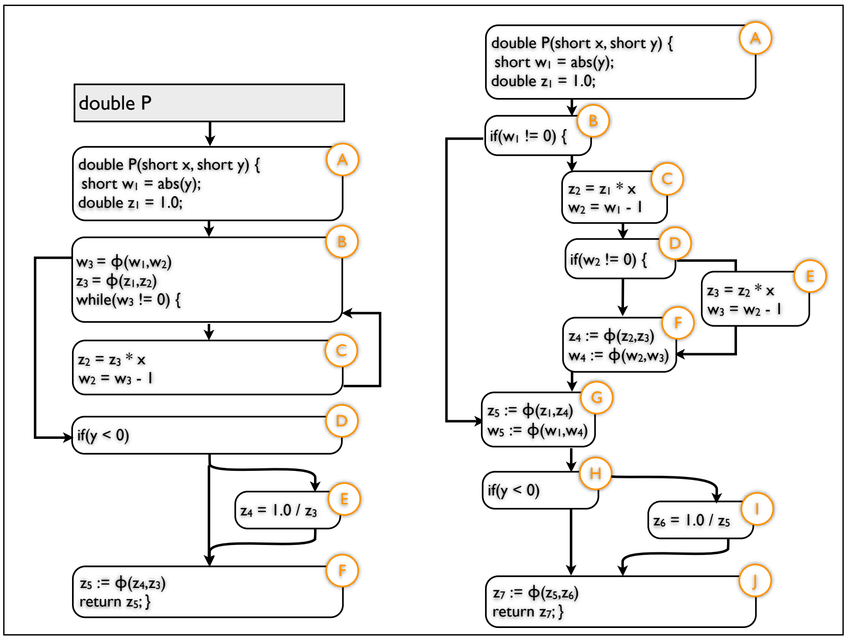
$v_3 := \phi(v_1, v_2)$
while D do C

$v_3 = v_1 \vee (D1 \wedge C1$
$\wedge\ !D2 \wedge v3{=}v2) \vee \ldots$

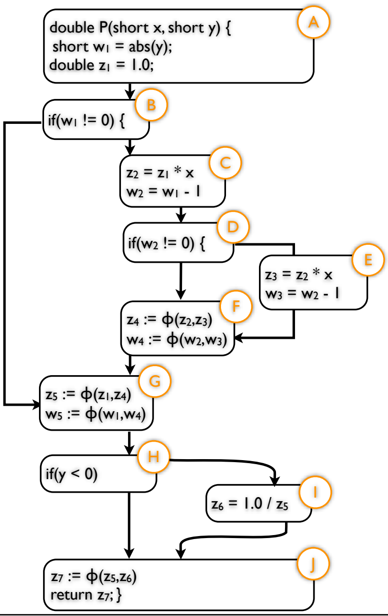
For conditionals and loops we need to get rid of the phi functions when converting to a constraint system. For a conditional this gives us a disjunction of the two possible outcomes of the condition (note that v3 is assigned within this disjunction). Loops need to be unrolled.



Conversion of the power function to SSA.



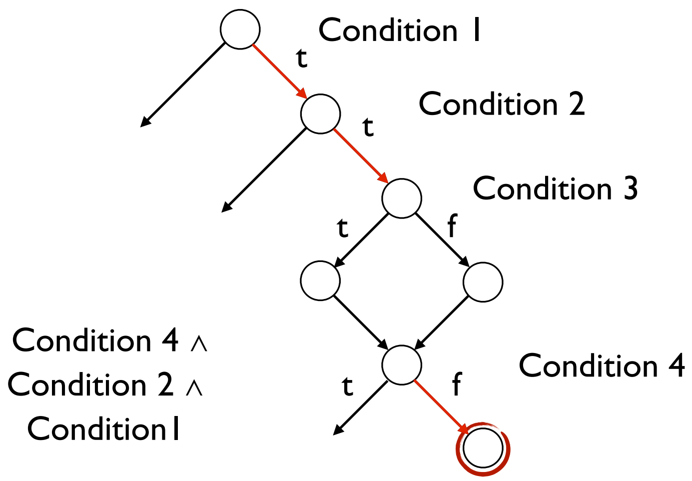
Loop unrolling (0,1, or 2 times)



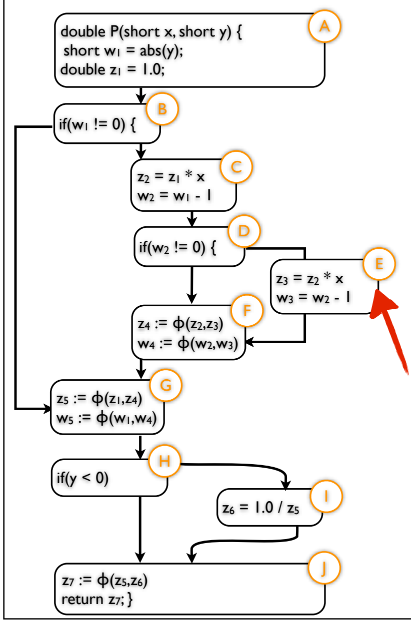
$w_1 = abs(y) \wedge$
$z_1 = 1.0 \wedge$
$z_2 = z_1 * x \wedge$
$w_2 = w_1 - 1 \wedge$
$z_3 = z_2 * x \wedge$
$w_3 = w_2 - 1 \wedge$
$((w_2 = 0 \wedge w_4 = w_2 \wedge z_4 = z_2) \vee$
$(w_2 \neq 0 \wedge w_4 = w_3 \wedge z_4 = z_3)) \wedge$
$((w_1 = 0 \wedge z_5 = z_1 \wedge w_5 = w_1) \vee$
$(w_1 \neq 0 \wedge z_5 = z_4 \wedge w_5 = w_4)) \wedge$
$z_6 = 1.0 / z_5 \wedge$
$((y >= 0 \wedge z_7 = z_5) \vee (y < 0 \wedge z_6))$

Resulting constraint system.

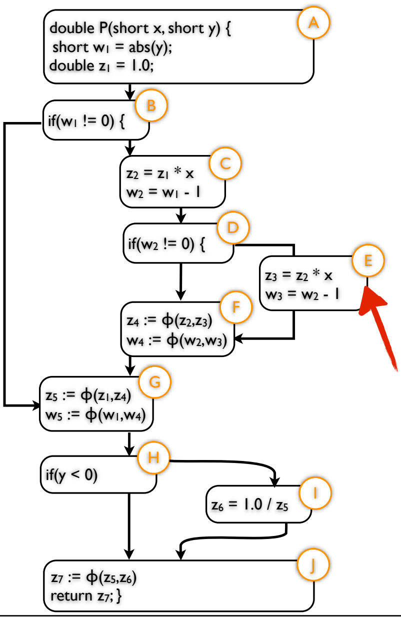
Test Objectives



To reach a certain point in the control flow the control dependencies need to be satisfied. For the constraint system, we add all the control dependent conditions.

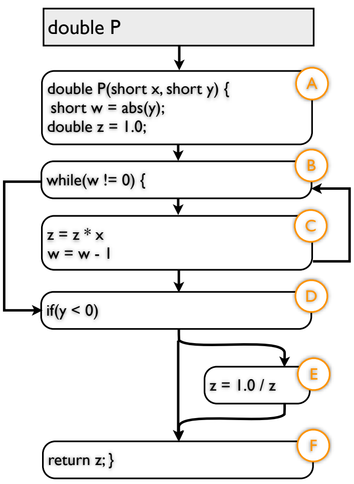


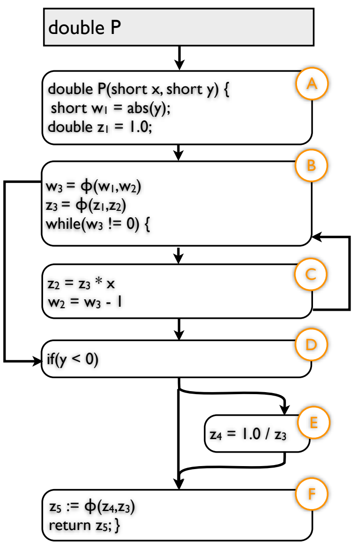
$w_1 != 0 \land w_2 != 0$



$w_1 = abs(y) \land$
$z_1 = 1.0 \land$
$z_2 = z_1 * x \land$
$w_2 = w_1 - 1 \land$
$z_3 = z_2 * x \land$
$w_3 = w_2 - 1 \land$
$((w_4 = w_2 \land z_4 = z_2 \land w_2 = 0) \lor$
$(w_4 = w_3 \land z_4 = z_3 \land w_2 != 0)) \land$
$((w_1 = 0 \land z_5 = z_1 \land w_5 = w_1) \lor$
$(w_1 != 0 \land z_5 = z_4 \land w_5 = w_4)) \land$
$z_6 = 1.0 / z_5 \land$
$((y >= 0 \land z_7 = z_5) \lor (y < 0 \land z_7 =$
$z_6)) \land$

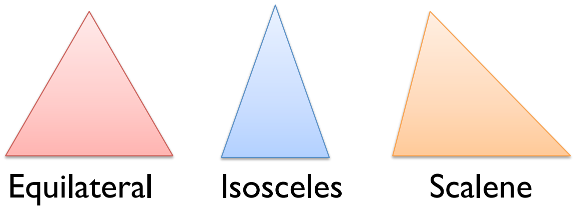
$w_1 != 0 \land w_2 != 0$





Example

Classify triangle by the length of the sides




```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;   // scalene
}
```

```
int triangle(int a, int b, int c) {
    int r = 3; // scalene

    if (a <= 0 || b <= 0 || c <= 0) {
      r = 4; // invalid
    } else {
      if (!(a + b > c && a + c > b && b + c > a)) {
        r = 4; // invalid
      } else {
        if (a == b && b == c) {
          r = 1; // equilateral
        } else {
          if (a == b || b == c || a == c) {
            r = 2; // isosceles
          }
        }
      }
    }
    return r;
}
```

```
int triangle(int a, int b, int c) {
    int r = 3; // scalene

    if (a <= 0 || b <= 0 || c <= 0) {
      r = 4; // invalid
    }

    if (!(a + b > c && a + c > b && b + c > a)) {
        r = 4; // invalid
    }

    if (a == b && b == c) {
       r = 1; // equilateral
    } else if (a == b || b == c || a == c) {
       r = 2; // isosceles
    }
    return r;
}
```

```
int triangle(int a, int b, int c) {
    int r0 = 3; // scalene

    if (a <= 0 || b <= 0 || c <= 0) {
      r1 = 4; // invalid
    } else {
      if (!(a + b > c && a + c > b && b + c > a)) {
        r2 = 4; // invalid
      } else {
        if (a == b && b == c) {
          r3 = 1; // equilateral
        } else {
          if (a == b || b == c || a == c) {
            r4 = 2; // isosceles
          }
          r5 = ϕ(r4, r0);
        }
        r6 = ϕ(r5, r3);
      }
      r7 = ϕ(r6, r2);
    }
    r8 = ϕ(r7, r1);
    return r8;
}
```

```
int triangle(int a, int b, int c) {
    int r0 = 3; // scalene

    if (a <= 0 || b <= 0 || c <= 0) {
      r1 = 4; // invalid
    } else {
      if (!(a + b > c && a + c > b && b + c > a)) {
        r2 = 4; // invalid
      } else {
        if (a == b && b == c) {
          r3 = 1; // equilateral
        } else {
          if (a == b || b == c || a == c) {
            r4 = 2; // isosceles
          }
          r5 = ϕ(r4, r0);
        }
        r6 = ϕ(r5, r3);
      }
      r7 = ϕ(r6, r2);
    }
    r8 = ϕ(r7, r1);
    return r8;
}
```

$(r0 = 3) \wedge$

$((r1 = 4 \wedge r8 = r1 \wedge$
$(a <= 0 \vee b <= 0 \vee c <= 0)) \vee$
$(r8 = r7 \wedge$
$!(a <= 0 \vee b <= 0 \vee c <= 0))) \wedge$

$((r2 = 4 \wedge r7 = r2 \wedge$
$(a+b<=c \vee a+c<=b \vee b+c<=a)) \vee$
$(r7 = r6 \wedge$
$!(a+b<=c \vee a+c<=b \vee b+c<=a))) \wedge$

$((r3 = 1 \wedge r6 = r3 \wedge a = b \wedge b = c) \vee$
$(r6 = r5 \wedge !(a = b \wedge b = c))) \wedge$

$((r4 = 2 \wedge r5 = r4 \wedge (a=b \vee b=c \vee a=c)) \vee$
$(r5 = r0 \wedge !(a=b \vee b=c \vee a=c)))$

```
int triangle(int a, int b, int c) {
    int r0 = 3; // scalene

    if (a <= 0 || b <= 0 || c <= 0) {
      r1 = 4; // invalid
    } else {
      if (!(a + b > c && a + c > b && b + c > a)) {
        r2 = 4; // invalid
      } else {
        if (a == b && b == c) {
          r3 = 1; // equilateral
        } else {
          if (a == b || b == c || a == c) {
            r4 = 2; // isosceles
          }
          r5 = ϕ(r4, r0);
        }
        r6 = ϕ(r5, r3);
      }
      r7 = ϕ(r6, r2);
    }
    r8 = ϕ(r7, r1);
    return r8;
}
```