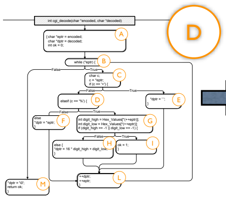
Constraint-based Testing

Program

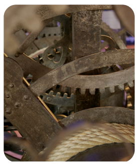
Test Goal



Constraint system

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

Constraint solver



Test data

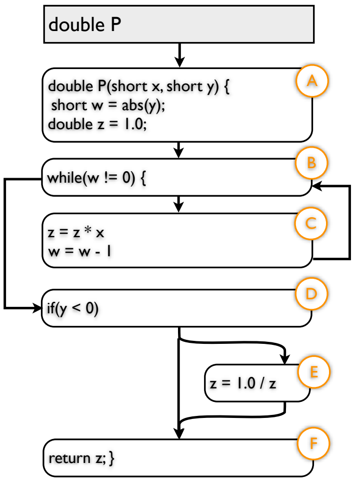
a = 1
b = 2
c = 3

In constraint-based testing, we represent the program under test and our test goal as a constraint system. Any solution to this constraint system is a valid test input that will satisfy the test goal. To derive a solution we can leverage powerful constraint solvers. There are two main approaches to constraint based testing: Path-based and goal based testing.

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);



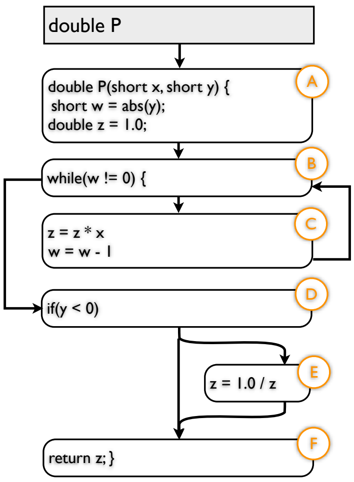
In path based testing, we select a path from the control flow graph and then use symbolic execution to derive constraints that represent this path. Symbolic execution can be done in a forward fashion, where at each execution step the symbolic state is updated according to the encountered expressions and conditions. The feasible path problem can be overcome by using dynamic symbolic execution, where we only follow paths that are reached by real executions.

Symbolic states:
<Path, State, Path Conditions>

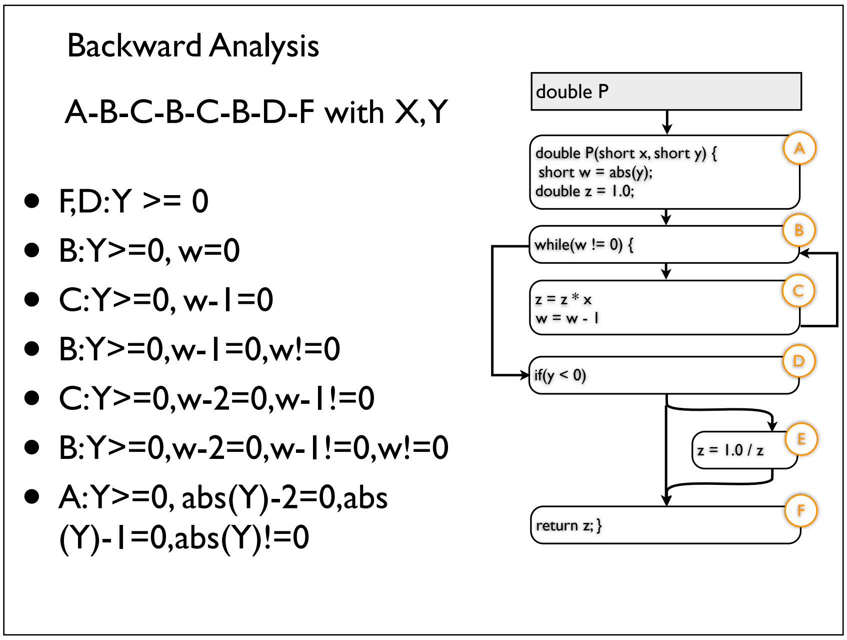
<A, {<z,⊥>, <w,⊥>}, true>

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

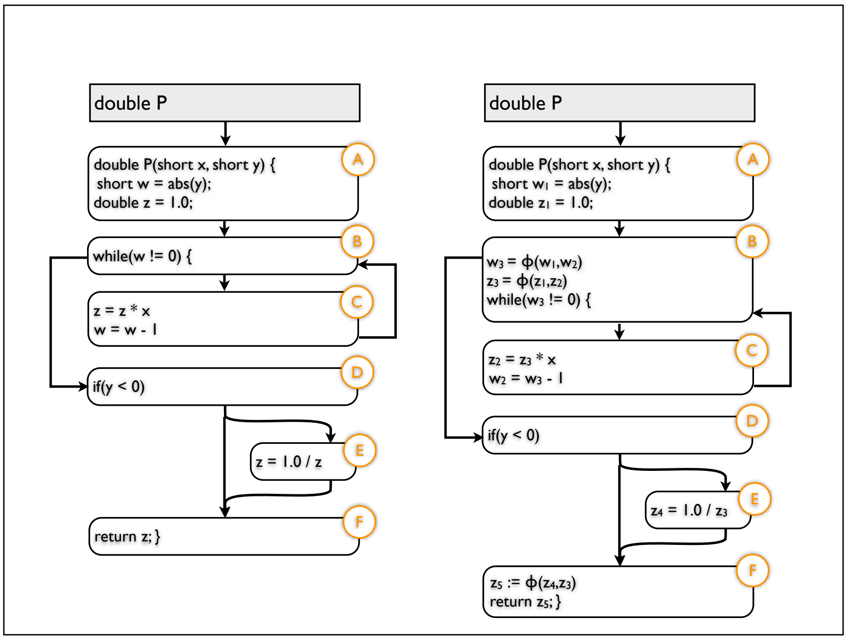
<A-B-C-B-C-B-D-F,
{<z,X^2>, <w,abs(Y)-2>},
(abs(Y)!=0) ∧(abs(Y)!=1) ∧
(abs(Y) =2) ∧(Y>=0)
>



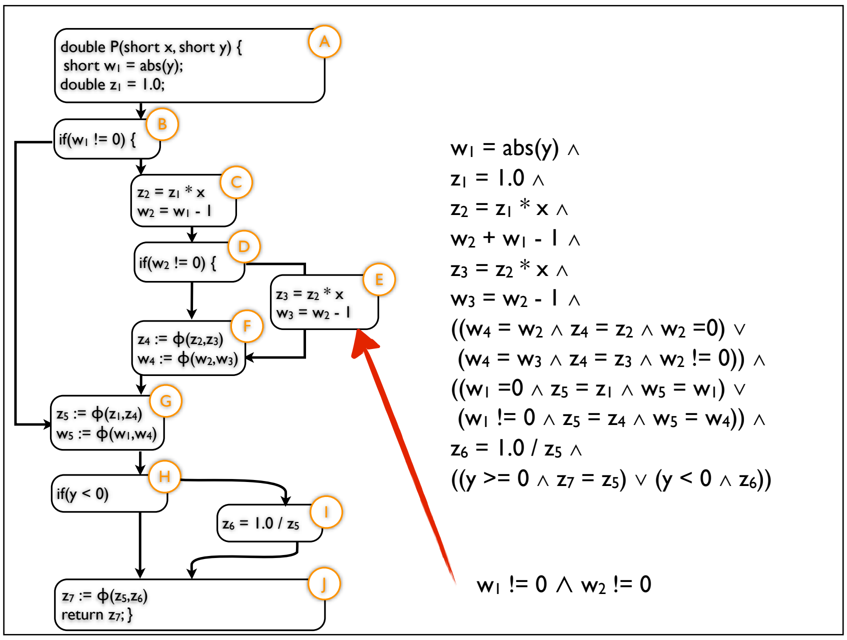
At the end of symbolic execution, the last symbolic state contains the set of constraints that need to be true for this path to be taken. By solving this constraint system, we generate test data for the chosen path. (In this case: (abs(Y)!=0) ∧ (abs(Y)!=1) ∧(abs(Y) =2) ∧(Y>=0)



The alternative to forward execution is backward analysis, where we again collect conditions, but this time rewrite these conditions according to the expressions we encounter. This means that the exact symbolic state is not known along the path, but therefore backward analysis is computationally cheaper.



For goal oriented testing, we first convert a program to Static Single Assignment form.



Then, from the SSA form and after unrolling loops, we can derive a constraint system. To this constraint system we can add a constraint that describes the test goal (e.g., the conjunction of the control dependencies) of the target branch.