Symbolic Reasoning

Mooly Sagiv

Slides from Koushick, Sen, Zvonimir Rakamaric

First-Order Logic

- A formal notation for mathematics, with expressions involving
  - Propositional symbols
  - Predicates
  - Functions and constant symbols
  - Quantifiers
- In contrast, propositional (Boolean) logic only involves propositional symbols and operators

First-Order Logic: Syntax

- As with propositional logic, expressions in first-order logic are made up of sequences of symbols
- Symbols are divided into *logical symbols* and *non-logical symbols* or *parameters*
- Example:

$$(x = y) \land (y = z) \land (f(z) \rightarrow f(x)+1)$$

First-Order Logic: Syntax

- Logical Symbols
  - Propositional connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$,…
  - Variables: V1, V2, . . .
  - Quantifiers: $\exists$, $\forall$
- Non-logical symbols/Parameters
  - Equality: =
  - Functions: +, -, %, bit-wise &, f(), concat, …
  - Predicates: $\leq$, is_substring, …
  - Constant symbols: 0, 1.0, null, …

Example

$\forall X{:}S.\ \ p(f(X),X) \Rightarrow \exists Y{:}S\ .\ p(f(g(X,Y)),g(X,Y)$

Quantifier-free Subset

- We will largely restrict ourselves to formulas without quantifiers (∀, ∃)
- This is called the quantifier-free subset/fragment of first-order logic with the relevant theory

Logical Theory

- Defines a set of parameters (non-logical symbols) and their meanings
- This definition is called a *signature*
- Example of a signature:

  Theory of linear arithmetic over integers

  Signature is (0,1,+,-,·) interpreted over Z

Presbruger Arithmetic

- Signature (0, 1, +, =) interpreted over Z
- Axioms
  - $\forall$X: Z. $\neg$ ((X+1) = 0)
  - $\forall$X, Y: Z. (X+1) = (Y+1) $\Rightarrow$ X + Y
  - $\forall$X: Z. X+0 = X
  - $\forall$X, Y: Z. X+(Y+1) = (X+ Y)+1
  - Let P(X) be a first order formula over X
    - (P(0) $\wedge$ $\forall$X: Z. P(X) $\Rightarrow$P(X+1)) $\Rightarrow$ $\forall$Y: Z. P(Y)

Many Sorted First Order Vocabulary

- A finite set of sorts S
- A finite set of function symbols F each with a fixed signature $S^* \to S$
  - Zero arity functions are constant symbols
- A finite set of relation symbols R each with a fixed arity S^*

An Interpretation ι

- A domain D_s for every $s \in S$
  - $D = \cup_{s \in S:}$ Ds
- For every function symbol $f \in F$, an interpretation $\iota[f]: D_{s1} \times D_{s2} \times \ldots \times D_{sn} \rightarrow$ Ds
- For every relation symbol $r \in R$, an interpretation

  $\iota[r] \subseteq D_{s1} \times D_{s2} \times \ldots \times D_{sm}$

Many-Sorted First Oder Formulas

- Logical Variables
  - Begin with Capital variables
- Typed Terms

  <term> ::= <variable> | f [(<term>, … <term>)]
- Formulas

  <form> ::= <term> = <term> | r(<term>, … <term>) // atomic

  <form> $\vee$ <form> | <form> $\wedge$<form> | $\neg$ <form> // Boolean

  $\exists$X: s <form> | $\forall$ X : s. <form> // Quantifications

Free Variables

- FV: <term>, <formula> $\rightarrow 2^{Var}$
- Terms
  - $FV(X) = \{X\}$
  - $FV(f(<t_1, t_2, \ldots, t_n)) = \cup_{i=1..n:} FV(t_i)$
- Formulas
  - $FV(t1 = t2)= FV(t1) \cup FV(t2)$
  - $FV(r(<t_1, t_2, \ldots, t_n)) = \cup_{i \in 1..n} FV(t_i)$
  - $FV(f1 \vee, \wedge f2) = FV(f_1) \cup FV(f_2)$
  - $FV(\neg f2) = FV(f)$
  - $FV(\exists X:s.f) = FV(f) – \{X\}$
  - $FV(\forall X:s. f) = FV(f) – \{X\}$

Assignments and Models

- Assignment A: Var $\rightarrow$ D
- Extended to terms
  - $A(f(t_1, t_2, \ldots, t_n) = \iota[f](A(t_1), A(t_2), \ldots, A(t_n))$
- An assignment A models a formula f under interpretation ι (denoted by $A, \iota \models f$) if f is true in A (Tarsky's semantics)
- $A, \iota \models t_1 = t_2$ if $A(t_1) = A(t_2)$
- $A, \iota \models r(t_1, t_2, \ldots, t_n)$ if $<A(t_1), A(t_2), \ldots, A(t_n)> \in \iota[r]$
- $A, \iota \models f_1 \vee f_2$ if $A, \iota \models f_1$ or $A, \iota \models f_2$
- $A, \iota \models \neg f$ if not $A, \iota \models f$
- $A, \iota \models \exists X\colon t.\ f$ if there exists $d \in D_t$ such that $A[X \mapsto d]$ if $A, \iota \models f$

A T-Interpretation

- A domain D_s for every $s \in S$
  - $D = \cup_{s\in S:}$ Ds
- For every function symbol $f \in F$, an interpretation ι [f]: $D_{s1} \times D_{s2} \times \ldots \times D_{sn} \rightarrow Ds$
- For every relation symbol $r \in R$, an interpretation ι [r] $\subseteq D_{s1} \times D_{s2} \times \ldots \times D_{sm}$
- The domain and the interpretations satisfy the theory requirements(axioms)

Example Linear Arithmetic

- S ={int}, F ={$\mathbf{0}^0$, $\mathbf{1}^1$, $+^2$}, r = {$\leq^2$}
- Domain
  - $D_{int} = Z$
- Functions
  - $[\![\mathbf{0}]\!] = 0$
  - $[\![\mathbf{1}]\!] = 1$
  - $[\![+]\!] = \lambda x, y\text{: int. } x + y$
- Relations
  - $[\![\leq]\!] = \lambda x, y\text{: int. } x \leq y$

Assignments and T-Models

- Assignment A: Var → D
- Extended to terms
  - $A(f(t_1, t_2, \ldots, t_n) = \iota[f](A(t_1), A(t_2), \ldots, A(t_n))$
- An assignment A which models a theory T, T-models a formula f under interpretation ι (denoted by $A, \iota \models_T f$) if f is true in A (Tarsky's semantics)
- $A, \iota \models_T t_1 = t_2$ if $A(t_1) = A(t_2)$
- $A, \iota \models_T r(t_1, t_2, \ldots, t_n)$ if $<A(t_1), A(t_2), \ldots, A(t_n)> \in \iota[r]$
- $A, \iota \models_T f_1 \vee f_2$ if $A, \iota \models_T f_1$ or $A, \iota \models_T f_2$
- $A, \iota \models_T \neg f$ if not $A, \iota \models_T f$
- $A, \iota \models_T \exists X\colon t.\ f$ if there exists $d \in D_t$ such that $A[X \mapsto d]$ if $A, \iota \models_T f$

The SMT decision problem

- Input: A quantifier-free formula f over a theory T
- Does there exist an T-interpretation ι and an assignment A:FV(f) $\rightarrow$ D such that A $\models_T$ f
- The complexity depends on the complexity of the theory solvers
  - NPC-Undecidable

Summary of Decidability Results

Theory		Quantifiers Decidable	QFF Decidable
T_E	Equality	NO	YES
T_{PA}	Peano Arithmetic	NO	NO
$T_{\mathbb{N}}$	Presburger Arithmetic	YES	YES
$T_{\mathbb{Z}}$	Linear Integer Arithmetic	YES	YES
$T_{\mathbb{R}}$	Real Arithmetic	YES	YES
$T_{\mathbb{Q}}$	Linear Rationals	YES	YES
T_A	Arrays	NO	YES

Summary of Complexity Results

Theory		Quantifiers	QF Conjunctive
PL	Propositional Logic	NP-complete	O(n)
T_E	Equality	–	O(n log n)
$T_{\mathbb{N}}$	Presburger Arithmetic	O(2^2^2^(kn))	NP-complete
$T_{\mathbb{Z}}$	Linear Integer Arithmetic	O(2^2^2^(kn))	NP-complete
$T_{\mathbb{R}}$	Real Arithmetic	O(2^2^(kn))	O(2^2^(kn))
$T_{\mathbb{Q}}$	Linear Rationals	O(2^2^(kn))	PTIME
T_A	Arrays	–	NP-complete

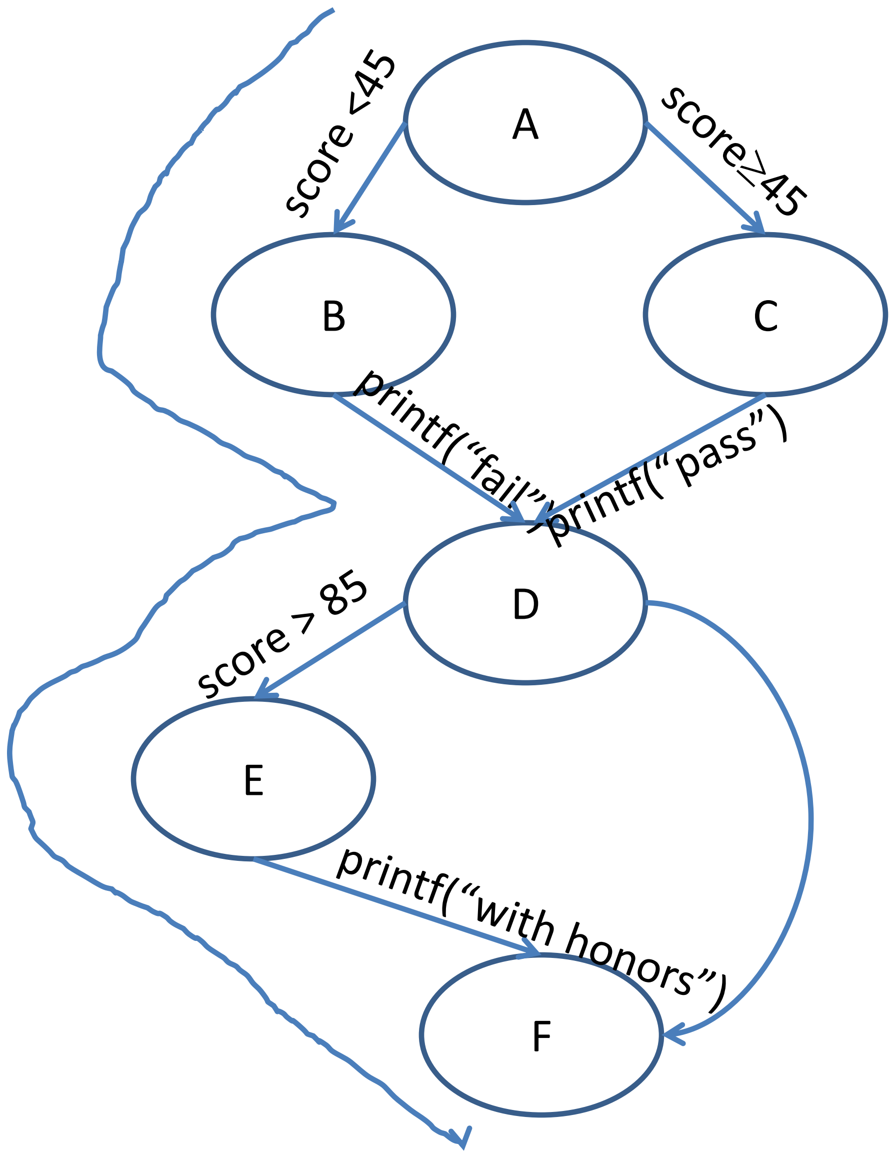
n – input formula size; k – some positive integer

Program Path

- Program Path
  - A path in the control flow of the program
    - Can start and end at any point
    - Appropriate for imperative programs
- Feasible program path
  - There exists an input that leads to the execution of this path
- Infeasible program path
  - No input that leads to the execution

Infeasible Paths

```
void grade(int score) {
A:  if (score <45) {
B:    printf("fail");
    }
    else
 C:    printf("pass");
    }
 D:  if (score > 85) {
 E:     printf("with honors");
        }
F:
}
```



Concrete vs. Symbolic Executions

- Real programs have many infeasible paths
  - Ineffective concrete testing
- Symbolic execution aims to find rare errors

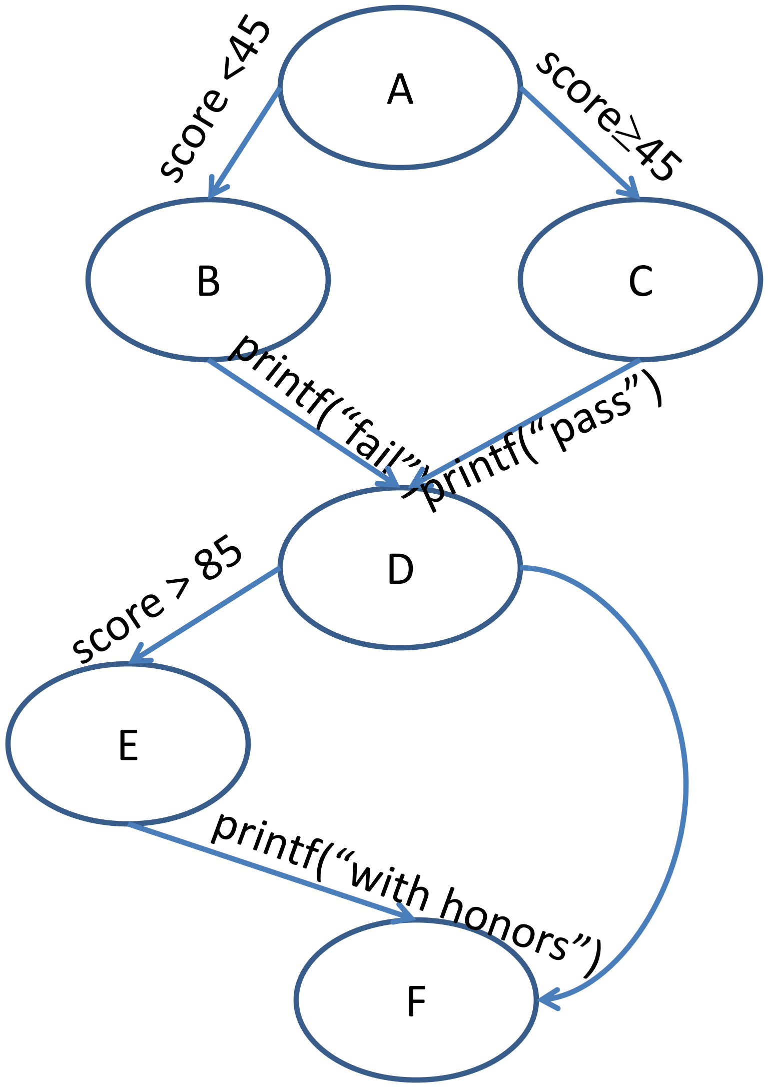
Symbolic Testing Tools

- EFFIGY [King, IBM 76]
- PEX [MSR]
- SAGE [MSR]
- SATURN[Stanford]
- KLEE[Stanford]
- Java pathfinder[NASA]
- Bitscope [Berkeley]
- Cute [UIUC, Berkeley]
- Calysto [UBC]

Finding Infeasible Paths Via SMT

```
void grade(int score) {
A:  if (score <45) {
B:    printf(“fail”);
    }
    else
 C:    printf(“pass”);
    }
 D:  if (score > 85) {
 E:     printf(“with honors”);
        }
F:
}
```

$\boxed{\text{score} < 45 \wedge \text{score} > 85}$ UNSAT

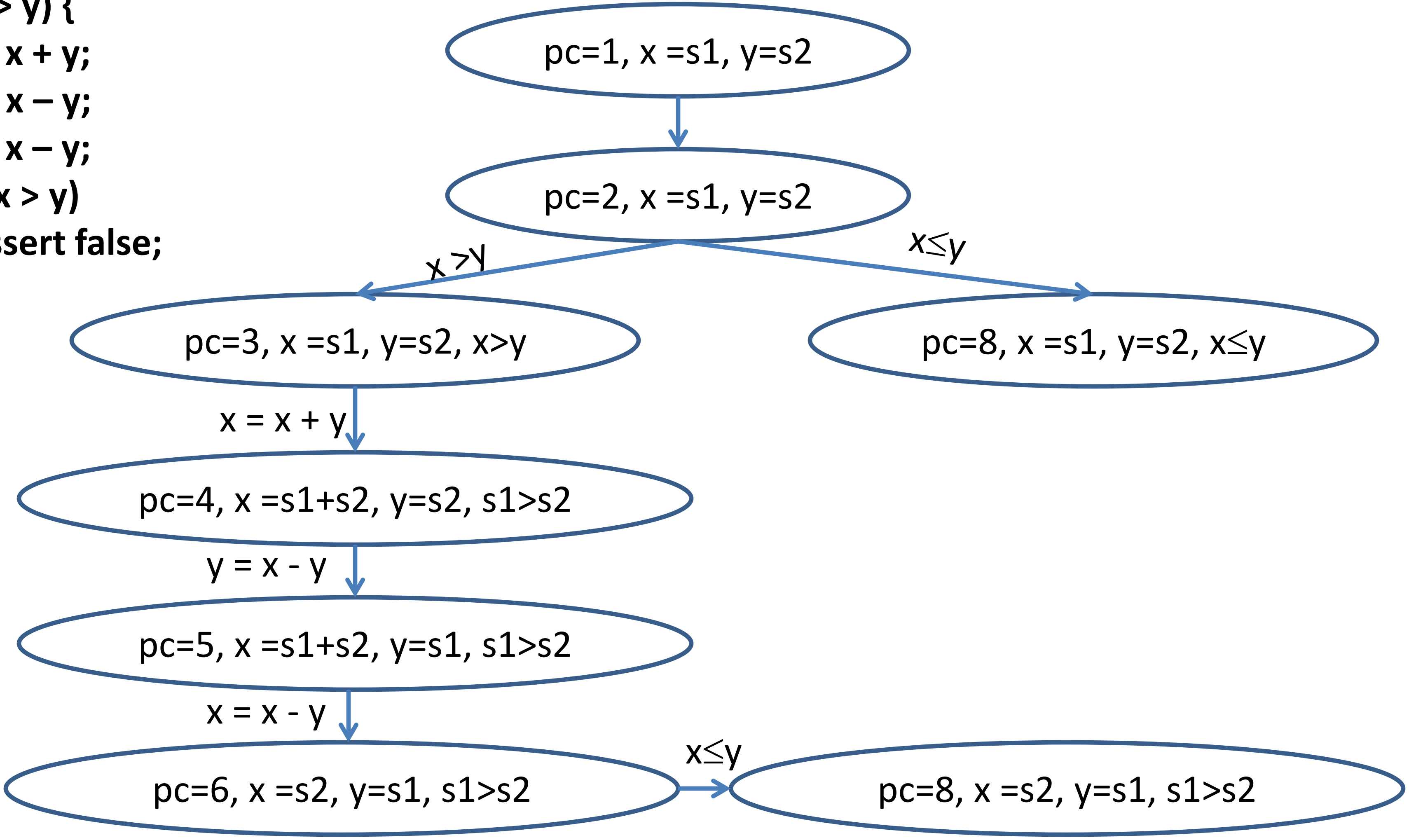


Symbolic Execution Tree

- The constructed symbolic execution paths
- Nodes
  - Symbolic Program States
- Edges
  - Potential Transitions

Simple Example

```
1)int x, y;
2)if (x > y) {
  3) x = x + y;
  4) y = x – y;
  5) x = x – y;
  6) if (x > y)
    7) assert false;
8)}
```



Issues in Symbolic Executions

- PL features
  - Loops
  - Procedures
  - Pointers
  - Data structures
  - Libraries
  - Concurrency
  - Large code base
  - Tricky semantics
- Limitations of Solvers
  - Quantifications
  - Interesting theories
- Scalability

Concolic Testing

- Combine concrete testing (concrete execution) and symbolic testing (symbolic execution)

  [PLDI'05, FSE'05, FASE'06, CAV'06, HVC'06]

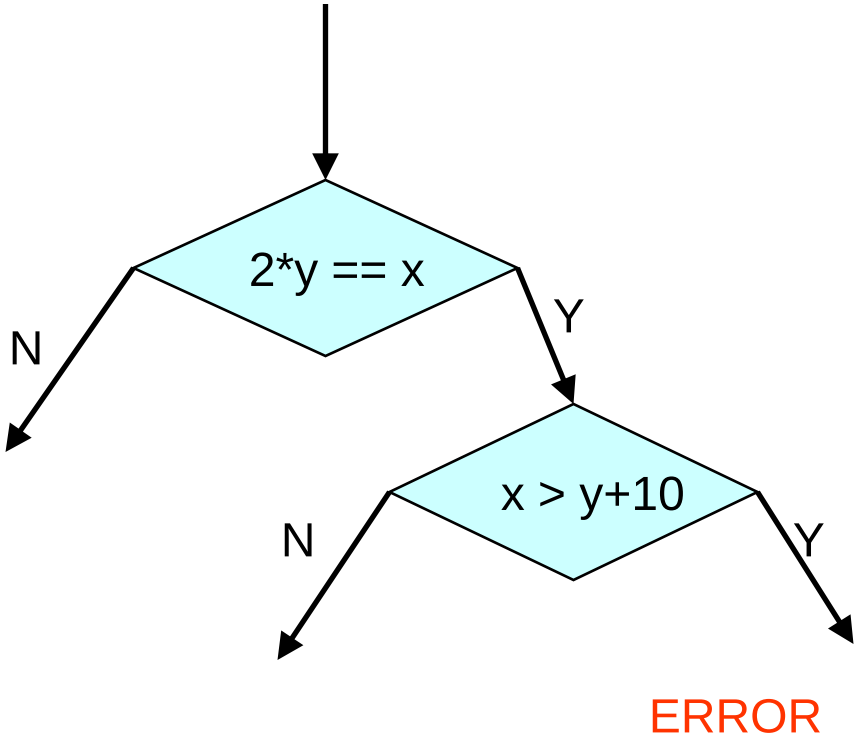
Concrete + Symb**olic** = Concolic

Example

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

Example

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```



Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
$x = 22, y = 7$	$x = x_0, y = y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 22, y = 7, z = 14	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

     return 2*v;
}

void testme (int x, int y) {

     z = double (y);

     if (z == x) {

          if (x > y+10) {

               ERROR;
          }
     }

}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 != x_0$
x = 22, y = 7, z = 14	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

Solve: $2*y_0 == x_0$
Solution: $x_0 = 2$, $y_0 = 1$

$2*y_0 \mathrel{!=} x_0$

$x = 22$, $y = 7$, $z = 14$ | $x = x_0$, $y = y_0$, $z = 2*y_0$

Concolic Testing Approach

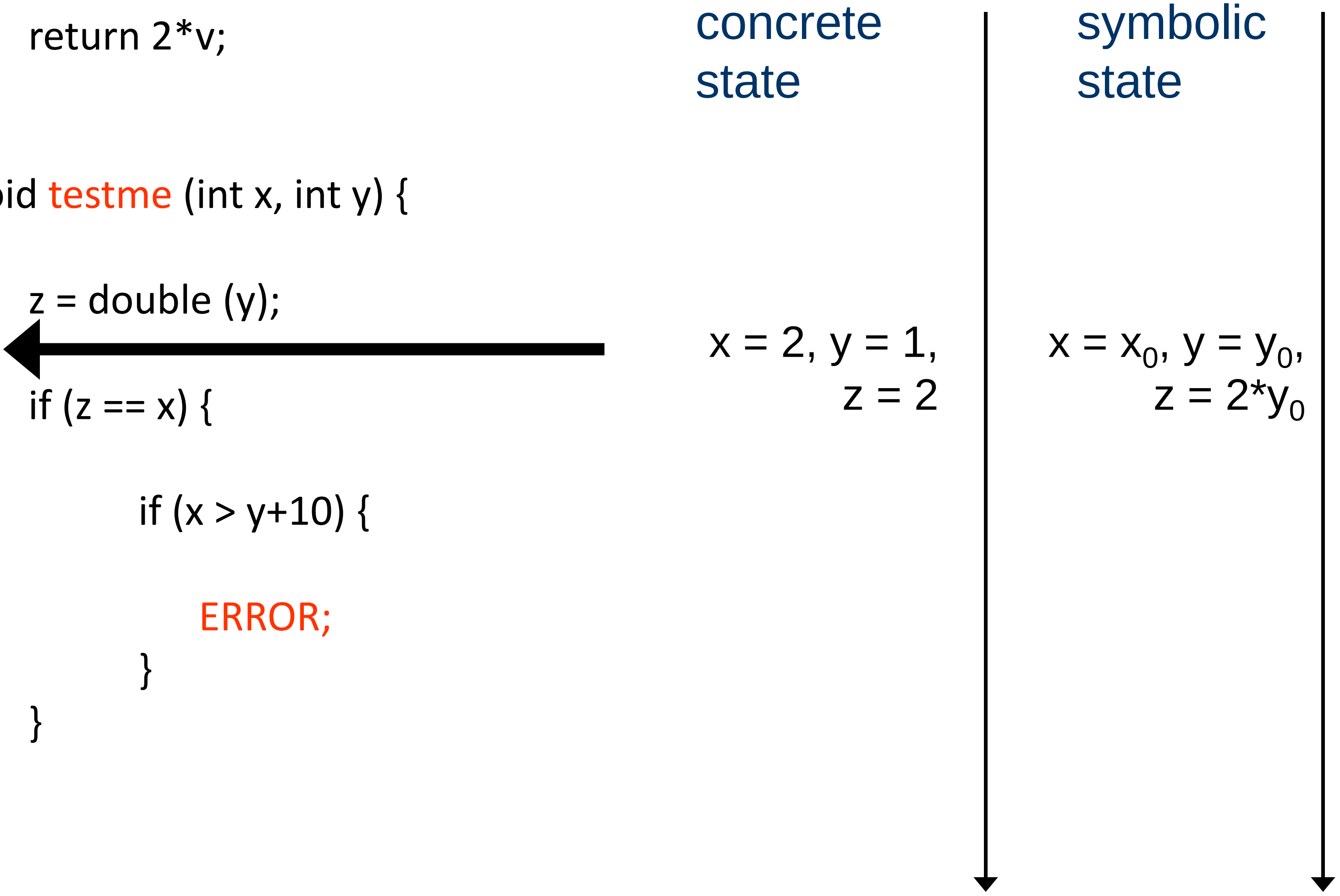
```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	$x = 2, y = 1$		$x = x_0, y = y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	$x = 2, y = 1, z = 2$		$x = x_0, y = y_0, z = 2^*y_0$	



Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 == x_0$
$x = 2, y = 1, z = 2$	$x = x_0, y = y_0, z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 == x_0$
		$x_0 \cdot y_0+10$
x = 2, y = 1, z = 2	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

Solve: $(2*y_0 == x_0) \wedge (x_0 > y_0 + 10)$
Solution: $x_0 = 30$, $y_0 = 15$

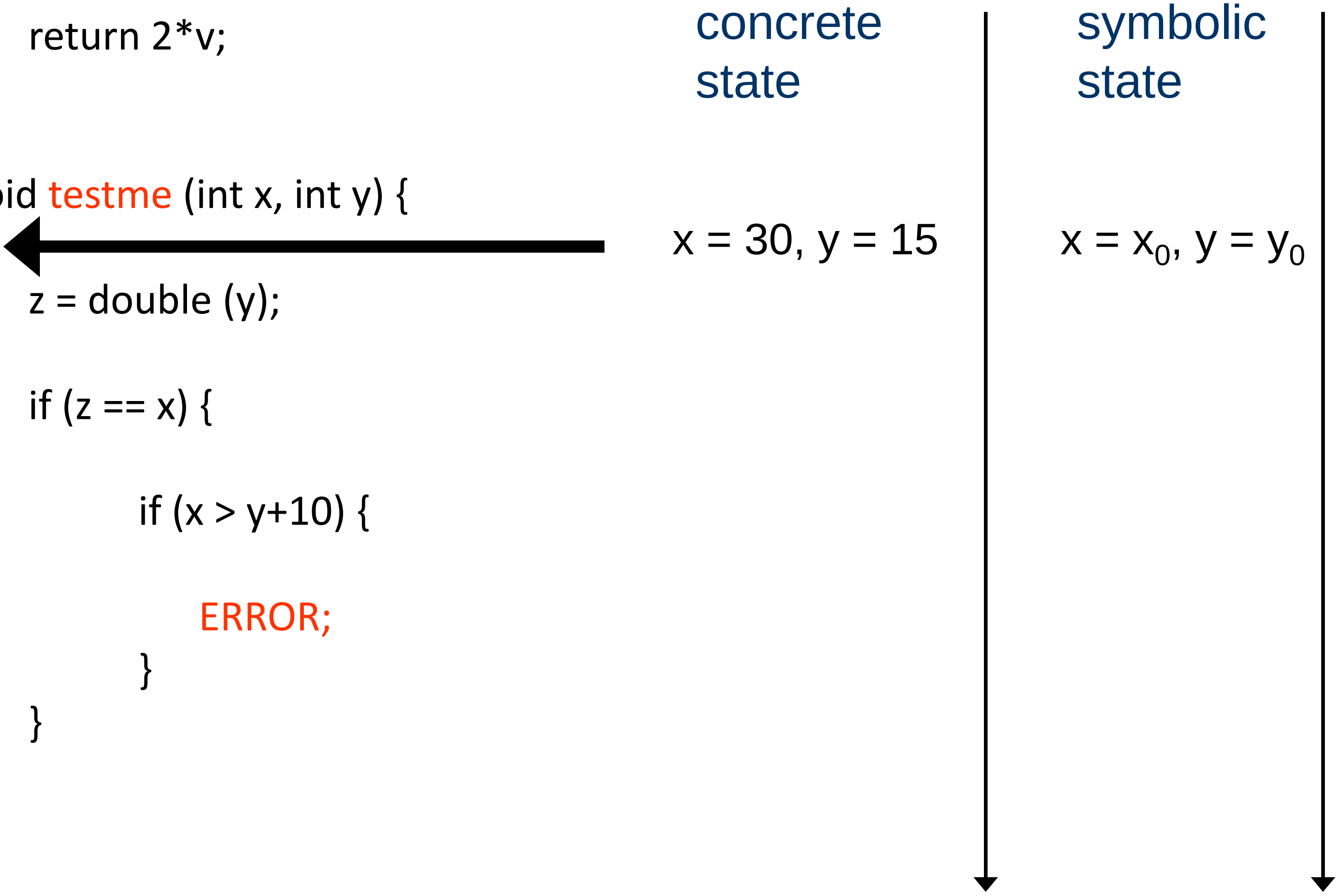
$2*y_0 == x_0$

$x_0 \cdot y_0+10$

$x = 2, y = 1, z = 2$ | $x = x_0, y = y_0, z = 2*y_0$

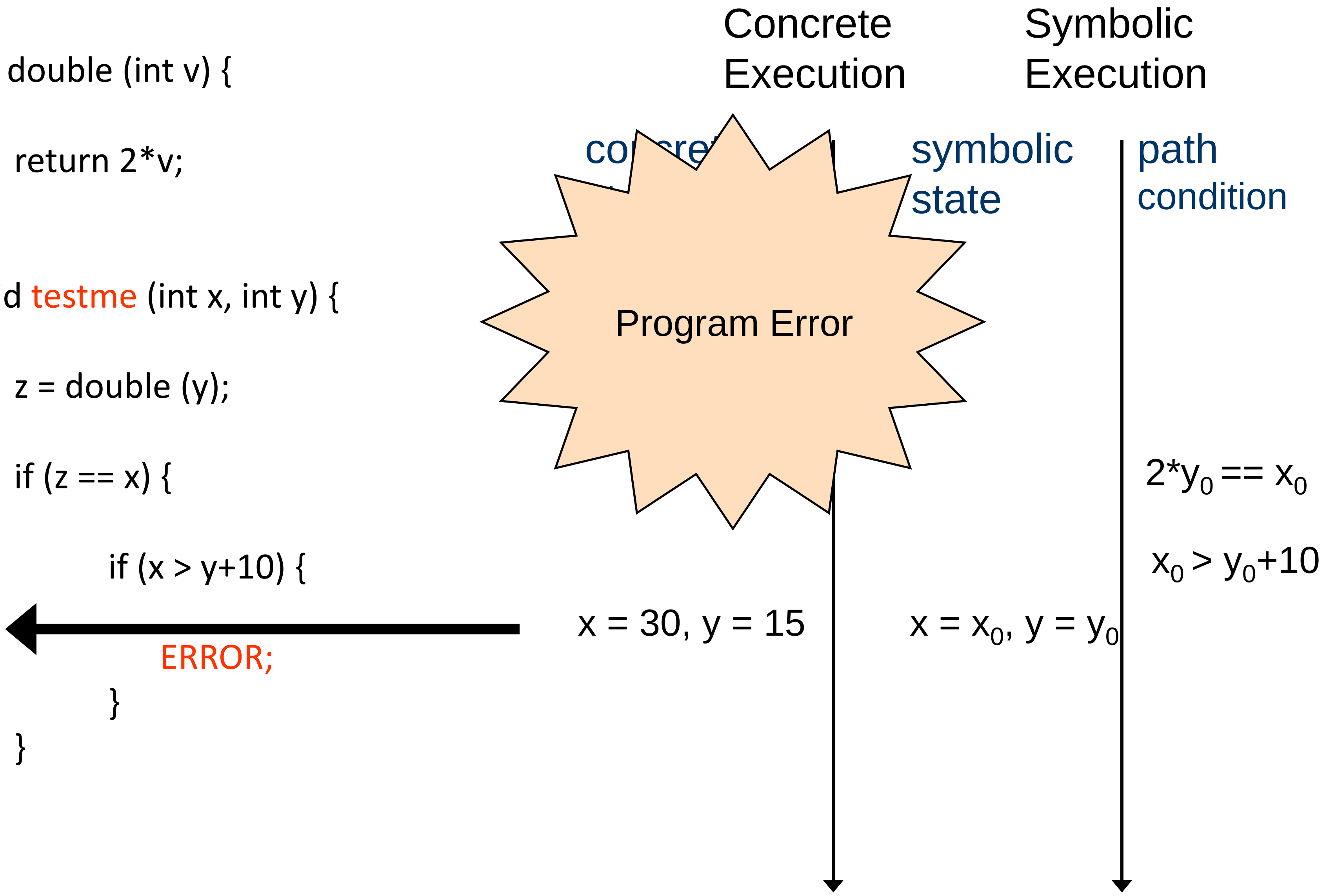
Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

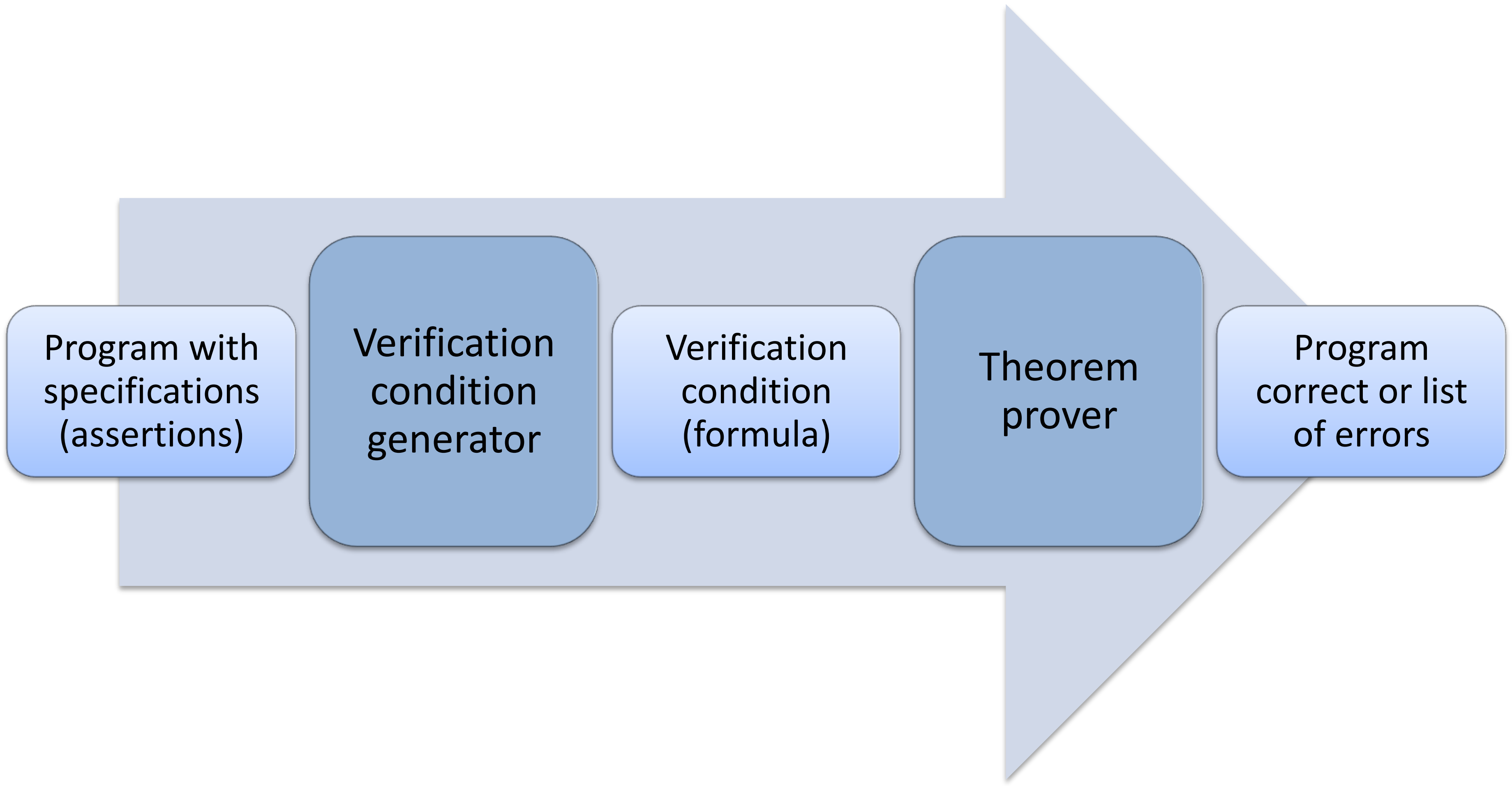


Concolic Testing Approach

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```



Basic Verifier Architecture


Program with specifications (assertions)
Verification condition generator
Verification condition (formula)
Theorem prover
Program correct or list of errors

Verification Condition Generator

- Creates verification conditions (mathematical logic formulas) from program's source code
  - If VC is valid – program is correct
  - If VC is invalid – possible error in program
- Based on the theory of Hoare triples
  - Formalization of software semantics for verification
- Verification conditions computed automatically using *weakest preconditions* (wp)

Simple Command Language

x := E

havoc x

assert P

assume P

S ; T [sequential composition]

S □ T [choice statement]

Program States

- Program state *s*
  - Assignment of values (of proper type) to all program variables
  - Sometimes includes program counter variable *pc*
    - Holds current program location
- Example

  $s : (x \mapsto -1, y \mapsto 1)$

  $s : (pc \mapsto \mathrm{L}, a \mapsto 0, i \mapsto 3)$
- Reachable state is a state that can be reached during some computation

Program States cont.

- A set of program states can be described using a FOL formula
- Example

  Set of states:

  s : { ($x \mapsto 1$), ($x \mapsto 2$), ($x \mapsto 3$) }

  FOL formulas defining s:

  $x = 1$ Ç $x = 2$ Ç $x = 3$

  $0 < x$ Æ $x < 4$ [if x is integer]

Hoare Triple

- Used for reasoning about (program) executions

$$\{ P \}\ S\ \{ Q \}$$

- S is a command
- P is a precondition – formula about program state before S executes
- Q is a postcondition – formula about program state after S executes

Hoare Triple Definition

$$\{ P \}\ S\ \{ Q \}$$

- When a state *s* satisfies precondition P, every terminating execution of command S starting in *s*
  - does not go wrong, and
  - establishes postcondition Q

Hoare Triple Examples

- $\{a = 2\}$ b := a + 3; $\{b > 0\}$
- $\{a = 2\}$ b := a + 3; $\{b = 5\}$
- $\{a > 3\}$ b := a + 3; $\{a > 0\}$
- $\{a = 2\}$ b := a * a; $\{b > 0\}$

Weakest Precondition [Dijkstra]

- The most general (i.e., weakest) P that satisfies

  $$\{ P \}\ S\ \{ Q \}$$

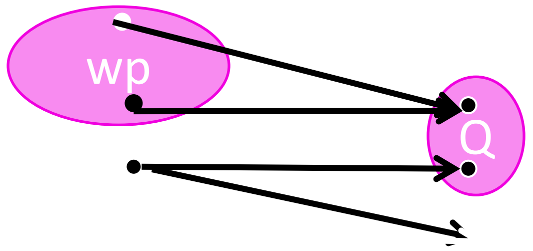
  is called the weakest precondition of S with respect to Q, written:

  $$wp(S, Q)$$

- To check { P } S { Q } prove $P \rightarrow wp(S, Q)$

Weakest Precondition

- wp: Stm $\rightarrow$(Ass$\rightarrow$Ass)
- wp $[\![S]\!]$(Q) – the weakest condition such that every terminating computation of S results in a state satisfying Q
- $\sigma \models \text{wp } [\![S]\!](Q) \leftrightarrow \quad \forall\sigma' : \sigma\ [\![S]\!]\ \sigma' \rightarrow \sigma' \models Q$



Weakest Precondition [Dijkstra]

- The most general (i.e., weakest) P that satisfies

  $$\{ P \}\ S\ \{ Q \}$$

  is called the weakest precondition of S with respect to Q, written:

  $$wp(S, Q)$$

- To check { P } S { Q } prove $P \rightarrow wp(S, Q)$
- Example

  {?P?} b := a + 3; {b > 0}

  {a + 3 > 0} b := a + 3; {b > 0}

  $$wp(b := a + 3, b > 0) = a + 3 > 0$$

Strongest Postcondition

- The strongest Q that satisfies

  $$\{ P \}\ S\ \{ Q \}$$

  is called the strongest postcondition of S with respect to P, written:

  $$sp(S, P)$$

- To check $\{ P \}\ S\ \{ Q \}$ prove $sp(S, P) \rightarrow Q$
- Strongest postcondition is (almost) a dual of weakest precondition

Weakest Preconditions Cookbook

- wp(x := E, Q) = $Q[\, E / x \,]$
- wp(havoc x, Q) = $(\forall x \,.\, Q)$
- wp(assert P, Q) = $P \wedge Q$
- wp(assume P, Q) = $P \rightarrow Q$
- wp(S ; T, Q) = wp(S, wp(T, Q))
- wp(S $\Box$ T, Q) = wp(S, Q) $\wedge$ wp(T, Q)

Checking Correctness with wp

{true}

```
x := 1;

y := x + 2;

assert y = 3;
```

{true}

Checking Correctness with wp cont.

{true}

wp(x := 1, x + 2 = 3) = 1 + 2 = 3 ∧ true

```
x := 1;
```

wp(y := x + 2, y = 3) = x + 2 = 3 ∧ true

```
y := x + 2;
```

wp(assert y = 3, true) = y = 3 ∧ true

```
assert y = 3;
```

{true}

Check: true → 1 + 2 = 3 ∧ true

Example II

{x > 1}

```
y := x + 2;

assert y > 3;
```

{true}

Example II cont.

{x > 1}

wp(y := x + 2, y > 3) = x + 2 > 3

```
y := x + 2;
```

wp(assert y > 3, true) = y > 3 $\wedge$ true = y > 3

```
assert y > 3;
```

{true}

Check: x > 1 $\rightarrow$ (x + 2 > 3)

Example III

{true}

```
assume x > 1;

y := x * 2;

z := x + 2;

assert y > z;
```

{true}

Example III cont.

{true}

wp(assume $x > 1$, $x * 2 > x + 2$) = $x>1 \rightarrow x*2 > x+2$

```
assume x > 1;
```

wp($y := x * 2$, $y > x + 2$) = $x * 2 > x + 2$

```
y := x * 2;
```

wp($z := x + 2$, $y > z$) = $y > x + 2$

```
z := x + 2;
```

wp(assert $y > z$, true) = $y > z \wedge \text{true}$ = $y > z$

```
assert y > z;
```

{true}

Structured if Statement

- Just a “syntactic sugar”:

  if E then S else T

  gets desugared into

  (assume E ; S) □ (assume :E ; T)

Absolute Value Example

```
if (x >= 0) {
   abs_x := x;
} else {
   abs_x := -x;
}
assert abs_x >= 0;
```

While Loop

loop condition

loop body

```
while E
do
   S
end
```

- Loop body S executed as long as loop condition E holds

Desugar While Loop by Unrolling N Times

```
while E do S end =
if E {
   S;
   if E {
      S;
      if E {
         S;
         if E {assume false;} // blocks execution
      }
   }
}
```

Example

```
i := 0;
while i < 2 do i := i + 1 end

i := 0;
if i < 2 {
   i := i + 1;
   if i < 2 {
      i := i + 1;
      if i < 2 {
         i := i + 1;
         if i < 2 {assume false;} // blocks execution
      }
   }
}
```

First Issue with Unrolling

```
i := 0;
while i < 4 do i := i + 1 end

i := 0;
if i < 4 {
   i := i + 1;
   if i < 4 {
      i := i + 1;
      if i < 4 {
         i := i + 1;
         if i < 4 {assume false;} // blocks execution
      }
   }
}
```

Second Issue with Unrolling

```
i := 0;
while i < n do i := i + 1 end

i := 0;
if i < n {
   i := i + 1;
   if i < n {
      i := i + 1;
      if i < n {
         i := i + 1;
         if i < n {assume false;} // blocks execution
      }
   }
}
```

While Loop with Invariant

```
while E
    invariant J
do
    S
end
```

loop condition

loop invariant

loop body

- Loop body S executed as long as loop condition E holds
- Loop invariant J must hold on every iteration
  - J must hold initially and is evaluated before E
  - J must hold even on final iteration when E is false
  - J must be inductive
  - Provided by a user or inferred automatically

Desugaring While Loop Using Invariant

- while E invariant J do S end

check that the loop invariant holds initially

```
assert J;
havoc x; assume J;
(
assume E; S; assert J; assume false
□
assume ¬E
)
```

jump to an arbitrary iteration of the loop

where x denotes the assignment targets of S

check that the loop invariant is maintained by the loop body

exit the loop

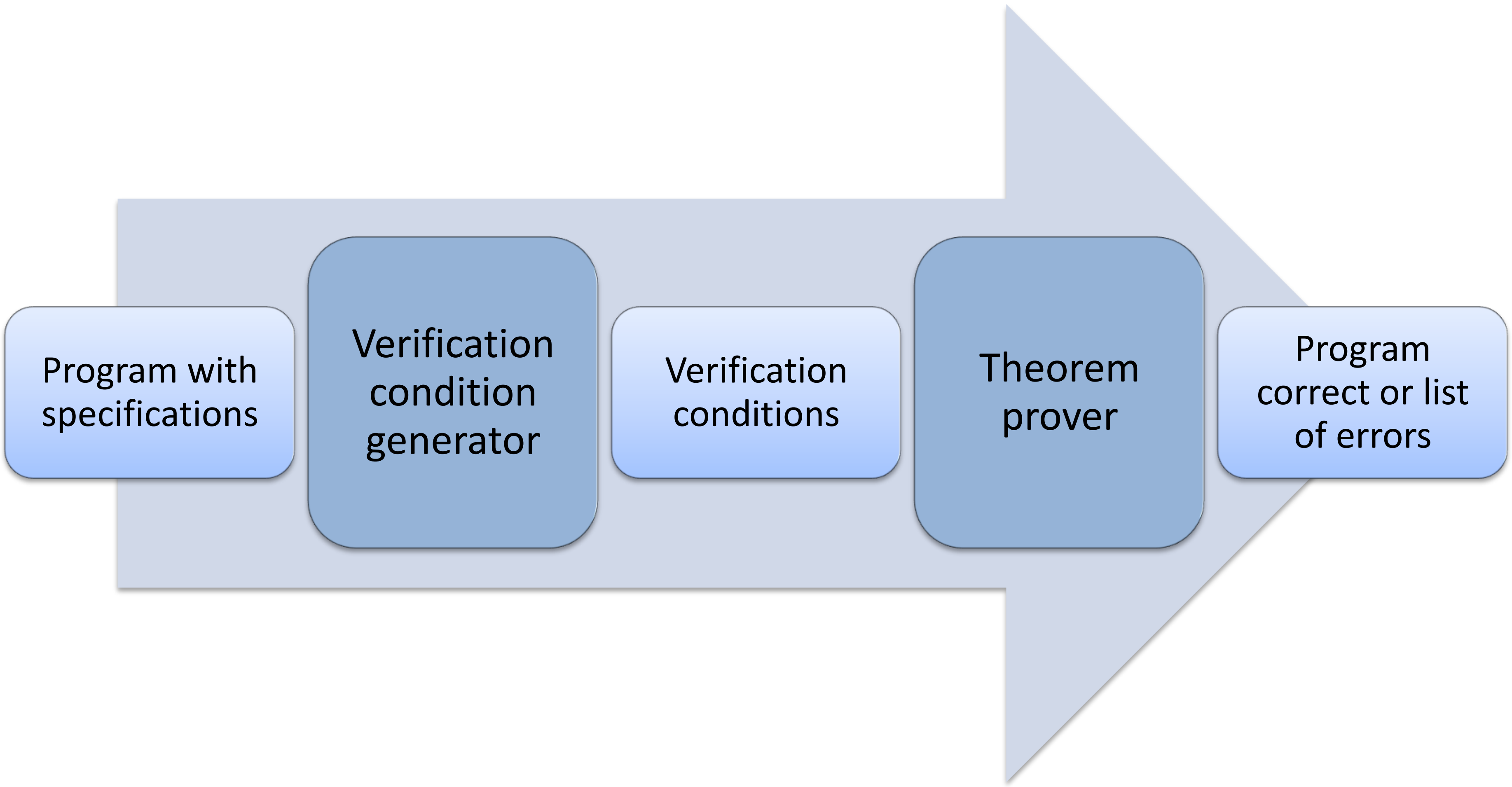
Weakest Precondition of While

- wp(while E invariant J do S end, Q) =

Dafny

- Simple “verifying compiler”
  - Proves procedure contracts statically for all possible inputs
  - Uses theory of weakest preconditions
- Input
  - Annotated program written in simple imperative language
    - Preconditions
    - Postconditions
    - Loop invariants
- Output
  - Correct or list of failed annotations

Dafny Architecture


Program with specifications
Verification condition generator
Verification conditions
Theorem prover
Program correct or list of errors