

# Synthesis of Loop-free Programs

Sumit Gulwani (MSR), **Susmit Jha** (UC Berkeley),  
Ashish Tiwari (SRI) and Ramarathnam  
Venkatesan(MSR)

# From Verification to Synthesis

**Automated synthesis** of systems is the *holy grail* of computer science and engineering.

## Back to the future

“We propose a method of constructing concurrent programs in which the *synchronization skeleton of the program is **automatically synthesized*** from a high-level (branching time) Temporal Logic specification.”

- Edmund M. Clarke, E. Allen Emerson

‘Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic’ Logic of Programs 1981: 52-71.

# From Verification to Synthesis

- Many formal verification techniques exploit the advancements in **constraint solving**: SAT, SMT
- Can we **extend** verification techniques for automated synthesis.
- Synthesis as an **aid to designers and developers**
- Focus on **tedious and non-intuitive** parts of programs which are
  - hard-to-get right by humans and
  - more amenable to automated search based on constraint solvers.

# Motivating Example 1: Floor of two integers' average

$$\text{floor-average}(x, y) = \left\lfloor \frac{x+y}{2} \right\rfloor$$

Challenge is to avoid overflow when  $x$  and  $y$  are large.

From Google Research Blog:

<http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>

# Motivating Example 1: Floor of two integers' average

$$\text{floor-average}(x, y) = \left\lfloor \frac{x+y}{2} \right\rfloor$$

“On computing the semi-sum of two integers” by Salvatore Ruggieri in *Information Processing Letters*, Volume 87 Issue 2, 31 July 2003

*An alternative using bitwise and arithmetic operators from Hacker's Delight book:*

$$\text{floor-average}(x, y) = x|y - ((x \oplus y) \gg 1) - 1$$

( 5 )

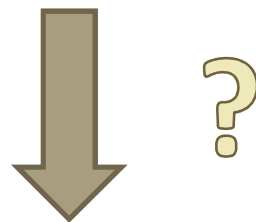
# Motivating Example 1: Floor of two integers' average

$$\text{floor-average}(x, y) = \left\lfloor \frac{x+y}{2} \right\rfloor$$

Logical Specification of floor-average( $x, y$ )

+

*A library of bitwise and arithmetic operators*



$$\text{floor-average}(x, y) = x|y - ((x \oplus y) \gg 1) - 1$$

# Motivating Example 2: Bit twiddling programs

## Turn off rightmost contiguous 1 bits

10**11**0 → 10**00**0

110**1**0 → 110**0**0

Arithmetic: add, subtract, etc

Logical: bitwise-or, bitwise-and,  
bitwise-xor, left-shift, etc.

- Performance critical
- Non-intuitive to write

```
TurnoffRmOnes (x) {  
  i = length(x) - 1;  
  while( x[i] == 0 ){  
    i--;  
    if (i < 0) return x;  
  }  
  x[i] = 0; i--;  
  while( x[i] == 1 ){  
    x[i] = 0; i--;  
    if (i < 0) return x;  
  }  
  return x;  
}
```

# Motivating Example 2: Bit twiddling programs

Turn off rightmost contiguous 1 bits

10**11**0 → 10**00**0  
110**1**0 → 110**0**0

?

```
TurnoffRmOnes (x) {  
  r1 = x - 1;  
  r2 = x || r1 ;  
  r3 = r2 + 1;  
  r4 = r3 && x  
  return r4;  
}
```

```
TurnoffRmOnes (x) {  
  i = length(x) - 1;  
  while( x[i] == 0 ){  
    i--;  
    if (i < 0) return x;  
  }  
  x[i] = 0; i--;  
  while( x[i] == 1 ){  
    x[i] = 0; i--;  
    if (i < 0) return x;  
  }  
  return x;  
}
```

# Problem Definition

## Given:

- Library of components with their functional specification
- Logical Specification of desired behavior
  - Inefficient programs
  - Logical formula over input and output

Obtain: Loop-free Programs using given components with desired behavior.

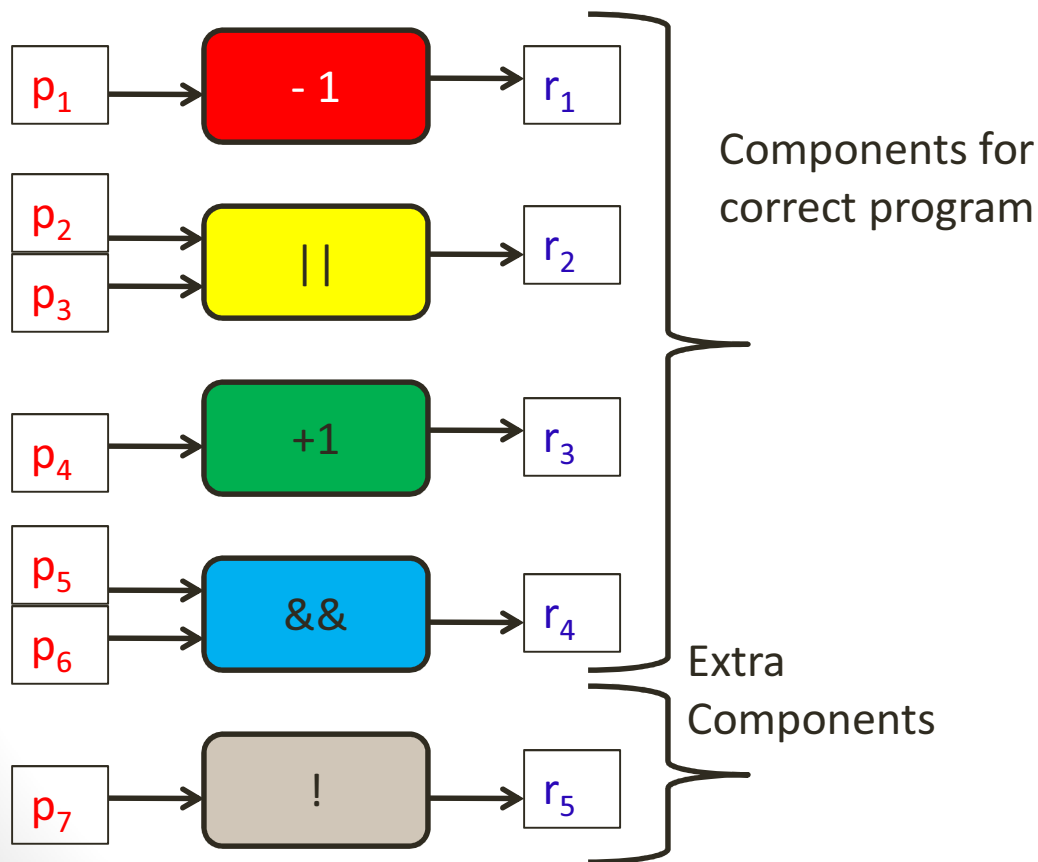
# In rest of the talk

- Encoding Program Space Symbolically
- Counter-example Guided Search for Correct Program
- Correctness Guarantees
- Experimental Results
- Conclusion

# Back to Example

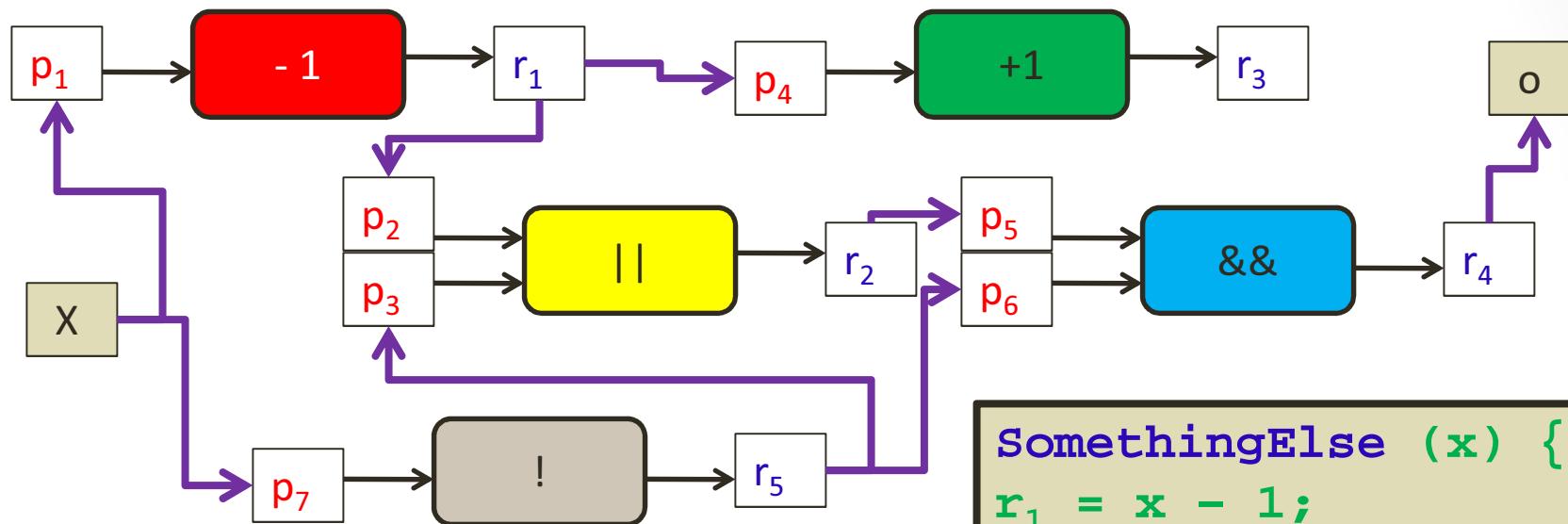
Turn off rightmost contiguous 1 bits

## Component Library



Discover composition of these components that satisfies given specification

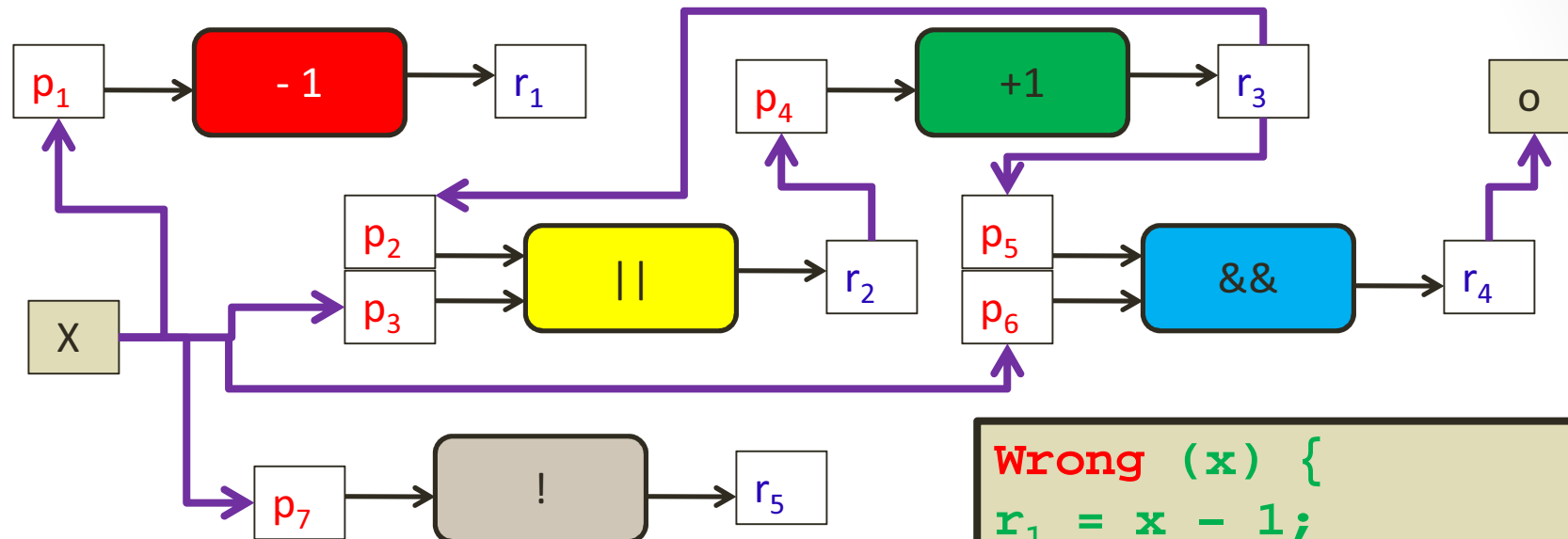
# Component Composition



Each program form corresponds to some composition topology.

```
SomethingElse (x) {  
  r1 = x - 1;  
  r5 = !x  
  r2 = r5 || r1;  
  r4 = r2 && r5;  
  return r4;  
}
```

# Component Composition



Some composition topology do not represent a valid program.

**UNDEFINED VAR ERROR !**

```
Wrong (x) {  
  r1 = x - 1;  
  r2 = x || r3 ;  
  r3 = r2 + 1;  
  r4 = r3 && x  
  return r4;  
}
```

# Component Composition

Program Synthesis Reduces to Searching Over Valid Composition of Library Components

- Encoding Valid Compositions into a logical formula
- Searching over this using satisfiability solving.

# Component Composition

- Represent different compositions of the components as a logical formula parameterized by auxiliary variables  $L$ .

$$\phi_{impl}(I, O, compI, compO, L)$$

- One  $l_x \in L$  variable for each  $x \in I \cup O \cup compI \cup compO$  such that

$$l_x = l_y \text{ iff } x = y$$

These form the interconnection constraints  $\phi_{conn}(I, O, compI, compO, L)$

- Functionality of library components encoded as library constraints  $\phi_{lib}(compI, compO)$ , for example: a bitwise-or component with component inputs  $p_2, p_3$  and output  $r_2$  yields constraint  $r_2 = p_2 || p_3$
- Well-formedness constraints  $\phi_{wff}(L)$  over  $L$ 
  - Variables defined before being used
  - Deterministic Design: Fixing Input  $I$ , fixes all intermediate inputs and outputs as well as output  $O$ .

# Component Composition

- Represent different compositions of the components as a logical formula parameterized by auxiliary variables  $L$ .

$$\phi_{impl}(I, O, compI, compO, L)$$

≡

$$\phi_{wff}(L) \wedge \phi_{lib}(intI, intO) \wedge \phi_{conn}(I, O, intI, intO, L)$$

# Component Composition

After encoding, we require

The correct program produces the same output as the specification

$$\exists L$$
$$\forall I. \exists O, compI, compO$$
$$\phi_{impl}(I, O, compI, compO, L) \wedge \phi_{spec}(I, O)$$

We call this the **synthesis constraint**. with 3 Quantifier Alternations.

# Component Composition

After encoding, we require

The correct program produces the same output as the specification

$$\begin{aligned} & \exists L \\ & \forall I. \exists O, compI, compO \\ & \phi_{impl}(I, O, compI, compO, L) \wedge \phi_{spec}(I, O) \end{aligned}$$

Solve **synthesis constraint** using Induction from example input, outputs similar to **Counter-example Guided Inductive Synthesis** (Sketch, ASPLOS 06)

# Component Composition

How do we get these example?

For any candidate program (L), get an input on which it is incorrect

$$\begin{aligned} & \exists I \\ & \exists O, compI, compO \\ & (\phi_{lib}(intI, intO) \wedge \phi_{conn}(I, O, intI, intO, L) \wedge \sim\phi_{spec}) \end{aligned}$$

We call this the **verification constraint**.

# Component Composition

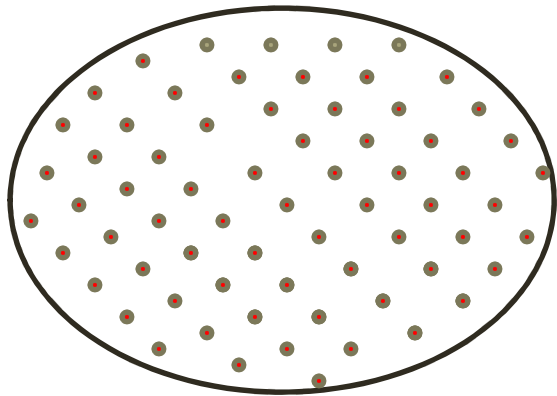
How do we get these example?

For any candidate program (L), get an input on which it is incorrect

$$\exists I$$
$$\exists O, \text{comp}I, \text{comp}O$$
$$(\phi_{lib}(intI, intO) \wedge \phi_{conn}(I, O, intI, intO, L) \wedge \sim\phi_{spec})$$

- *L is always a valid program since synthesis constraints only searches over valid compositions.*
- *Valid compositions are deterministic.*

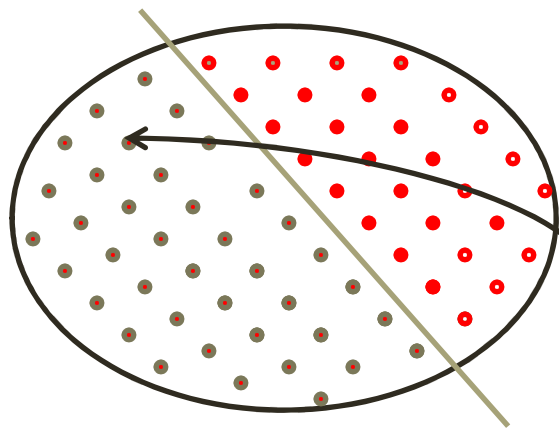
# Approach



Space of all possible programs.  
Each dot represents a program  
corresponding to some value of  $L$

# Approach

Example I/O set  $E := \{(I_1, O_1)\}$  such that  $\phi_{spec}(I_1, O_1)$



Space of all possible programs

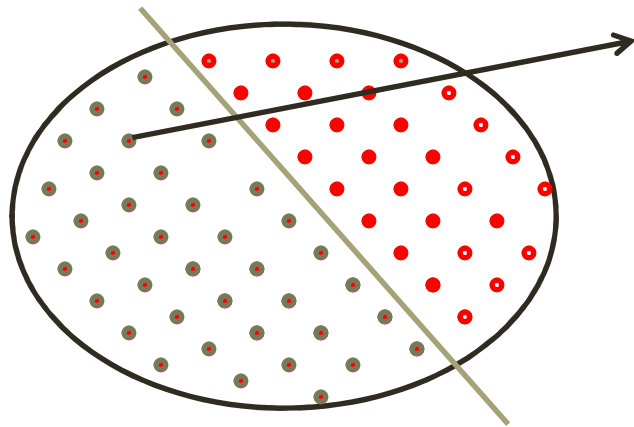
Synthesis Constraint over E



$L_1$

# Approach

Example I/O set  $E := \{(I_1, O_1)\}$



Verification Constraint on  $L_1$



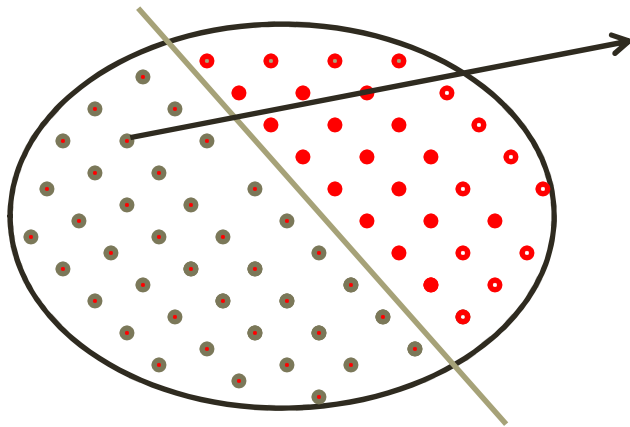
$i_2$

Space of all possible programs

$$\phi_{impl}(I, O, compI, compO, L_1) = \phi_{spec}(I, O) ?$$

# Approach

Example I/O set  $E := \{(I_1, O_1), (I_2, O_2)\}$  such that  $\phi_{spec}(I_2, O_2)$

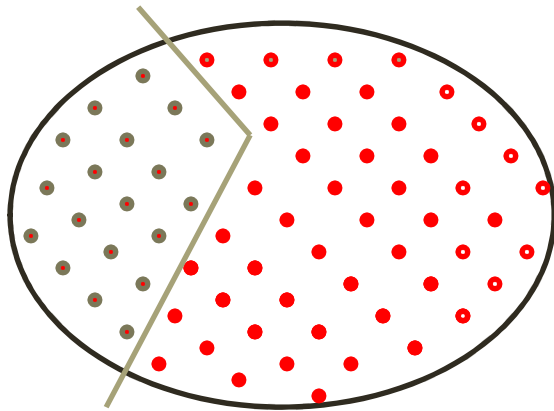


Space of all possible programs

$\phi_{impl}(I, O, compI, compO, L_1) = \phi_{spec}(I, O) ?$   
*No, we get a satisfying model  $I = i_2$*

# Approach

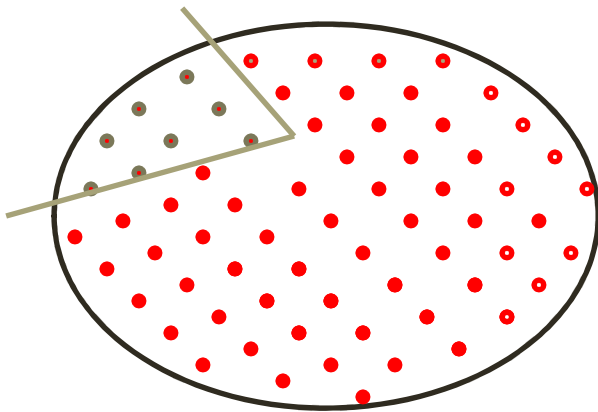
Example I/O set  $E := \{(I_1, O_1), (I_2, O_2)\}$



Space of all possible programs

# Approach

Example I/O set  $E := \{(I_1, O_1), (I_2, O_2), \dots\}$



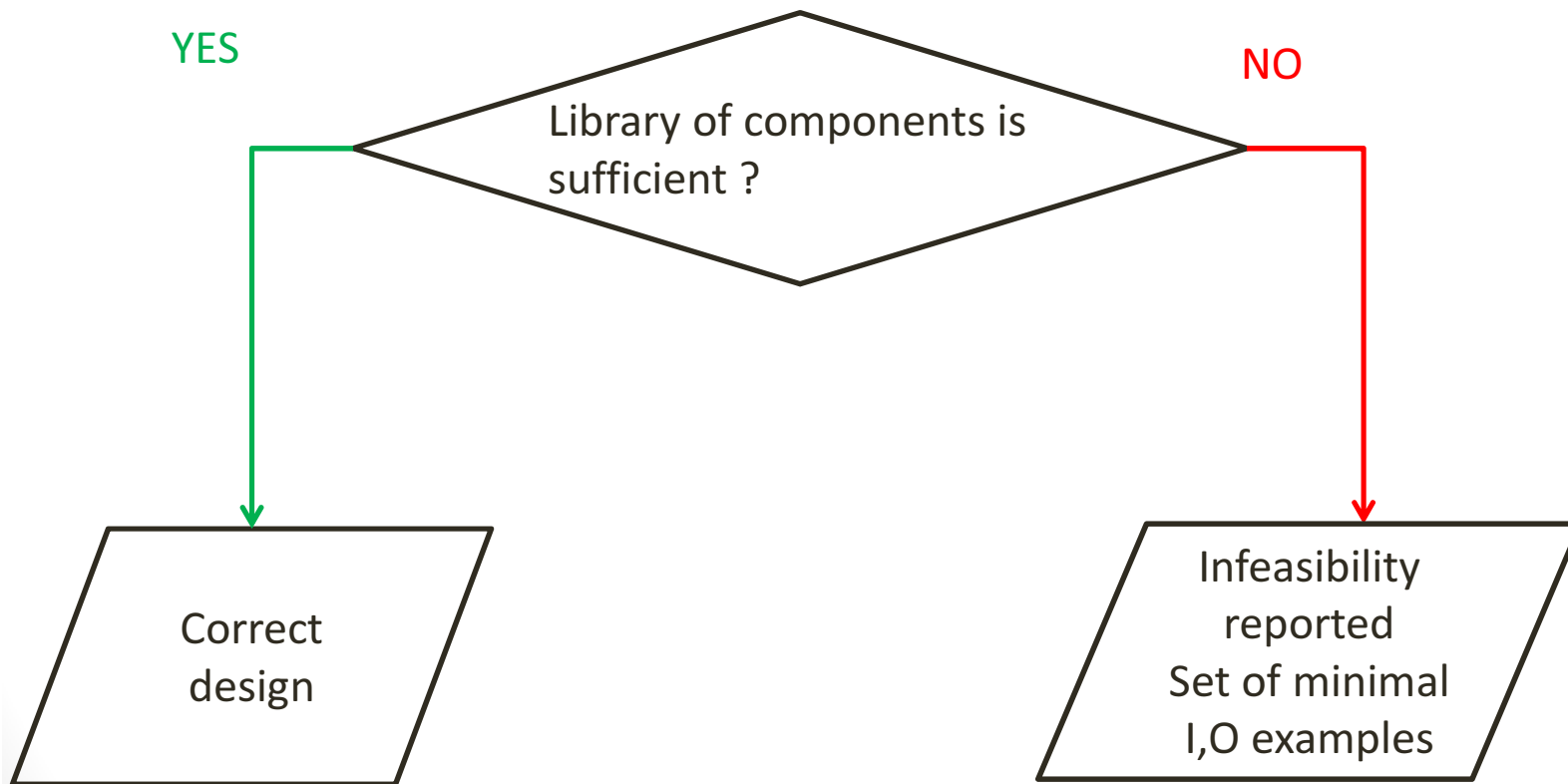
Space of all possible programs

Every verification call

either finds one example  
which eliminates at least one  
wrong program

or reports that no such  
example exists in which case  
we report it as correct  
program.

# Correctness



# Examples of Bitvector Algorithms

**P24: Round up to next highest power of 2**

```
o1 := sub(x,1);
o2 := shr(o1,1);
o3 := or(o1,o2);
o4 := shr(o3,2);
o5 := or(o3,o4);
o6 := shr(o5,4);
o7 := or(o5,o6);
o8 := shr(o7,8);
o9 := or(o7,o8);
o10 := shr(o9,16);
o11 := or(o9,o10);
res := add(o10,1);
```

**P25: Higher order half of product of x and y**

```
o1 := and(x,0xFFFF);
o2 := shr(x,16);
o3 := and(y,0xFFFF);
o4 := shr(y,16);
o5 := mul(o1,o3);
o6 := mul(o2,o3);
o7 := mul(o1,o4);
o8 := mul(o2,o4);
o9 := shr(o5,16);
o10 := add(o6,o9);
o11 := and(o10,0xFFFF);
o12 := shr(o10,16);
o13 := add(o7,o11);
o14 := shr(o13,16);
o15 := add(o14,o12);
res := add(o15,o8);
```

# Runtime and Iterations:

Program		Brahma	
Name	lines	iters	time
P1	2	2	3
P2	2	3	3
P3	2	3	1
P4	2	2	3
P5	2	3	2
P6	2	2	2
P7	3	2	1
P8	3	2	1
P9	3	2	6
P10	3	14	76
P11	3	7	57
P12	3	9	67

Program		Brahma	
Name	lines	iters	time
P13	4	4	6
P14	4	4	60
P15	4	8	119
P16	4	5	62
P17	4	6	78
P18	6	5	46
P19	6	5	35
P20	7	6	108
P21	8	5	28
P22	8	8	279
P23	10	8	1668
P24	12	9	224
P25	16	11	2779

# Result Highlights

- Synthesized over 35 bit-manipulation programs from *Hacker's delight* – Bible of bit-manipulation.
- Efficient Polynomial Evaluation
- Computing powers of a number efficiently.
- Program length: 2-16
- Number of input/output examples: 2 to 15.
- Total runtime: < 1 second to 50 minutes.

# Some Related Work

- Bansal et al. **Automatic Generation of Peephole Superoptimizers** ASPLOS 06
  - *Enumerates short sequences of instructions followed by fingerprint based testing and SAT based equivalence checking*
- Solar-Lezama et al. **Combinatorial sketching for finite programs.** ASPLOS 06
  - *2QBF Boolean satisfiability problem solved using counter-examples generated by equivalence checking*
- Jha et al. **Oracle-guided component-based program synthesis.** ICSE 10
  - *Specification is an input/output blackbox*

# Limitations

- Library Size ?
- What to put in the library ?
- Runtime
  - Number of Components
  - Type of components: ITE, Multiplication are `hard` .

Thanks !

Comments and Questions ?

## Synthesis of Loop-free Programs

Sumit Gulwani (MSR), **Susmit Jha** (UC Berkeley),  
Ashish Tiwari (SRI) and Ramarathnam  
Venkatesan(MSR)