Synthesis of Loop-free Programs

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

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.

Motivating Example 1: Floor of two integers’ average

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

“Oh 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:

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

+ 

\textbf{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**

\[10110 \rightarrow 10000\]
\[11010 \rightarrow 11000\]

- **Arithmetic:** add, subtract, etc
- **Logical:** bitwise-or, bitwise-and, bitwise-xor, left-shift, etc.

• Performance critical
• Non-intuitive to write

```c
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

10110 → 10000
11010 → 11000

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

\[ p_1 \rightarrow -1 \rightarrow r_1 \]
\[ p_2 \rightarrow || \rightarrow r_2 \]
\[ p_3 \rightarrow +1 \rightarrow r_3 \]
\[ p_4 \rightarrow \&\& \rightarrow r_4 \]
\[ p_5 \rightarrow ! \rightarrow r_5 \]

Components for correct program

Discover composition of these components that satisfies given specification

Extra Components
Component Composition

Each program form corresponds to some composition topology.

```python
SomethingElse (x) {
    r1 = x - 1;
    r5 = !x
    r2 = r5 || r1;
    r4 = r2 && r5;
    return r4;
}
```
Some composition topology do not represent a valid program.

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 $L$.

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

• One $l_x \in L$ variable for each $x \in I \cup O \cup \text{compI} \cup \text{compO}$ such that

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

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

• 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.

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

\[ \equiv \]

\[ \phi_{wff}(L) \land \phi_{lib}(intI, intO) \land \phi_{conn}(I, O, intI, intO, L) \]
Component Composition

After encoding, we require

The correct program produces the same output as the specification

\[ \exists L \\
\forall I. \exists O, \text{compI}, \text{compO} \\
\phi_{impl}(I, O, \text{compI}, \text{compO}, L) \land \phi_{spec}(I, O) \]

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

\[ \exists L \]
\[ \forall I. \exists O, \text{compI}, \text{compO} \]
\[ \phi_{\text{impl}}(I, O, \text{compI}, \text{compO}, L) \land \phi_{\text{spec}}(I, O) \]

Solve \textit{synthesis constraint} using Induction from example input, outputs similar to \textit{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

\[ \exists I \exists O, \text{compI, compO} (\phi_{lib}(\text{intI, intO}) \land \phi_{conn}(I, O, \text{intI, intO, L}) \land \sim \phi_{spec}) \]

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

\[
\exists I \\
\exists O, \text{compI}, \text{compO} \\
(\phi_{\text{lib}}(\text{intI}, \text{intO}) \land \phi_{\text{conn}}(I, O, \text{intI}, \text{intO}, L) \land \sim \phi_{\text{spec}})
\]

- \( L \) 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
Approach

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

Verification Constraint on $L_1$

Space of all possible programs

$\phi_{impl}(I, O, \text{comp}I, \text{comp}O, L_1) = \phi_{spec}(I, O)$ ?
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 $E := \{(I_1, O_1), (I_2, O_2), \ldots\}$

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

Library of components is sufficient?  

YES  
Correct design

NO  
Infeasibility reported  
Set of minimal I,O examples
Examples of Bitvector Algorithms

P24: Round up to next highest power of 2

\[
\begin{align*}
o1 & := \text{sub}(x,1); \\
o2 & := \text{shr}(o1,1); \\
o3 & := \text{or}(o1,o2); \\
o4 & := \text{shr}(o3,2); \\
o5 & := \text{or}(o3,o4); \\
o6 & := \text{shr}(o5,4); \\
o7 & := \text{or}(o5,o6); \\
o8 & := \text{shr}(o7,8); \\
o9 & := \text{or}(o7,o8); \\
o10 & := \text{shr}(o9,16); \\
o11 & := \text{or}(o9,o10); \\
\text{res} & := \text{add}(o10,1);
\end{align*}
\]

P25: Higher order half of product of x and y

\[
\begin{align*}
o1 & := \text{and}(x,0xFFFF); \\
o2 & := \text{shr}(x,16); \\
o3 & := \text{and}(y,0xFFFF); \\
o4 & := \text{shr}(y,16); \\
o5 & := \text{mul}(o1,o3); \\
o6 & := \text{mul}(o2,o3); \\
o7 & := \text{mul}(o1,o4); \\
o8 & := \text{mul}(o2,o4); \\
o9 & := \text{shr}(o5,16); \\
o10 & := \text{add}(o6,o9); \\
o11 & := \text{and}(o10,0xFFFF); \\
o12 & := \text{shr}(o10,16); \\
o13 & := \text{add}(o7,o11); \\
o14 & := \text{shr}(o13,16); \\
o15 & := \text{add}(o14,o12); \\
\text{res} & := \text{add}(o15,o8);
\end{align*}
\]
## Runtime and Iterations:

<table>
<thead>
<tr>
<th>Program</th>
<th>Name</th>
<th>lines</th>
<th>iters</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>P10</td>
<td>3</td>
<td>14</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>P11</td>
<td>3</td>
<td>7</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>P12</td>
<td>3</td>
<td>9</td>
<td>67</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Name</th>
<th>lines</th>
<th>iters</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P13</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P14</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P15</td>
<td>4</td>
<td>8</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>P16</td>
<td>4</td>
<td>5</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>P17</td>
<td>4</td>
<td>6</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>P18</td>
<td>6</td>
<td>5</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>P19</td>
<td>6</td>
<td>5</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>P20</td>
<td>7</td>
<td>6</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>P21</td>
<td>8</td>
<td>5</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>P22</td>
<td>8</td>
<td>8</td>
<td>279</td>
<td></td>
</tr>
<tr>
<td>P23</td>
<td>10</td>
<td>8</td>
<td>1668</td>
<td></td>
</tr>
<tr>
<td>P24</td>
<td>12</td>
<td>9</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>P25</td>
<td>16</td>
<td>11</td>
<td>2779</td>
<td></td>
</tr>
</tbody>
</table>
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)