

Quantitative Relational Synthesis With Semantic Preference Objectives

Presented By Sumit Lahiri¹

¹PRAISE Group, CSE Dept. IIT Kanpur

M.Tech Defense

A property that relates multiple executions of (the same or different) program(s) is referred to as a *relational property*.

A property that relates multiple executions of (the same or different) program(s) is referred to as a *relational property*.

Hoare Triple.

$$\{\text{Pre}(\vec{x})\} \mathcal{P} \{\text{Post}(y)\}$$

A property that relates multiple executions of (the same or different) program(s) is referred to as a *relational property*.

Hoare Triple.

$$\{\text{Pre}(\vec{x})\} \mathcal{P} \{\text{Post}(y)\}$$

Relational Property as a Hoare Triple.

$$\{\text{Pre}(\vec{x}_1, \vec{x}_2)\} \mathcal{P}_1, \mathcal{P}_2 \{\text{Post}(y_1, y_2)\}$$

A property that relates multiple executions of (the same or different) program(s) is referred to as a *relational property*.

Hoare Triple.

$$\{\text{Pre}(\vec{x})\} \mathcal{P} \{\text{Post}(y)\}$$

Relational Property as a Hoare Triple.

$$\{\text{Pre}(\vec{x}_1, \vec{x}_2)\} \mathcal{P}_1, \mathcal{P}_2 \{\text{Post}(y_1, y_2)\}$$

When both programs \mathcal{P}_1 and \mathcal{P}_2 are the same programs, i.e \mathcal{P} , *relational properties* become **hyper-properties**!.

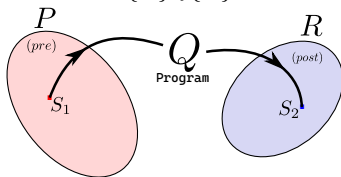
What is program sketching?

A partial program (referred to as a *sketch*), which leaves out certain *holes* for the synthesizer to fill such that the completed program satisfies the required specification.

What is program sketching?

A partial program (referred to as a *sketch*), which leaves out certain *holes* for the synthesizer to fill such that the completed program satisfies the required specification.

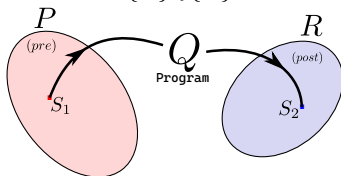
Specification as a **Hoare Triple**,
 $\{P\}Q\{R\}$



What is program sketching?

A partial program (referred to as a *sketch*), which leaves out certain holes for the synthesizer to fill such that the completed program satisfies the required specification.

Specification as a **Hoare Triple**,
 $\{P\}Q\{R\}$



```
1  int P(int n){  
2      // PRE :  assume(n > 1);  
3      int i = 0, x = 0;  
4      while(i < n){  
5          i = i + 1;  
6          x = .  
7      }  
8      // POST :  assert(x > 2 * n);  
9      return x;  
10 }
```


Given partial programs $\mathcal{P}_1^{[\cdot]}$ and $\mathcal{P}_2^{[\cdot]}$, find completion \mathcal{E} , where $\mathcal{E}.H$ and $\mathcal{E}.G$ respective completions of $\mathcal{P}_1^{[\cdot]}$ and $\mathcal{P}_2^{[\cdot]}$ that satisfy a specification.

Given partial programs $\mathcal{P}_1^{[\cdot]}$ and $\mathcal{P}_2^{[\cdot]}$, find completion \mathcal{E} , where $\mathcal{E}.H$ and $\mathcal{E}.G$ respective completions of $\mathcal{P}_1^{[\cdot]}$ and $\mathcal{P}_2^{[\cdot]}$ that satisfy a specification.

Relational Synthesis

$$\exists \mathcal{E}. \{ \text{Pre}(\vec{x}_1, \vec{x}_2) \} \mathcal{P}_1^{[\mathcal{E}.H]}, \mathcal{P}_2^{[\mathcal{E}.G]} \{ \text{Post}(y_1, y_2) \}$$

Program equivalence requires that, any two executions of a pair of programs, \mathcal{P}_1 and \mathcal{P}_2 on same the same input \vec{x} , must yield the same outputs.

Program equivalence requires that, any two executions of a pair of programs, \mathcal{P}_1 and \mathcal{P}_2 on same the same input \vec{x} , must yield the same outputs.

A Verification Problem

$$\forall \vec{x}. \mathcal{P}_1(\vec{x}) = \mathcal{P}_2(\vec{x})$$

Given a reference program \mathcal{P}_1 and a *partial* program \mathcal{P}_2 that has a hole \square , we are interested in *completing* the partial program by synthesizing an expression to fill the hole such that the two programs are rendered semantically equivalent.

Given a reference program \mathcal{P}_1 and a *partial* program \mathcal{P}_2 that has a hole \square , we are interested in *completing* the partial program by synthesizing an expression to fill the hole such that the two programs are rendered semantically equivalent.

Formal Definition

$$\exists \mathcal{E} \in \mathcal{L}(\mathcal{G}). \forall \vec{x}. \mathcal{P}_1(\vec{x}) = \mathcal{P}_2^{[\mathcal{E}]}(\vec{x})$$

Sketching for Program Equivalence

```
1 int  $\mathcal{P}_1$ (int n){
2     assume(n > 1);
3     int i = 0, ans = 0;
4     while(i < (n - 1)){
5         i = i + 1;
6         ans = ans + (5 * i) + 1;
7     }
8     return ans + 1;
9 }
```

(a) Program 1

```
1 int  $\mathcal{P}_2^{[\cdot]}$ (int n){
2     assume(n > 1);
3     int x = 0, y = 0, z = n;
4     while(z  $\neq$  0){
5         z = z - 1;
6         x =  $\boxed{\cdot}$ ;
7         y = y + 1;
8     }
9     return x + y;
10 }
```

(b) Program 2 (with hole $\boxed{\cdot}$)

Sketching for Program Equivalence

```
1 int P1(int n){
2     assume(n > 1);
3     int i = 0, ans = 0;
4     while(i < (n - 1)){
5         i = i + 1;
6         ans = ans + (5 * i) + 1;
7     }
8     return ans + 1;
9 }
```

(a) Program 1

```
1 int P2[·](int n){
2     assume(n > 1);
3     int x = 0, y = 0, z = n;
4     while(z ≠ 0){
5         z = z - 1;
6         x = [·];
7         y = y + 1;
8     }
9     return x + y;
10 }
```

(b) Program 2 (with hole [·])

Post condition for sketching.

$$\exists \mathcal{E} \in \mathcal{L}(\mathcal{G}). \forall \vec{x}. P_1(\vec{x}) = P_2^{[\mathcal{E}]}(\vec{x})$$

Sketching for Program Equivalence

```
1 int P1(int n){  
2     assume(n > 1);  
3     int i = 0, ans = 0;  
4     while(i < (n - 1)){  
5         i = i + 1;  
6         ans = ans + (5 * i) + 1;  
7     }  
8     return ans + 1;  
9 }
```

(a) Program 1

```
1 int P2[·](int n){  
2     assume(n > 1);  
3     int x = 0, y = 0, z = n;  
4     while(z ≠ 0){  
5         z = z - 1;  
6         x = ·;  
7         y = y + 1;  
8     }  
9     return x + y;  
10 }
```

(b) Program 2 (with hole ·)

Post condition for sketching.

$$\exists \mathcal{E} \in \mathcal{L}(\mathcal{G}). \forall \vec{x}. P_1(\vec{x}) = P_2^{[\mathcal{E}]}(\vec{x})$$

assert(ans + 1 = x + y)

```
1 int  $\mathcal{P}_1$ (int n){  
2     assume(n > 1);  
3     int i = 0, ans = 0;  
4     while(i < (n - 1)){  
5         i = i + 1;  
6         ans = ans + (5 * i) + 1;  
7     }  
8     return ans + 1;  
9 }
```

(a) Program 1

```
1 int  $\mathcal{P}_2^{[,]}$ (int n){  
2     assume(n > 1);  
3     int x = 0, y = 0, z = n;  
4     while(z  $\neq$  0){  
5         z = z - 1;  
6         x =  $x + 6 \cdot y - n$ ;  
7         y = y + 1;  
8     }  
9     return x + y;  
10 }
```

(b) Program 2 (with completion)

Program equivalence posed as a **relational** property.

Program equivalence posed as a **relational** property.

Relational Synthesis for Program Equivalence

$$\exists \mathcal{E}. \{x_1 = x_2\} \quad \mathcal{P}_1^{[\mathcal{E}.H]}, \mathcal{P}_2^{[\mathcal{E}.G]} \quad \{\mathcal{P}_1^{[\mathcal{E}.H]}(x_1) = \mathcal{P}_2^{[\mathcal{E}.G]}(x_2)\}$$

Given public inputs, (\vec{x}_1, \vec{x}_2) and secret inputs (s_1, s_2) , program (\mathcal{P}) does not reveal any information about the secret inputs.

$$\exists \mathcal{E}. \{ \vec{x}_1 = \vec{x}_2 \} \mathcal{P}^{[\mathcal{E}]} \{ \mathcal{P}^{[\mathcal{E}]}(s_1, \vec{x}_1) = \mathcal{P}^{[\mathcal{E}]}(s_2, \vec{x}_2) \}$$

Weak Equivalence

$$\exists \mathcal{E}. \{ \vec{x}_1 = \vec{x}_2 \} \mathcal{P}_1^{[\mathcal{E}.H]}, \mathcal{P}_2^{[\mathcal{E}.G]} \{ \| \mathcal{P}_1^{[\mathcal{E}.H]}(\vec{x}_1) - \mathcal{P}_2^{[\mathcal{E}.G]}(\vec{x}_2) \| \leq c \}$$

Weak Non-Interference

$$\exists \mathcal{E}. \{ \vec{x}_1 = \vec{x}_2 \} \mathcal{P}^{[\mathcal{E}]} \{ \| \mathcal{P}^{[\mathcal{E}]}(s_1, \vec{x}_1) - \mathcal{P}^{[\mathcal{E}]}(s_2, \vec{x}_2) \| \leq c \}$$

Robustness requires that small changes in the inputs must not lead to large difference in the responses of the program.

$$\exists \mathcal{E}. \{ \|\vec{x}_1 - \vec{x}_2\| \leq d_1 \} \mathcal{P}^{[\mathcal{E}]} \{ \|\mathcal{P}^{[\mathcal{E}]}(\vec{x}_1) - \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2)\| \leq f(\vec{x}_1, \vec{x}_2) \}$$

Program must not use the *sensitive attribute* (s) to be unfair to the minority population.

$$\exists \mathcal{E}. \{s_1 \leq s_2 \wedge \vec{x}_2 \sqsubseteq \vec{x}_1\} \mathcal{P}^{[\mathcal{E}]} \{ \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2, s_2) \leq \mathcal{P}^{[\mathcal{E}]}(\vec{x}_1, s_1) \}$$

Monotonicity is a hyper-property that requires that for any two executions of the program, if the inputs are ordered, so must be the outputs.

$$\exists \mathcal{E}. \{ \vec{x}_1 \sqsubseteq \vec{x}_2 \} \mathcal{P}^{[\mathcal{E}]} \{ \mathcal{P}^{[\mathcal{E}]}(\vec{x}_1) \sqsubseteq \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2) \}$$

Relational property with (semantic) quantitative objectives.

Monotonicity, Robustness: A preference on a completion could be the one that minimizes the distance between any two responses of the program.

$$\Gamma(\mathcal{P}^{[\mathcal{E}]}(\vec{x}_1), \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2)) \triangleq \|\mathcal{P}^{[\mathcal{E}]}(\vec{x}_1) - \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2)\|$$

Weak Non-Interference, Weak Equivalence: A preference on a completion could be the one that minimizes the distance between any two responses of the program.

$$\Gamma(\mathcal{P}^{[\mathcal{E}]}(\vec{x}_1), \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2)) \triangleq \|\mathcal{P}^{[\mathcal{E}]}(\vec{x}_1) - \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2)\|$$

Group Fairness: One may design many preference metrics over completions. One metric could be to prefer completions where the deviation in responses between candidates of two populations is small for similar candidates.

$$\Gamma(\mathcal{P}^{[\mathcal{E}]}(\vec{x}_1, s_1), \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2, s_2)) \triangleq \begin{cases} \text{if } (s_1 < s_2 \wedge \vec{x}_1 \sim \vec{x}_2) \\ \text{then } \|\mathcal{P}^{[\mathcal{E}]}(\vec{x}_1, y_1) - \mathcal{P}^{[\mathcal{E}]}(\vec{x}_2, y_2)\| \text{ else } 0 \end{cases}$$

Monotonicity Example

```
1  int  $\mathcal{P}^{[\cdot]}$  (int a, int b){  
2      assume((0 < a) && (a < b));  
3      while (a < b) {  
4          c = c +  $\square$ ;  
5          a = a + 1;  
6      }  
7      return c;  
8  }
```

Figure: Program Sketch for Monotonicity

Monotonicity Example

```
1 int  $\widehat{\mathcal{P}}[\cdot]$ (int  $a_1$ , int  $b_1$ , int  $a_2$ , int  $b_2$ ){
2   assume((0 <  $a_1$ ) && ( $a_1$  <  $b_1$ ));
3   assume((0 <  $a_2$ ) && ( $a_2$  <  $b_2$ ));
4   int  $c_1$  = 0,  $c_2$  = 0;
5   while ( ( $a_1$  <  $b_1$ ) || ( $a_2$  <  $b_2$ ) ) {
6     if ( ( $a_1$  <  $b_1$ ) ) {
7        $c_1$  =  $c_1$  +  $\square$ ;
8        $a_1$  =  $a_1$  + 1;
9     }
10    if ( ( $a_2$  <  $b_2$ ) ) {
11       $c_2$  =  $c_2$  +  $\square$ ;
12       $a_1$  =  $a_1$  + 1;
13    }
14  }
15  return  $c_1$ ,  $c_2$ ;
16 }
```

Figure: Product program for Monotonicity from Fig. 3.

Machine configuration, Benchmark Sources, Domain-specific Language.

Instances without quantitative objectives.

Bench	Property	Time(s)
b26	Strict Equivalence	4
b10	Strict Equivalence	3
b18	Strict Equivalence	2
b16	Strict Equivalence	1
b21	Strict Equivalence	3
b27	Strict Equivalence	4
b04	Strict Equivalence	3
b34	Strict Equivalence	10
b05	Strict Equivalence	7
nonintf01	Strict Non-Interference	7
nonintf02	Strict Non-Interference	8
nonintf05	Strict Non-Interference	6

Bench	Property	Time(s)	Best?
mono01	Monotonicity	329	✓
mono02	Monotonicity	311	✓
mono02	Monotonicity	310	✓
weak01	Weak Equivalence	210	✓
weak02	Weak Equivalence	198	✓
weak03	Weak Equivalence	128	✓
weak04	Weak Equivalence	168	✓
robust01	Robustness	95	✓
robust02	Robustness	102	✓
fair01	Group Fairness	82	✓
nonintf03	Weak Non-Interference	70	✓
nonintf04	Weak Non-Interference	75	✓

Thank You!