Lecture 11 – Partial Programs, Program Repair, and Sketching

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav
Previously

- Synthesis from examples
- SMARTEdit
- String processing in spreadsheet from examples (a little bit)
Today

- Program Repair as a Game
  - angelic non-determinism in the program
- Sketching
  - completion of partial programs using a backing SAT solver

- Acks
  - Program Repair slides from Barbara Jobstmann
  - Sketching slides cannibalized from Armando Solar-Lezama
Reactive Systems

Systems (e.g., servers)
• Interact with environment
• Infinite duration (non-terminating)
• Finite data (or data abstractions)
• Control-oriented

Specifications
• Set of good behaviors (a language)
• Temporal logic (including safety and liveness)

Analysis

Synthesis

Verification

How to verify a Reactive System?

Core algorithm for linear-time temporal logic:

1. Source code $\rightarrow_{\text{auto/manual}}$ transition system (FSM)
2. Specification $\rightarrow_{\text{auto/manual}}$ monitor violations
3. Check if model has a violating trace
   - product of trans. system and monitor
   - check for exists of a trace in product (emptiness)

$$L(\text{Program}) \subseteq L(\text{Specification})$$

$$L(\text{Program}) \cap L(\neg \text{Specification}) = \emptyset$$

$$L(\text{Program} \times \neg \text{Specification}) = \emptyset$$
Source Code

unsigned int got_lock = 0;

...  
1: while(*) {
   ...
2:   if (*) {
3:      lock();
4:      got_lock++;
   }
   ...
5:   if (got_lock != 0) {
6:      unlock();
   }
7:   got_lock--;
   ...
}
Step 1: Transition System

```c
int[0,1,2] got_lock = 0;
...
1: while(*) {
   ...
2:   if (*) {
3:      lock();
   lock:  {LOCK:=1;}
4:      got_lock++;
   }
   ...
5:   if (got_lock != 0) {
6:      unlock();
   unlock: {LOCK:=0;}
   }
7:   got_lock--; 
   ...
}
8: 
```

Trans. system variables: line (l), got_lock (gl)
Specification

P1: do not acquire a lock twice
P2: do not call unlock without holding the lock

**P1:** always( line=lock implies next( line!=lock w-Until line=unlock ) )

**P2:** ( line!=unlock w-until line=lock ) and
        always( line=unlock implies
                next( line!=unlock w-until line=lock ) )
Linear-Time Temporal Logic \[\text{[Pnueli77]}\]

- **Syntax:**
  - Atomic propositions, e.g., line=1, line!=1, got_lock=0
  - Boolean operators: **not, and, or, implies,** ...
  - Temporal operators:
    - **next (ϕ)** \(\ldots ϕ \) holds in the next step
    - **ϕ₁ until ϕ₂** \(\ldots ϕ₁ \) holds until at some point \(ϕ₂ \) holds

- Used in industrial spec languages PSL/SVA
- Can express many interesting properties, e.g., mutual exclusion, deadlock freedom, termination
Linear-Time Temporal Logic

- **Semantics**
  - defined with respect to infinite traces
  - in each step atomic propositions holds or not
  - E.g., line=1, got_lock≤1

Given a finite set of atomic proposition AP, a trace (or word) $w$ over AP is an infinite sequence of truth assignments to AP, i.e., $w \in (2^AP)^\omega$. 
Linear-Time Temporal Logic

- Semantics
  - \textbf{next} ($\phi$)... $\phi$ holds in the next step
    \[ \phi \xrightarrow{} \quad \xrightarrow{} \quad \xrightarrow{} \quad \xrightarrow{} \quad \xrightarrow{} \quad \text{.....} \]
  - $\phi_1 \textbf{ until } \phi_2$... $\phi_1$ holds until at some point $\phi_2$ holds
    \[ \phi_1 \xrightarrow{} \phi_1 \xrightarrow{} \phi_1 \xrightarrow{} \text{.....} \quad \phi_1 \xrightarrow{} \phi_2 \xrightarrow{} \text{.....} \]
- System S satisfies/models $\phi$, if all its behaviors satisfy $\phi$
How to verify a Reactive System?

Core algorithm for linear-time temporal logic:

1. Source code $\rightarrow_{\text{auto/manual}}$ transition system (FSM)
2. Specification $\rightarrow_{\text{auto/manual}}$ monitor violations
3. Check if model has a violating trace
   - product of trans. system and monitor
   - check for exists of a trace in product (emptiness)
Step 2: Monitor for Violations

P₁: always( line=lock implies
  next( line!=lock w-until line=unlock ))
= not eventually( line=lock and
  next( line!=unlock until line=lock ))

Why do we track bad and not good behaviors?

Automaton accepts trace/behavior if a green state is visited infinitely often (Büchi)

Why do we track bad and not good behaviors?

L(S) ⊆ L(φ): forall w: w ∈ L(S) → w ∈ L(φ)
  ¬ exists w: w ∈ L(S) ∧ w ∈ L(¬φ)
Step 3: Product

\[
\begin{align*}
&l=1, gl=0 \\
&l=2, gl=0 \\
&l=3, gl=0 \\
&l=lock, gl=0 \\
&l=4, gl=0 \\
&l=5, gl=0 \\
&l=unlock, gl=0 \\
&l=7, gl=0 \\
&l=8, gl=0 \\
&l=1, gl=1 \\
&l=2, gl=1 \\
&l=3, gl=1 \\
&l=lock, gl=1 \\
&l=4, gl=1 \\
&l=5, gl=1 \\
&l=unlock, gl=1 \\
&l=7, gl=1 \\
&l=8, gl=1 \\
&l=1, gl=2 \\
&l=2, gl=2 \\
&l=3, gl=2 \\
&l=lock, gl=2 \\
&l=4, gl=2 \\
&l=5, gl=2 \\
&l=unlock, gl=2 \\
&l=7, gl=2 \\
&l=8, gl=2 \\
\end{align*}
\]
Step 3: Product

\[
\begin{align*}
\text{l} &= 1, \quad \text{gl} = 0 \\
\text{l} &= 2, \quad \text{gl} = 0 \\
\text{l} &= 3, \quad \text{gl} = 0 \\
\text{l} &= \text{lock}, \quad \text{gl} = 0 \\
\text{l} &= 4, \quad \text{gl} = 0, \quad \text{gl} = 2 \\
\text{l} &= 5, \quad \text{gl} = 0 \\
\text{l} &= 6, \quad \text{gl} = 1 \\
\text{l} &= \text{unlock}, \quad \text{gl} = 2 \\
\text{l} &= 7, \quad \text{gl} = 0 \\
\text{l} &= 8, \quad \text{gl} = 0 \\
\end{align*}
\]
Step 3: Product

Recall, we want to show a violation:
Step 3: Product

Recall, we want to show a violation:
non-determinism in transition system and in monitor pull in the same direction
(both can be used to violate property)
Source Code

```c
int[0,1,2] got_lock = 0;
...
1: while(*) {
   ...
2:  if (*) {
3:    lock();
   lock:   {LOCK:=1;}
4:    got_lock++;
   }
   ...
5:  if (got_lock != 0) {
6:    unlock();
   unlock: {LOCK:=0;}
   }
7:  got_lock--;
   ...
}
8: 
```
How to verify a Reactive System?

Core algorithm for linear-time temporal logic:
1. Source code $\rightarrow_{\text{auto/manual}}$ transition system (FSM)
2. Specification $\rightarrow_{\text{auto/manual}}$ monitor violations
3. Check if model has a violating trace
   - product of trans. system and monitor
   - check for exists of a trace in product (emptiness)

But how to repair it?
How to repair a Reactive System?

1. Add freedom (choice for the system, allowed ways to modify system)
2. Source code $\rightarrow_{a/m}$ transition system (game)
3. Specification $\rightarrow_{a/m}$ monitor acceptance
4. Check if we can find system choices s.t. model is accepted by monitor
   - product of trans. system and monitor
   - search for winning strategy in game
Step 1: Freedom

```c
int[0,1,2] got_lock = 0;
int[0,1,2] freedom;
...
1: while(*) {
   ...
2:   if (*) {
3:      lock();
lock:   {LOCK:=1;}
4:      got_lock:=freedom;
   }
   ...
5:   if (got_lock != 0) {
6:      unlock();
unlock: {LOCK:=0;}
   }
7:   got_lock:=freedom;
   ...
}
(We can also extend to fault localization)
```
Step 2: Game

\[
\begin{align*}
\text{int}[0,1,2] \ got\_lock &= 0; \\
\text{int}[0,1,2] \ freedom;
\end{align*}
\]

\[
\begin{align*}
1: & \quad \text{while}(*) \{ \\
& \quad \quad \ldots \\
2: & \quad \text{if} (*) \{ \\
3: & \quad \quad \text{lock}(); \\
& \quad \quad \text{lock:} \quad \{\text{LOCK:=1;}\} \\
4: & \quad \quad \text{got_lock:=freedom}; \\
& \quad \} \\
& \quad \ldots \\
5: & \quad \text{if} (\text{got_lock} \neq 0) \{ \\
6: & \quad \quad \text{unlock}(); \\
& \quad \quad \text{unlock:} \quad \{\text{LOCK:=0;}\} \\
7: & \quad \quad \text{got_lock:=freedom}; \\
& \quad \} \\
8: & \\
\end{align*}
\]
Step 2: Game

```
int[0,1,2] got_lock = 0;
int[0,1,2] freedom;
...
1: while(*) {
...
2:   if (*) {
3:     lock();
lock:  {LOCK:=1;}
4:     got_lock:=freedom;
}
...
5:   if (got_lock != 0) {
6:     unlock();
unlock: {LOCK:=0;}
}
7:   got_lock:=freedom;
...
}
8: 
```

Two types of non-determinism!
Step 2: Game

\[
\begin{align*}
\text{int}[0,1,2] & \text{ got\_lock} = 0; \\
\text{int}[0,1,2] & \text{ freedom}; \\
\ldots
\end{align*}
\]

1: while(*) {
   \ldots
2:   if (*) {
3:      lock();
lock:      {LOCK:=1;}
4:      got\_lock:=freedom;
   }
   \ldots
5: if (got\_lock != 0) {
6:   unlock();
unlock:      {LOCK:=0;}
7:      got\_lock:=freedom;
   \ldots
}
Step 3: Monitor for Acceptance

\[ P_1: \text{always}( \text{line}=\text{lock} \implies \text{next}( \text{line}! = \text{lock} \text{ w-until line}=\text{unlock} )) \]

Since game has two types of non-determinism, we need to be careful with non-determinism in monitor.
Problem with Nondeterminism

- Coffee machine is correct if there is no water or if button is pressed machine serves coffee:

  eventually always(not water) or
  always(pressed implies eventually coffee)
  and
  always(not water implies not coffee)

(Coffee machine wins if it visits a green state infinitely often)
Step 3: Det. Monitor for Acceptance

P1: always( line=lock implies next( line!=lock w-until line=unlock ))

Classical approach: make it deterministic (more powerful acceptance required)
Step 3: Product

TS for got_lock in \{0, 1\}

Deterministic automaton
Step 3: Produce
Step 4: Winning States

\[ l = \text{lock}, \quad gl = 0 \]
\[ l = \text{unlock}, \quad gl = 1 \]
Step 4: Winning States

The diagram illustrates the possible states and transitions for a system with conditions involving lock, unlock, and a variable $g$. Each state is represented by a node, and the transitions are indicated by arrows. The conditions at each state are specified, such as $l=lock$, $l=unlock$, and combinations of these with $gl=0$ or $gl=1$. Arrows between states indicate possible transitions based on these conditions.
Step 4: Winning States

![Diagram of winning states]
Step 4: Winning States

l = 1, gl = 0
l = 2, gl = 0
l = 3, gl = 0
l = lock, gl = 0
l = 4, gl = 0
l = unlock, gl = 0
l = 5, gl = 1
l = 6, gl = 1
l = 7, gl = 1
l = 8, gl = 0
l = 8, gl = 1

l = 4, gl = 0
l = 4, gl = 1
l = 4, gl = 2
l = 5, gl = 1
l = 5, gl = 2
l = unlock, gl = 1

l = 3, gl = 0
l = lock, gl = 0
Step 4: Winning Strategy

In general: strategy is function of program and monitor state

Strategy to Repair:
if (l=4 & gl=0 & s=1) freedom:=0
if (l=4 & gl=1 & s=1) freedom:=1
if (l=4 & gl=0 & s=0) freedom:=1
if (l=7 & gl=0 & s=1) freedom:=0
if (l=7 & gl=1 & s=1) freedom:=0
.
freedom := f(l,gl,s)
if (line=4) freedom := (gl=1) | (s=2)
if (line=7) freedom := 0

What we actually do: merge states before picking the strategy
Step 4: Winning Strategy
Step 4: Winning Strategy

(line=4): freedom = 1

(line=7): freedom = 0
unsigned int got_lock = 0;  
... 
1: while(*) {  
    ...  
2:   if (*) {  
3:      lock();  
4:      got_lock = 1;  
}  
    ...  
5:   if (got_lock != 0){  
6:      unlock();  
}  
7:   got_lock = 0;  
}  

lock() {  
lock: LOCK:=1;}  
unlock(){  
unlock: LOCK:=0;}

Specification  
P1: do not acquire a lock twice  
P2: do not call unlock without holding the lock  

P1: always( line=lock implies next( line!=lock w-until line=unlock ))  
P2: ( line!=unlock w-until line=lock ) and always( line=unlock implies  
    next( line!=unlock w-until line=lock ))

(slide adapted with permission from Barbara Jobstmann)
Recap: How to Repair a Reactive System?

1. Add freedom
   - choice for the system, space of permitted modifications to the system

2. Source code $\rightarrow$ transition system (game)
   - non-determinism in the program (demonic)
   - non-determinism in permitted modification (angelic)

3. Specification $\rightarrow$ monitor acceptance

4. Check if we can find system choices s.t. model is accepted by monitor
   - product of trans. system and monitor
   - search for winning strategy in game
Program Repair

- Program
- Finite-state program
- Monitor
- Specification

Game
- Game TS: program with freedom

Monitor TS: Winning condition

Solve game
- (Simple) Strategy
- Correct Program

with Bloem, Griesmayer, Staber in CAV 2005, CHARME 2005 (+ext to fault localization)
Classical Controller Synthesis

FSM + freedom + monitor

Initially defined for invariants

Game

Game TS + winning cond.

Solve game

(Simple) Strategy

Correct Program

Ramadge, Wonham 87, Book by Cassandras, Lafortune 99/07
Synthesis from Temporal Logics

Monitor + interface definition → Monitor TS: Winning condition → Solve game

Specification → Game

(Simple) Strategy → Correct Program

Church (1962), Büchi/Landweber (1969, games), Rabin (1972, trees), Pnueli/Rosner (1989, LTL)
Program Synthesis

Modern Controller Synthesis, see overview papers by Walukiewicz et al., Rutten & Girault, ...
Issues?

How to abstract?

Program

FSM + freedom

How to construct efficiently?

Monitor

Winning condition

How expressive?

Monitor TS

Size?

Game

Game TS

Solve game

How to pick a strategy?

Strategy

(Simple)

Correct

Program

How to map back?

Related research areas:
PL, AV, Control Theory,
Game and Automata Theory

How to specify?

Specification

How to abstract?
Issues with Monitor for LTL

- Determinization construction (Safra’s)
- 2EXP worst case complexity
  - LTL is very succinct

Monitoring TS:
- Winning condition
- How expressive?
- Size?
Some Solutions

- Concentrate on subsets (different types of games)
  - Ramadge, Wonham (Proc IEEE'89)
  - Asarin, Maler, Pnueli, Sifakis (SSC'98)
  - Alur, La Torre (LICS'01)
  - Alur, Madhusudan, Nam (BMC'03, STTT'05)
  - Wallmeier, Hütter, Thomas (CIAA'03)
  - Harding, Ryan, Schobbens (TACAS'05)
  - Jobstmann, Bloem (CAV’05)
  - Piterman, Pnueli, Sa'ar (VMCAI'06)
    (base of our work on synthesizing AMBA)

- Optimize or avoid determinization construction
  - Althoff, Thomas, Wallmeier (CIAA'05, TCS'06)
  - Piterman, Henzinger (CSL'06)
  - Kupferman, Vardi (FOCS'05)
  - Kupferman, Piterman, Vardi (CAV'06)
  - Schewe, Finkbeiner (ATVA'07), Filiot, Jin, Raskin (CAV'09)
  - Safety, Reachability Büchi, co-Büchi
  - Det. generators for several subsets
  - Safety+ using SAT, QBF, and BDDs
  - Request-Response
  - Work with nondet. automaton
  - Identified syntactic subset
  - Generalized Reactivity-1 (GR-1)
  - Implemented Safra
  - Improved Safra, Good-for-game
  - Bounded Synthesis (using co-Büchi)

- Symbolic representation (e.g., using BDDs)
Next

- More partial programs...
  - this time with Sketching
What is sketching?

- A program synthesis system
  - generates small fragments of code
  - checks their validity against a specification

- A programming aid
  - help you write tricky programs
  - cleverness and insight come from you
    - sorry, no programming robots
  - computer helps with low-level reasoning
The sketching experience

sketch

implementation (completed sketch)
Sketch language basics

- Sketches are programs with holes
  - write what you know
  - use holes for the rest

- 2 semantic issues
  - specifications
    - How does SKETCH know what program you actually want?
  - holes
    - Constrain the set of solutions the synthesizer may consider
Specifications

- Specifications constrain program behavior
  - assertions
    
    ```
    assert x > y;
    ```
  - function equivalence
    
    ```
    blockedMatMul(Mat a, Mat b) implements matMul
    ```

Is this enough?
Holes

- Holes are placeholders for the synthesizer
  - synthesizer replaces hole with concrete code fragment
  - fragment must come from a set defined by the user
Define sets of integer constants

Example: Hello World of Sketching

**spec:**
```java
int foo (int x)
{
    return x + x;
}
```

**sketch:**
```java
int bar (int x) implements foo
{
    return x * ??;
}
```

Integer Hole
Integer Holes → Sets of Expressions

- Example: Find least significant zero bit
  - 0010 0101 → 0000 0010
    
    \[
    \text{int } W = 32; \\
    \text{bit}[W] \text{ isolate}(\text{bit}[W] \times) \{ \quad // \text{W: word size} \\
    \text{bit}[W] \text{ ret} = 0; \\
    \text{for (int i = 0; i < W; i++)} \\
    \quad \text{if (!x[i])} \{ \text{ret}[i] = 1; \text{return ret; } \}
    \}
    
- Trick:
  - Adding 1 to a string of ones turns the next zero to a 1
  - i.e. 000111 + 1 = 001000

\[(x + ??) \& (x + ??) \]

→

\[\!(x + 1) \& (x + o)\]

\[\!(x + 0) \& (x + 1)\]

\[\!(x + 1) \& (x + oxFFFF)\]

\[\!(x + oxFFFF) \& (x + 1)\]
Example: Least Significant Zero Bit

0010 0101 → 0000 0010

```c
int W = 32;

bit[W] isolate0 (bit[W] x) { // W: word size
    bit[W] ret = 0;
    for (int i = 0; i < W; i++)
        if (!x[i]) { ret[i] = 1; return ret; }
}
```

```c
bit[W] isolateSk (bit[W] x) implements isolate0 {
    return !(x + ??) & (x + ??);
}
```
Integer Holes → Sets of Expressions

- Least Significant One Bit
  - 0010 0100 → 0000 0100

```c
int W = 32;

bit[W] isolateo (bit[W] x) {
  // W: word size
  bit[W] ret = 0;
  for (int i = 0; i < W; i++)
    if (x[i]) { ret[i] = 1; return ret; }
}
```

- Will the same trick work?
  - try it out
- Integer Holes → Sets of Expressions

- Expressions with `??` == sets of expressions
  - linear expressions  \[ x^{??} + y^{??} \]
  - polynomials  \[ x^{x^{??}} + x^{??} + ?? \]
  - sets of variables  \[ ?? ? x : y \]

- Semantically powerful but syntactically clunky
  - Regular Expressions are a more convenient way of defining sets
Regular Expression Generators

- `{ | RegExp | }`

- RegExp supports choice `'|' and optional `?'`
  - can be used arbitrarily within an expression
    - to select operands `{ | (x | y | z) + 1 | }`
    - to select operators `{ | x (+ | -) y | }`
    - to select fields `{ | n(.prev | .next)? | }`
    - to select arguments `{ | foo( x | y, z ) | }`

- Set must respect the type system
  - all expressions in the set must type-check
  - all must be of the same type
Least Significant One revisited

- How did I know the solution would take the form
  \( !(x + ???) \& (x + ???) \).

- What if all you know is that the solution involves \( x, +, \& \) and \(!\).

```c
bit[W] tmp=0;
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
{| x | tmp |} = {| (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
return tmp;
```

This is now a set of statements
(and a really big one too)
Sets of statements

- Statements with holes = sets of statements

- Higher level constructs for Statements too
  - repeat

```plaintext
bit[W] tmp=0;
repeat(3){
  \{| x | tmp |\} = \{| (!)?((x | tmp) (& | +) (x | tmp | ??)) |\};
}
return tmp;
```
repeat

- Avoid copying and pasting
  - repeat(n){ s}  \(\Rightarrow\) \(s; s; \ldots s;\)
  - each of the n copies may resolve to a distinct stmt
  - n can be a hole too.

```plaintext
bit[W] tmp=0;
repeat(??){
  { | x | tmp | } = { | (!)?((x | tmp) (& | +) (x | tmp | ??)) |};
}
return tmp;
```

- Keep in mind:
  - the synthesizer won’t try to minimize n
  - use --unrollamnt to set the maximum value of n
Example: logcount

```c
int pop (bit[W] x)
{
    int count = 0;
    for (int i = 0; i < W; i++) {
        if (x[i]) count++;
    }
    return count;
}
```
Procedures and Sets of Procedures

- 2 types of procedures
  - standard procedures
    - represents a single procedure
    - all call sites resolve to the same procedure
    - identified by the keyword `static`
  - generators
    - represents a set of procedures
    - each call site resolves to a different procedure in the set
    - default in the current implementation
Example

```c
int rec(int x, int y, int z){
    int t = ??;
    if(t == 0){return x;}
    if(t == 1){return y;}
    if(t == 2){return z;}
    if(t == 3){return rec(x,y,z) * rec(x,y,z);}
    if(t == 4){return rec(x,y,z) + rec(x,y,z);}
    if(t == 5){return rec(x,y,z) - rec(x,y,z);}
}

int sketch( int x, int y, int z ) implements spec{
    return rec(x,y, z);
}

int spec( int x, int y, int z ){
    return (x + x) * (y - z);
}
```
Step 1: Turn holes into special inputs

- The ?? Operator is modeled as a special input
  - we call them control inputs

```plaintext
bit[W] isolSk(bit[W] x)
{
    return ~(x + ??) & (x + ??);
}
```

- Bounded candidate spaces are important
  - bounded unrolling of `repeat` is important
  - bounded inlining of generators is important

```plaintext
{
    return ~(x + c1) & (x + c2);
}
```
Step 2: Constraining the set of controls

- Correct control
  - causes the spec & sketch to match for all inputs
  - causes all assertions to be satisfied for all inputs

- Constraints are collected into a predicate
  
  \[ Q(\text{in}, \text{c}) \]
int popSketched (bit[W] x)
    implements pop {
        loop (??) {
            x = (x & ??) 
            + ((x >> ??) & ??);
        }
        return x;
    }
Ex : Population count.

```c
int pop (bit[W] x)
{
    int count = 0;
    for (int i = 0; i < W; i++) {
        if (x[i]) count++;
    }
    return count;
}
```

\[ Q(\text{in}, c) \triangleq S(x, c) = \neg F(x) \]

\[ F(x) = \]
A Sketch as a constraint system

Synthesis reduces to constraint satisfaction

$$\exists c \forall x. Q(x, c)$$

Constraints are too hard for standard techniques

- Universal quantification over inputs
- Too many inputs
- Too many constraints
- Too many holes
Insight

Sketches are not arbitrary constraint systems
  - They express the high level structure of a program

A small set of inputs can fully constrain the soln
  - focus on corner cases

\[ \exists c \forall x \in E. Q(x, c) \] where \( E = \{x_1, x_2, ..., x_k\} \)

This is an inductive synthesis problem!
  - but how do we find the set \( E \)?
  - and how do we solve the inductive synthesis problem?
Step 3: Counterexample Guided Inductive Synthesis

Idea: Couple inductive synthesizer with a verifier

- Verifier is charged with detecting convergence

![Diagram]

- Inductive Synthesizer: Derive candidate implementation from concrete inputs
  \[ \exists c \forall x \text{ in } E.Q(x,c) \]

- Observation set E

- Standard implementation uses SAT based bounded checker
Inductive Synthesis

Deriving a candidate from a set of observations

\[ \exists c \forall x \text{ in } E \cdot Q(x, c) \text{ where } E = \{x_1, x_2, \ldots, x_k\} \]

Encode C as a bit-vector

- natural encoding given the integer holes

Encode Q(x_i, c) as boolean constraints on the bit-vector

Solve constraints using SAT solver

- with lots of preprocessing in between
Summary