From verification to synthesis

Verification:
  first write program (or model of a system), then specify formal properties, then check correctness.

Synthesis:
  first specify formal properties, then let synthesizer automatically generate a correct program.

Put another way:
  from imperative (how) to declarative (what) design; “raising the level of abstraction”.
What is synthesis?

Roughly:

\[ \exists P : \forall x : \varphi(x, P(x)) \]

Many different variants, depending on what is \( P \), \( \varphi \), and how search is done.

Very old topic (Church, 1960s) recently rejuvenated.

Program synthesis and proofs

From 2\textsuperscript{nd} order formula

\[ \exists P : \forall x : \varphi(x, P(x)) \]

to 1\textsuperscript{st} order formula

\[ \forall x : \exists y : \varphi(x, y) \]

Synthesizing program \( P \) can be done by proving \textit{constructively} that the above formula is valid.

\textbf{Deductive} program synthesis.
### Dimensions in Synthesis (Gulwani)

**Concept Language** (Application)
- Programs
  - Straight-line programs
- Automata
- Queries
- Sequences

**User Intent** (Ambiguity)
- Logic, Natural Language
- Examples, Demonstrations/Traces

**Search Technique** (Algorithm)
- SAT/SMT solvers (Formal Methods)
- A*-style goal-directed search (AI)
- Version space algebras (Machine Learning)

Also: logic synthesis

---

### Compilers vs. Synthesizers (Gulwani)

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Compilers</th>
<th>Synthesizers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept Language</td>
<td>Executable Program</td>
<td>Variety of concepts: Program, Automata, Query, Sequence</td>
</tr>
<tr>
<td>User Intent</td>
<td>Structured language</td>
<td>Variety/mixed form of constraints: logic, examples, traces</td>
</tr>
<tr>
<td>Search Technique</td>
<td>Syntax-directed translation (No new algorithmic insights)</td>
<td>Uses some kind of search (Discovers new algorithmic insights)</td>
</tr>
</tbody>
</table>
Designing controllers can be tricky and time consuming

Example: Electrical Power Generation and Distribution System (EPS) of a modern aircraft

Thanks to:
Pierluigi Nuzzo
Antonio Iannapollo
Designing controllers can be tricky and time consuming
Example: EPS requirements (in English)

Assumptions:

A2) At least one power source is always “healthy” (i.e. it is operational and can be inserted into the network to deliver power);
A3) Failures can only affect the power sources; once a power source becomes “unhealthy” (i.e. it is not operational and cannot be inserted into the network to deliver power), it will never return to be “healthy” (e.g., turned back on) during the cruising phase of the mission;
A4) An AC bus is correctly powered if the root-mean-square (RMS) voltage at its loads is between 110 V and 120 V and the frequency is 400 Hz.

Guarantees:

Under the above assumptions, the BPCU offers the following guarantees:

G1) At start-up all the power source contactors are “open”;
G2) In normal conditions (i.e. no faults or failures in the system) G_l and G_h are “on” and provide power for the left side and the right side of the system, respectively; auxiliary power units are “off”; C_4 and C_10 are open (“off”);
G3) No AC bus is powered by more than one power source at the same time, i.e. AC power sources can never be paralleled;
G4) It never happens that both the APU are inserted into the network at the same time;
G5) AC buses cannot be unpowered for more than a well-defined length of time;
G6) DC buses have to stay powered, at least in a “reduced performance” mode, which occurs when only one HVRU is used;
G7) The left AC bus B_l must always be powered from the first available source from the ordered list (G_1, A_1, A_2, G_3);
G8) The right AC bus B_r must always be powered from the first available source from the ordered list (G_10, A_10, A_11, G_1).

Designing controllers can be tricky and time consuming
Example: EPS requirements (in English) – zooming in

A2) At least one power source is always “healthy” (i.e. it is operational and can be inserted into the network to deliver power);

G1) At start-up all the power source contactors are “open”;

G3) No AC bus is powered by more than one power source at the same time, i.e. AC power sources can never be paralleled;
Designing controllers can be tricky and time consuming

Example: EPS “hand-written” controller
Designing controllers can be tricky and time consuming
Example: EPS “hand-written” controller
Design time ~ 1 week [Nuzzo] (but have to verify also)
For a real controller, it could be months [e.g., robotic controllers, Willow Garage]

Can design time be improved?

Declarative specification of controllers

At the outset the controller is just a box with inputs and outputs:
Declarative specification of controllers

At the outset the controller is just a box with inputs and outputs:

Example: EPS controller

We can specify the input-output behavior of the controller in a high-level language, e.g., in **temporal logic**.
Declarative specification of controllers

Example: LTL specification for EPS

~40 lines

# Assumptions

\[
\begin{align*}
& \neg \text{gl\_healthy} \land \neg \text{gr\_healthy} \land \neg \text{al\_healthy} \land \neg \text{ar\_healthy} \\
& \neg \text{gl\_healthy} \lor \neg \text{gr\_healthy} \lor \neg \text{al\_healthy} \lor \neg \text{ar\_healthy} \\
& \neg \text{gl\_healthy} \implies X(\neg \text{gl\_healthy}) \\
& \neg \text{gr\_healthy} \implies X(\neg \text{gr\_healthy}) \\
& \neg \text{al\_healthy} \implies X(\neg \text{al\_healthy}) \\
& \neg \text{ar\_healthy} \implies X(\neg \text{ar\_healthy})
\end{align*}
\]

# Guarantees

\[
\begin{align*}
& \neg \text{c1} \land \neg \text{c2} \land \neg \text{c3} \land \neg \text{c4} \land \neg \text{c5} \land \neg \text{c6} \land \neg \text{c7} \land \neg \text{c8} \land \neg \text{c9} \land \neg \text{c10} \land \neg \text{c11} \land \neg \text{c12} \land \neg \text{c13} \\
& \neg \text{c7} \land X(\text{c8}) \land X(\text{c11}) \land X(\text{c12}) \land X(\text{c13}) \\
& \neg (\text{c2} \land \text{c3}) \\
& \neg (\text{c1} \land \text{c5} \land (\text{al\_healthy} \lor \text{ar\_healthy})) \\
& \neg (\text{c4} \land \text{c6} \land (\text{al\_healthy} \lor \text{ar\_healthy})) \\
& (X(\text{gl\_healthy}) \land X(\text{gr\_healthy})) \implies X(\neg \text{c2}) \land X(\neg \text{c3}) \land X(\neg \text{c9}) \land X(\neg \text{c10}) \\
& (X(\neg \text{gl\_healthy}) \land X(\neg \text{gr\_healthy})) \implies X(\text{c9}) \land X(\text{c10}) \\
& X(\neg \text{gl\_healthy}) \implies X(\neg \text{c1}) \\
& X(\neg \text{gr\_healthy}) \implies X(\neg \text{c4}) \\
& X(\neg \text{al\_healthy}) \implies X(\neg \text{c2}) \\
& X(\neg \text{ar\_healthy}) \implies X(\neg \text{c3}) \\
& X(\text{gl\_healthy}) \implies X(\text{c1}) \\
& X(\text{gr\_healthy}) \implies X(\text{c4}) \\
& \ldots \\
& \neg \text{gl\_healthy} \implies X(\text{c5}) \\
& \neg \text{gr\_healthy} \implies X(\text{c6}) \\
& (X(\text{gl\_healthy}) \land X(\text{gr\_healthy})) \implies (X(\neg \text{c5}) \land X(\neg \text{c6})) \\
& \neg X(\text{gl\_healthy}) \land X(\text{gr\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy}) \\
& \neg X(\text{gl\_healthy}) \land X(\text{gr\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy}) \\
&& \implies (X(\text{c2}) \land X(\text{c3})) \\
& \neg X(\text{gl\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy}) \land X(\text{ar\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy}) \\
&& \implies X(\text{c2}) \\
& \neg X(\text{al\_healthy}) \land X(\text{c2}) \\
& X(\text{al\_healthy}) \land X(\text{c2}) \\
& \neg X(\text{ar\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy}) \\
&& \implies X(\text{c3}) \\
& (X(\text{gl\_healthy}) \land X(\text{gr\_healthy}) \land X(\text{al\_healthy}) \land X(\text{ar\_healthy})) \implies (X(\text{c2}) \land X(\text{c3}))
\end{align*}
\]
The controller synthesis problem

Given formula specification (e.g., in LTL) synthesize controller (e.g., FSM) which implements the specification (or state that such a controller does not exist).

Automatic controller synthesis from declarative specifications
Example: controller for EPS synthesized from previous LTL spec using Tulip (Caltech) ~3k lines of Matlab
Automatic controller synthesis from declarative specifications
Example: controller for EPS synthesized using Tulip (Caltech), ~40 states – zooming in

```
switch state
    case 1
        if isequal(g1_healthy, 1) && isequal(g2_healthy, 1) && isequal(g3_healthy, 1) && isequal(g4_healthy, 1)
            state = 0;
            c6 = 0;
            c3 = 0;
            c9 = 0;
            c8 = 0;
            c2 = 0;
            c13 = 0;
            c12 = 0;
            c11 = 0;
            c10 = 0;
            c7 = 0;
            c6 = 0;
            c5 = 0;
            c4 = 0;
        else
            disp('Cannot find a valid successor, environment assumption is likely to be violated');
            c6 = 0;
            c3 = 0;
            c9 = 0;
            c8 = 0;
            c2 = 0;
```

Synthesis in these two lectures

Part 1: Controller synthesis and game solving.

Part 2: Example-guided and syntax-guided synthesis.
Declarative specification of controllers

At the outset the controller is just a box with inputs and outputs:

We can specify the input-output behavior of the controller in a high-level language, e.g., in temporal logic.
Controller synthesis (reactive synthesis)
[Pnueli-Rosner, POPL 1989]

Given interface of controller:

\[
\begin{array}{c}
p_1 \quad \vdots \quad p_n \\
\quad \vdots \\
q_1 \quad \vdots \quad q_m
\end{array}
\]

and given temporal logic formula $\phi$ over set of input/output variables,

synthesize a controller (= state machine) $M$, such that all behaviors of $M$ (for any sequence of inputs) satisfy $\phi$.

Note: other notions of controller synthesis exist in the literature. See "Bridging the gap" paper on the course web site for details.

Examples

Consider controller interface:

\[
\begin{array}{c}
p \\
\rightarrow \\
\rightarrow \quad q
\end{array}
\]

and specifications

\[
\begin{align*}
\varphi_1 &= G(p \rightarrow Xq) \\
\varphi_2 &= G(p \leftrightarrow Xq) \\
\varphi_3 &= G(q \leftrightarrow Xp)
\end{align*}
\]
Examples

Consider controller interface:

\[
\begin{align*}
\varphi_1 &= G(p \rightarrow Xq) \\
\varphi_2 &= G(p \leftrightarrow Xq) \\
\varphi_3 &= G(q \leftrightarrow Xp)
\end{align*}
\]

No solution: controller cannot foresee the future!

Satisfiability vs. realizability

Satisfiability: exists some behavior that satisfies the specification. (In this behavior, we may choose both inputs and outputs as we wish.)

Realizability: exists controller that implements the specification. Must work for all input sequences, since inputs are uncontrollable.

Inherently different problems, also w.r.t. complexity:

LTL satisfiability: PSPACE
LTL realizability: 2EXPTIME
Controller synthesis algorithms: computing strategies in games

Solving safety games

Solving reachability games

Solving deterministic Büchi games (liveness)

Remarks on the general LTL synthesis problem
Safety automata

In some fortunate cases, the LTL specification can be translated to a safety automaton.

Example: \( \varphi = G(p \rightarrow q) \)

Automaton:

“Spreading” a safety automaton to a game


We need to separate the input moves from the output moves:

Automaton:

Game:
Safety games

Input (environment) states:

Output (controller) states:

Bad state:

Goal: find **winning strategy** = avoiding bad state

Solving safety games

1. Compute set of losing states, starting with $Losing := \{\bullet\}$.
2. If initial state in $Losing$, no strategy exists.
3. Otherwise, all remaining states are winning. Extract strategy from them by choosing outputs that avoid the losing states.
Solving safety games

1. Compute set of losing states, starting with \( \text{Losing} := \{ \square \} \);
   - repeat
     - \( \text{UncontrollablyLosing} := \{ s \mid s \text{ has uncontrollable succ in } \text{Losing} \} \);
     - \( \text{ControllablyLosing} := \{ s \mid \text{all controllable succs of } s \text{ are in } \text{Losing} \} \);
     - \( \text{Losing} := \text{Losing} \cup \text{UncontrollablyLosing} \cup \text{ControllablyLosing} \);
   - until \( \text{Losing} \) does not change;
Solving safety games

- Extracting the strategy: “cut” controllable transitions in order to avoid losing states.
- Strategy is **state-based** (also called “positional”, or “memoryless”).

Controller synthesis algorithms

Solving safety games

**Solving reachability games**

Solving deterministic Büchi games (liveness)

Remarks on the general LTL synthesis problem
Reachability games: dual of safety games

Reachability game: trying to reach a target state.

Observation: what is Losing for the safety player is Winning for the reachability player (and vice versa).

Solving reachability games: direct algorithm

1. Compute set of Winning states;
   - \( \text{Winning} := \{ \text{happy} \} \);
   - repeat
     - \( \text{Winning} := \text{Winning} \cup \text{ForceNext}(\text{Winning}) \);
   - until \( \text{Winning} \) does not change;
     - \( \text{ForceNext}(S) := \{ s \mid \text{all uncontrollable succs of } s \text{ are in } S \} \cup \{ s \mid s \text{ has controllable succ in } S \} \)
How to extract strategies in reachability games?

Similarly as for safety games:

Extract strategy from **ForceNext**(S): ensure you choose the right controllable transition that leads in winning state.

Is strategy state-based?
Yes!

---

How to extract strategies in reachability games?

Similarly as for safety games: **BUT**, a subtlety:

Need to fix successor the first time state is added in **Winning**.
Controller synthesis algorithms

Solving safety games

Solving reachability games

**Beyond safety and reachability games**

Remarks on the general LTL synthesis problem

---

What about other types of properties?

**Bounded response** specifications can be translated to safety automata/games:

\[ \varphi = G(p \rightarrow (q \lor Xq \lor XXq)) \]

Automaton:
What about liveness properties?

What about unbounded response?

\[ p \rightarrow q \quad \varphi = G(p \rightarrow Fq) \]

More interesting example:

\[ p_1 \rightarrow q_1 \quad \varphi = G(p_1 \rightarrow Fq_1) \land G(p_2 \rightarrow Fq_2) \land G(\neg(q_1 \land q_2)) \]

---

Synthesis for general LTL specifications

Given LTL specification \( \varphi \):

If \( \varphi \) can be translated to a deterministic Büchi automaton, then can extend the previous ideas to solving Büchi games.

Otherwise, solution involves more advanced topics, such as tree automata. Will not be covered in this course.

Note: LTL cannot always be translated to deterministic Büchi automata.
Büchi automata

Syntactically same as finite state automata:

\[ A = (\Sigma, S, s_0, \delta, F) \]

- **alphabet**
- **states**
- **initial state**
- **transition function**
- **accepting states:** \( F \subseteq S \)

But Büchi automata accept **infinite words**.
A run must visit an accepting state **infinitely often**.

From LTL to Büchi automata

Consider unbounded response property:

\[ \varphi = G(p \rightarrow Fq) \]

Büchi automaton:

```
\[ \bar{p} + q \quad \bar{q} \]
```

accepting state

\[ p\bar{q} \quad q \]
From LTL to Büchi automata

Consider LTL formula:

\[ \varphi = F G q \]

Büchi automaton? (s.t. **there exists** an accepting run)

Is there a deterministic Büchi automaton for this spec?

No!

Controller synthesis algorithms

Solving safety games

Solving reachability games

**Solving deterministic Büchi games (liveness)**

Remarks on the general LTL synthesis problem
Spreading Büchi automata to Büchi games

Büchi automaton:

Büchi game:

Solving deterministic Büchi games

1. Compute set of RecurrentAccepting states = accepting states from which controller can force returning to an accepting state infinitely often.
2. Solve reachability game with target = RecurrentAccepting.
Solving deterministic Büchi games

1. Compute set of **RecurrentAccepting** states = accepting states from which controller can force returning to an accepting state infinitely often.
   - $\text{RecAcc} := $ set of all accepting states;
   - repeat
     - $\text{Revisit} := \emptyset$;
     - repeat
       - $\text{Revisit} := \text{Revisit} \cup \text{ForceNext}(\text{Revisit} \cup \text{RecAcc})$;
       - until $\text{Revisit}$ does not change;
     - $\text{RecAcc} := \text{RecAcc} \cap \text{Revisit}$;
   - until set $\text{RecAcc}$ does not change;

Recall

**ForceNext**($S$) := \{ $s$ | all uncontrollable succs of $s$ are in $S$ \} $\cup$ \{ $s$ | $s$ has controllable succ in $S$ \}

![Diagram](image-url)
Solving deterministic Büchi games – Example

- $\text{RecAcc} := \text{set of all accepting states}$;
- repeat
  - $\text{Revisit} := \emptyset$;
  - repeat
    - $\text{Revisit} := \text{Revisit} \cup \text{ForceNext(Revisit} \cup \text{RecAcc})$;
    - until $\text{Revisit}$ does not change;
    - $\text{RecAcc} := \text{RecAcc} \cap \text{Revisit}$;
- until set $\text{RecAcc}$ does not change;

Computing recurrent accepting states: a subtle relation with reachability games

1. Compute set of $\textbf{RecurrentAccepting}$ states = accepting states from which controller can force returning to an accepting state infinitely often.
   - $\text{RecAcc} := \text{set of all accepting states}$;
   - repeat
     - $\text{Revisit} := \emptyset$;
     - repeat
       - $\text{Revisit} := \text{Revisit} \cup \text{ForceNext(Revisit} \cup \text{RecAcc})$;
       - until $\text{Revisit}$ does not change;
       - $\text{RecAcc} := \text{RecAcc} \cap \text{Revisit}$;
     - until set $\text{RecAcc}$ does not change;
Solving reachability games vs. computing Revisit

1. **Compute set of Winning states:**
   - \( \text{Winning} := \{ \} \);
   - repeat
     - \( \text{Winning} := \text{Winning} \cup \text{ForceNext}(\text{Winning}) \);
   - until \( \text{Winning} \) does not change;

2. **Compute Revisit:**
   - \( \text{Revisit} := \{ \} \);
   - repeat
     - \( \text{Revisit} := \text{Revisit} \cup \text{ForceNext}(\text{Revisit} \cup \text{RecAcc}) \);
   - until \( \text{Revisit} \) does not change;

What is the difference? Does it matter?

Solving deterministic Büchi games – modified example

\[ \begin{array}{c}
\begin{array}{c}
\pmb{1} \quad \pmb{2} \\
\quad \text{true} \\
\quad \text{p} \\
\quad \text{p} \\
\quad \overline{q} \\
\quad \text{q} \\
\quad \text{ok} \\
\quad \text{p} \\
\quad \text{true} \\
\end{array}
\end{array} \]

- \( \text{RecAcc} := \text{set of all accepting states}; \)
- repeat
  - \( \text{Revisit} := \{ \} \);
  - repeat
    - \( \text{Revisit} := \text{Revisit} \cup \text{ForceNext}(\text{Revisit} \cup \text{RecAcc}) \);
  - until \( \text{Revisit} \) does not change;
  - \( \text{RecAcc} := \text{RecAcc} \cap \text{Revisit}; \)
- until set \( \text{RecAcc} \) does not change;
How to extract strategies in deterministic Büchi games?

Similarly as for reachability games:

Extract strategy from **ForceNext**(S): ensure you choose the right controllable transition that leads in winning state.

Careful to choose the transition the first time state is added to S.

Is strategy state-based?

Yes!

What about non-deterministic Büchi games?

Does same algorithm work?

Not quite: algorithm sound but incomplete.

Non-deterministic automaton for \((GFr \land G\overline{g}) \lor (FG \overline{r} \land FG \overline{g})\)

Non-deterministic Büchi game

[Ruediger Ehlers, PhD thesis, 2013]
Controller synthesis algorithms

Solving safety games

Solving reachability games

Solving deterministic Büchi games (liveness)

Remarks on the general LTL synthesis problem

Controller synthesis: EE vs. CS?

CS: synthesize outputs to implement $\varphi$: $\square$

EE: synthesize inputs to stabilize a physical process/plant:

Not different: plant inputs = controller outputs (and vice versa).
Can we capture plants in the CS synthesis problem?

CS: given plant P (say, a FSM), synthesize controller C, so that closed-loop system satisfies $\varphi$:

Can we reduce this problem to the standard LTL synthesis problem?

Remarks, assessment

Despite some (mostly isolated) success stories, controller synthesis hasn’t really caught on yet in practice.

Why is that?

- Normal: things like that take time (c.f. model-checking)
- 2EXPTIME is a horrible (worst-case) complexity (remember: even linear is too expensive because of state explosion!)
- Tools still impractical
- Synthesis of real, complex systems from complete specs impractical (imagine full synthesis of complete Intel microchip from LTL specs …)
- Lack of good debugging (e.g., counter-examples)
- Need: better tools, better methods (incremental, interactive, …)
- Great opportunities for research!
References


PROGRAM SYNTHESIS
The “modern” approach to program synthesis

- Interactive:
  - computer-aided programming
  - programmer solves key problems (e.g., provides program skeleton), synthesizer fills in (boring or tedious) details (e.g., missing guards/assignments)

- Search-for-patterns based:
  - synthesis = search among set of user-defined patterns

- Solver based:
  - heavily uses verifiers like SAT and SMT solvers
  - often in a counter-example guided loop

Example: programming by sketching
[Solar-Lezama, Bodik, et al.]

Parallel Parking by Sketching
Ref: Chaudhuri, Solar-Lezama (PLDI 2010)

Enables programmers to focus on high-level solution strategy
Using SAT and SMT solvers for synthesis

Recall: what is synthesis?

\[ \exists P: \forall x: \varphi(x, P(x)) \]

Usually re-written as:

\[ \exists P: \forall x: pre(x) \rightarrow post(x, P(x)) \]

i.e., if input satisfies precondition, then output will satisfy postcondition.

Using SAT and SMT solvers for synthesis

\[ \exists P: \forall x: pre(x) \rightarrow post(x, P(x)) \]

Example of pre(), post():

\[ pre(x_1, x_2): \text{number}(x_1) \land \text{number}(x_2) \]
\[ post(x_1, x_2, y): x_1 \leq y \land x_2 \leq y \land (x_1 = y \lor x_2 = y) \]

i.e., the spec for max(x1,x2).
First: using SAT and SMT solvers for verification

Suppose we already have a program P.

Then instead of checking whether P is correct

\[ \forall x : \text{pre}(x) \rightarrow \text{post}(x, P(x)) \]

we can check whether P is wrong

\[ \exists x : \text{pre}(x) \land \neg \text{post}(x, P(x)) \]

i.e., we can check satisfiability of the formula

\[ \text{pre}(x) \land \neg \text{post}(x, P(x)) \]

Hold on: are programs formulas?

Consider a simple loop-free program:

```c
function P(int x) returns (real y)
{
    int tmp := 0;
    if (x >= 0) then {
        tmp++;
        y := tmp * x;
    }
    else
        y := -x;
    return y;
}
```

Formula:

\[ P(x, y) = (x \geq 0 \land y = x) \lor (x < 0 \land y = -x) \]
Hold on: are programs formulas?

What about real programs?

Loops, data structures, libraries, pointers, threads, …

Translation to formulas much harder, but verification tools are available that do this, constantly making progress.

We will assume we have a formula $P(x,y)$ representing the program $P$: “$y$ is the output of $P$ for input $x$”.

Back to using SAT and SMT solvers for verification

We can check satisfiability of the formula

$$pre(x) \land \neg post(x, P(x))$$

or, writing $P$ as predicate on both input and output variables:

$$pre(x) \land P(x, y) \land \neg post(x, y)$$

Satisfiable $\Rightarrow$ $P$ is wrong: we get a counter-example $(x,y)$

Unsatisfiable $\Rightarrow$ $P$ is correct (for all $x$)
Using SAT and SMT solvers for synthesis

What can be done when we don't have the program \( P \)?

\[
pre(x) \land P(x, y) \land \neg post(x, y)
\]

Hint: what if we have a finite/small number of candidate programs?

Iterate and search!

Programs with “holes”

Almost-complete programs:

```plaintext
Err = 0.0;
for(t = 0; t<T; t+=dT){
  if(stage==STRAIGHT){
    if(t+??) stage= INTURN;
  }
  if(stage==INTURN){
    car_ang = car_ang < ??;
    if(t+??) stage= OUTTURN;
  }
  if(stage==OUTTURN){
    car_ang = car_ang + ??;
    if(t+??) break;
  }
  simulate_car(car);
  Err += check_collision(car);
}
Err += check_destination(car);
```
Programs with “holes”

What should we replace “??” with?

Patterns:
- integer constants
- linear expressions of the form \( ax + by + c \) where \( x, y \) are variables in the program
  ...

Even with these restrictions, infinite set of candidates …
Search may take a long time or never terminate.
Can we do better?

Asking the solver to find the program

Suppose our program has 1 hole, to be filled with an integer variable.

Then, the formula characterizing the program becomes

\[
P(h, x, y)
\]

Can we use the solver to find the right \( h \)?

Check satisfiability of

\[
\forall x, y : \text{pre}(x) \land P(h, x, y) \rightarrow \text{post}(x, y)
\]
Problem: universal quantification …

\[ \forall x, y: \text{pre}(x) \land P(h, x, y) \rightarrow \text{post}(x, y) \]

Today’s solvers check satisfiability of quantifier-free formulas (mostly).

What can we do about that?

Hint: what if we have a finite number of **positive examples**? i.e., I/O pairs \((x, y)\) satisfying \(\text{pre}(x) \land \text{post}(x, y)\).

---

Example-guided synthesis

Suppose we have a finite number of positive examples, say 2: \((x_1, y_1), (x_2, y_2)\).

That is: we know that these hold:

\[ \text{pre}(x_1), \text{pre}(x_2), \text{post}(x_1, y_1), \text{post}(x_2, y_2) \]

So it suffices to check satisfiability of

\[ P(h, x_1, y_1) \land P(h, x_2, y_2) \]
Example-guided synthesis

In general, for n positive examples and k hole variables:

$$\bigwedge_{i=1}^{n} P(h_1, h_2, ..., h_k, x_i, y_i)$$

We turned universal quantification into finite conjunction!

Example-guided synthesis

What if solver finds this formula unsatisfiable?

$$\bigwedge_{i=1}^{n} P(h_1, h_2, ..., h_k, x_i, y_i)$$

Unsatisfiable => no program exists!

This is **sound**: if no program exists that works even in this finite set of examples, we cannot hope to find a program that works for all examples.
Example-guided synthesis

What if solver finds this formula **satisfiable**?

\[ \bigwedge_{i=1}^{n} P(h_1, h_2, ..., h_k, x_i, y_i) \]

**Satisfiable** \(\Rightarrow P(h_1, h_2, ..., h_k)\) is only a **candidate**.

It still needs to be **verified** for all I/O pairs.

We can again use the solver for that!

---

Example-guided synthesis

Satisfiable \(\Rightarrow P(h_1, h_2, ..., h_k)\) is only a **candidate**.

Verify it by checking satisfiability of

\[ pre(x) \land P(h_1, h_2, ..., h_k, x, y) \land \neg post(x, y) \]

These are now fixed

If formula is unsatisfiable then we are done!

What if formula is satisfiable?

Our candidate is wrong. We get a counter-example: \((x^*, y^*)\)

What then?
Adding negative examples to the synthesizer’s inputs

In general, for \( n \) positive examples, \( m \) negative examples, and \( k \) hole variables:

\[
\bigwedge_{i=1}^{n} P(h_1, h_2, \ldots, h_k, x_i, y_i) \land \bigwedge_{i=1}^{m} \neg P(h_1, h_2, \ldots, h_k, x_i^*, y_i^*)
\]

Alternative: the user could provide the correct output for the counter-example input, or we could use a reference (correct and deterministic) program.

Counter-example guided synthesis

<table>
<thead>
<tr>
<th>no program exists!</th>
<th>Synthesizer (may also use solver internally)</th>
<th>Verifier (e.g., SMT solver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail</td>
<td>succeed</td>
<td>spec, e.g., pre, post</td>
</tr>
<tr>
<td>succeed</td>
<td>program skeleton, initial set of examples</td>
<td>OK</td>
</tr>
<tr>
<td>counter-example</td>
<td>found correct program!</td>
<td>Not OK</td>
</tr>
<tr>
<td>((x_i^<em>, y_i^</em>))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EECS 144/244, UC Berkeley: 85

EECS 144/244, UC Berkeley: 86
References