Synthesis of Reactive Programs with Structured Latent State

Ria Das, Joshua B. Tenenbaum, Armando Solar-Lezama, Zenna Tavares

MIT CSAIL

Motivation

Background: Without being told the rules, children can figure out how a new toy or video game works—a full causal theory of which stimuli cause which changes—after just minutes of observation. This flexible, data-efficient ability to discover causal models has yet to be replicated in a machine. Existing approaches are either (1) not expressive enough to concisely capture the complexity of many real-world mechanisms (e.g. causal graphical models), (2) data-hungry (e.g. deep RL), or (3) assume more information in the input than raw observations (e.g. temporal logic constraints).

Approach: To overcome these limits, we frame the problem of discovering a causal model from observed data as one of *program synthesis*. We represent a causal model as a program in a domain-specific language (DSL), and exploit two key advantages of programs: their ability to compactly express complex designs, and that they can often be synthesized from small data.

We focus on the domain of time-varying, Atari-like grid worlds, and represent causal models using a language called Autumn. Discovering the causal structure underlying an observation sequence is equivalent to identifying the *program* in the Autumn language that generates the observations.

Challenge: How do we synthesize the **time-varying latent state** in a causal program? For example, in the Mario program (Figure 1), the agent (red) moves around with arrow key presses and can collect coins (yellow). If the agent has collected a positive number of coins, on a click event, a bullet (purple) is released upwards from the agent's position, and the agent's coin count is decremented. The number of collected coins is not displayed anywhere on the grid at any time, so the only way to write an Autumn program for Mario is to define a *latent* or *invisible* variable that tracks the number of coins. Existing methods cannot synthesize this from data.

Our Solution: We introduce a novel program synthesis algorithm, called AutumnSynth, that approaches this synthesis challenge by integrating standard methods of synthesizing functions with an *automata synthesis* approach, used to discover the model's latent state.

The Autumn Language

The Autumn language was designed to concisely express a rich variety of causal mechanisms in interactive grid worlds. Variables are defined as streams of values over time, via the syntax `var = init expr1 next expr2`, and a variable's previous value can be accessed using the primitive `prev`. For example, the agent object in Mario is defined with `agent = init (Agent (Position 15 7)) next moveDownNoCollision (prev agent)`, indicating that at every time, the agent should move down unless it would collide with another object. The default `next` behavior for a variable may be overridden using *on-clauses*, which are expressed as

```
on event
   update_function,
```

where `update_function` is a variable update of the form `var = expr`, and `event` is a Boolean which, when true, triggers the override.

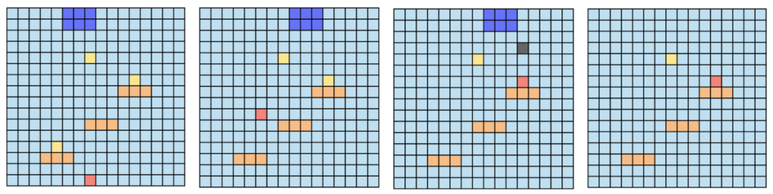


Figure 1: Stills (ordered but not consecutive) from Mario. The red agent initially cannot shoot bullets (purple), but after jumping and collecting coins (yellow), shoots on a click. The agent can no longer shoot when all its bullets are used up. A bullet kills the enemy.

The AutumnSynth Algorithm: An Overview

The AutumnSynth algorithm takes a sequence of grid frames and user actions as input and produces an Autumn program. Its four steps are

(1) **perception**, in which objects are parsed from the grid frames;

(2) **object tracking**, which assigns each object in a frame to either (a) an object in the next frame, deemed to be its transformed image, or (b) no object, indicating that the object was removed in the next time;

(3) **update function synthesis**, in which Autumn expressions, called update functions, describing each object-object mapping from Step 2 are determined; and

(4) **event synthesis**, in which Autumn predicates associated with each update function from Step 3 are sought, and new latent state in the form of automata is constructed upon search failure.

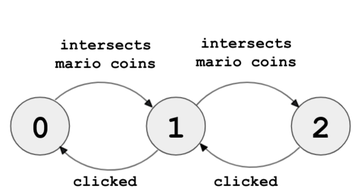
We illustrate the algorithm on the Mario program. First, the perception step identifies the objects in the model: the red agent, yellow coins, purple bullets, and blue enemy. Next, the tracking step identifies how these objects *change* across time, e.g. determining that the agent moves left between the first and second frames. Then, the update function synthesis step computes an Autumn expression that describes each change in Step 2, e.g. identifying that `agent = moveLeft agent` describes the agent moving left in the first time step. Finally, the event synthesis step searches for Autumn events that trigger each update function, i.e. evaluate to true at exactly the times that the update function takes place (e.g. `leftPress` above).

AutumnSynth: Learning Latent Automata

When a matching event cannot be found for an update function, we proceed by first finding the "closest" matching event, or the one that co-occurs with the update function but may also occur some other times. For bullet addition, this event is `clicked`, because every bullet is added on a click, but some clicks do not add a bullet. We then seek a `latentVar` such that

```
on clicked && (latentVar in [/* accept values */])
   bullets = addObj bullets (Bullet (prev mario).origin).
```

causes bullet addition to take place at exactly the observed times. This involves defining the initial value of the variable, as well as *transition* on-clauses that change the value in response to certain transition events. The desired `latentVar` definition is in Figure 2, along with the corresponding



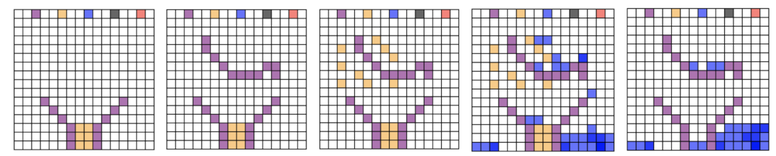

```
% initialize start state
numCoins : Int
numCoins = init 0 next (prev numCoins)
% state transitions
on intersects mario coins && (numCoins == 0)
   numCoins = 1
on intersects mario coins && (numCoins == 1)
   numCoins = 2
on clicked && (numCoins == 2)
   numCoins = 1
on clicked && (numCoins == 1)
   numCoins = 0
```

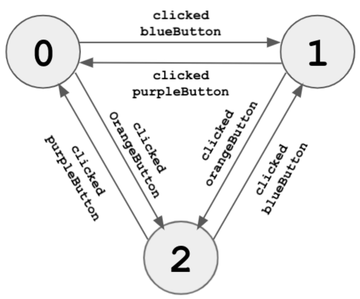
Figure 2: Automaton learned that tracks number of agent-collected coins.

automaton diagram. The accept values are 1 and 2, i.e. when the number of collected coins is positive. We synthesize this automaton using a heuristic algorithm based on state splitting.

Gallery of Synthesized Automata

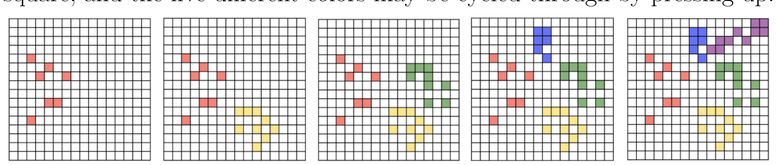
Water Plug: Clicking on an empty square adds a colored square. The color of the square depends on the last of the three leftmost buttons clicked in the top row.

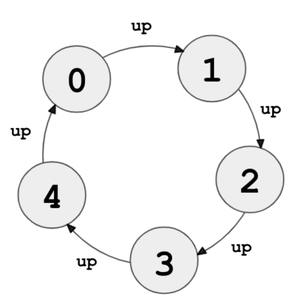





```
// initialize start state
active : Int
active = init 0 next (prev active)
// state transitions
on clicked blueButton
   active = 1
on clicked orangeButton
   active = 2
on clicked purpleButton
   active = 0
```

Paint: Inspired by MSFT Paint. Clicking an empty square adds a colored square, and the five different colors may be cycled through by pressing up.






```
% initialize start state
activeColor : Int
activeColor = init 0 next (prev activeColor)
% state transitions
on up && (activeColor == 0)
   activeColor = 1
on up && (activeColor == 1)
   activeColor = 2
on up && (activeColor == 2)
   activeColor = 3
on up && (activeColor == 3)
   activeColor = 4
on up && (activeColor == 4)
   activeColor = 0
```