Synthesis of Reactive Programs with Structured Latent State

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

MIT CSAIL

Motivation

Objective: To inductively synthesize functional reactive programs, i.e. from a finite data sequence.

Why does it matter? Reactive settings are plentiful in the real world (e.g. a robot or self-driving car operating on the street and updating its time-varying environment model, or a child learning how a video game works by watching for some time), but existing techniques do not learn these programs from data. Standard reactive synthesis inputs a logical formula and outputs an automaton. Programs are often more useful representations than automata, because large numbers of automaton states can be abstractly expressed in compact programs.

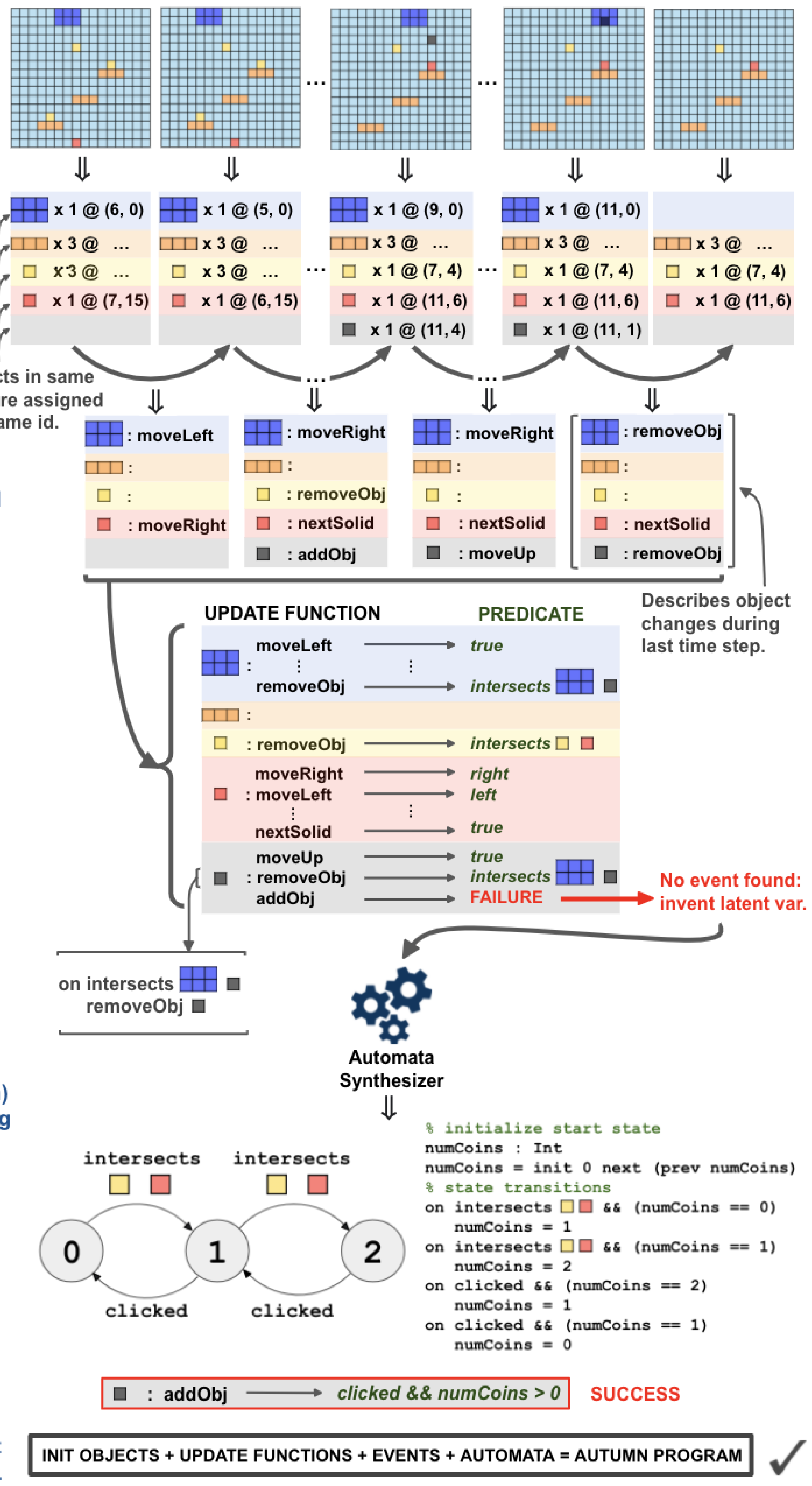
Why is it hard? While programs scale better and are more useful, standard methods for functional program synthesis cannot synthesize **time-varying latent state**, the core element of reactive settings. Precisely, functional synthesis expects its inputs and outputs to be fully observed, but both the inputs and outputs are *partially* observed in a reactive setting.

Our Solution: How can we inductively synthesize programs with time-varying latent state? Our approach is to **integrate functional and (inductive) automata synthesis**. We first try to synthesize the program using functional synthesis; if this fails, automata synthesis generates new latent state that then enables functional synthesis to succeed.

Methodology: We instantiate our algorithm in the domain of time-varying, interactive, Atari-like grid worlds, and write programs using a language called AUTUMN. An AUTUMN program defines *object* and *latent* (integer) variables, and describes grid-world dynamics using statements of the form `on event update`, where `update` changes a variable's value. Given a sequence of observed grid frames and user actions, we seek the program in the AUTUMN language that generates the observations. Concisely, we want to learn (latent) variables and *on-clauses*.

Running Example: In the Mario program (top-center column), the agent (red) moves around with arrow keys and collects coins (gold). If the agent has collected a positive number of coins, on a click event, a bullet (black) is released upwards, 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.

The AutumnSynth Algorithm: An Overview



INPUT
Sequence of grid frames and (not shown) user actions.

PERCEPTION & TRACKING
Objects are parsed from frames and mapped to objects in subseq. frames.

▦ x 1 @ (6, 0) | ▭ x 3 @ … | ☐ x 3 @ … | 🟥 x 1 @ (7, 15)

▦ x 1 @ (5, 0) | ▭ x 3 @ … | ☐ x 3 @ … | 🟥 x 1 @ (6, 15)

…

▦ x 1 @ (9, 0) | ▭ x 3 @ … | ☐ x 1 @ (7, 4) | 🟥 x 1 @ (11, 6) | ■ x 1 @ (11, 4)

…

▦ x 1 @ (11, 0) | ▭ x 3 @ … | ☐ x 1 @ (7, 4) | 🟥 x 1 @ (11, 6) | ■ x 1 @ (11, 1)

▭ x 3 @ … | ☐ x 1 @ (7, 4) | 🟥 x 1 @ (11, 6)

Objects in same row are assigned the same id.

UPDATE FUNCTION SYNTHESIS
Each object-object mapping is described with an Autumn expr. (*update function*).

▦ : moveLeft | ▭ : | ☐ : | 🟥 : moveRight

▦ : moveRight | ▭ : | ☐ : removeObj | 🟥 : nextSolid | ■ : addObj

▦ : moveRight | ▭ : | ☐ : | 🟥 : nextSolid | ■ : moveUp

▦ : removeObj | ▭ : | ☐ : | 🟥 : nextSolid | ■ : removeObj

Describes object changes during last time step.

EVENT SYNTHESIS
For each distinct update function in an object type, a trigger event is sought (i.e. one that is true at exactly the update func's times).

UPDATE FUNCTION	PREDICATE
▦ : moveLeft	*true*
⋮	⋮
removeObj	*intersects* ▦ ■
▭ :	
☐ : removeObj	*intersects* ☐ 🟥
🟥 : moveRight	*right*
moveLeft	*left*
⋮	⋮
nextSolid	*true*
■ : moveUp	*true*
removeObj	*intersects* ▦ ■
addObj	FAILURE

No event found: invent latent var.

on intersects ▦ ■
removeObj ■

AUTOMATA SYNTHESIS
When no matching event is found, new latent state (in the form of an automaton) is made so a matching event can be written. Here, we have named the latent variable *numCoins*.

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

■ : addObj → *clicked && numCoins > 0* SUCCESS

OUTPUT
Autumn program that creates observations.

INIT OBJECTS + UPDATE FUNCTIONS + EVENTS + AUTOMATA = AUTUMN PROGRAM ✓

AutumnSynth Algorithm: Automata Learning

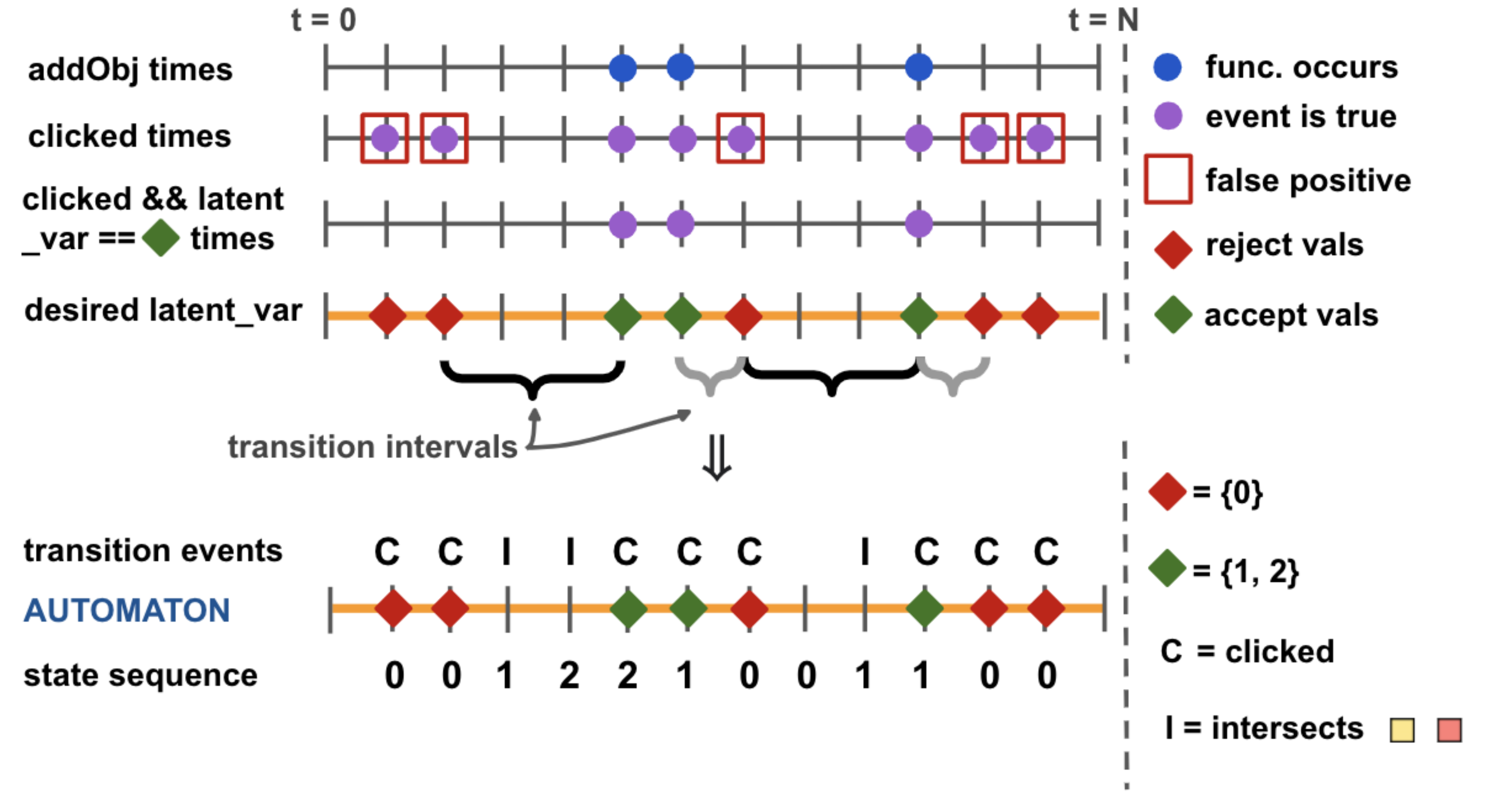
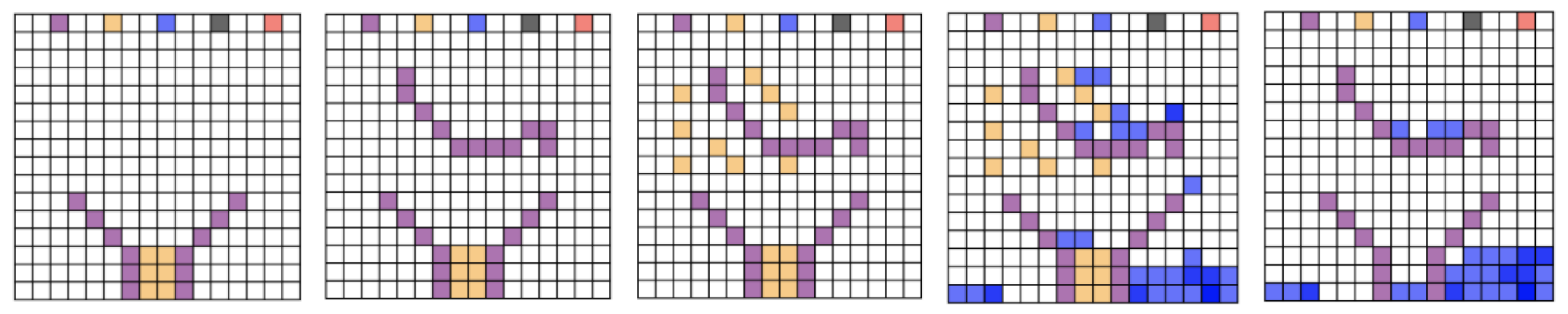
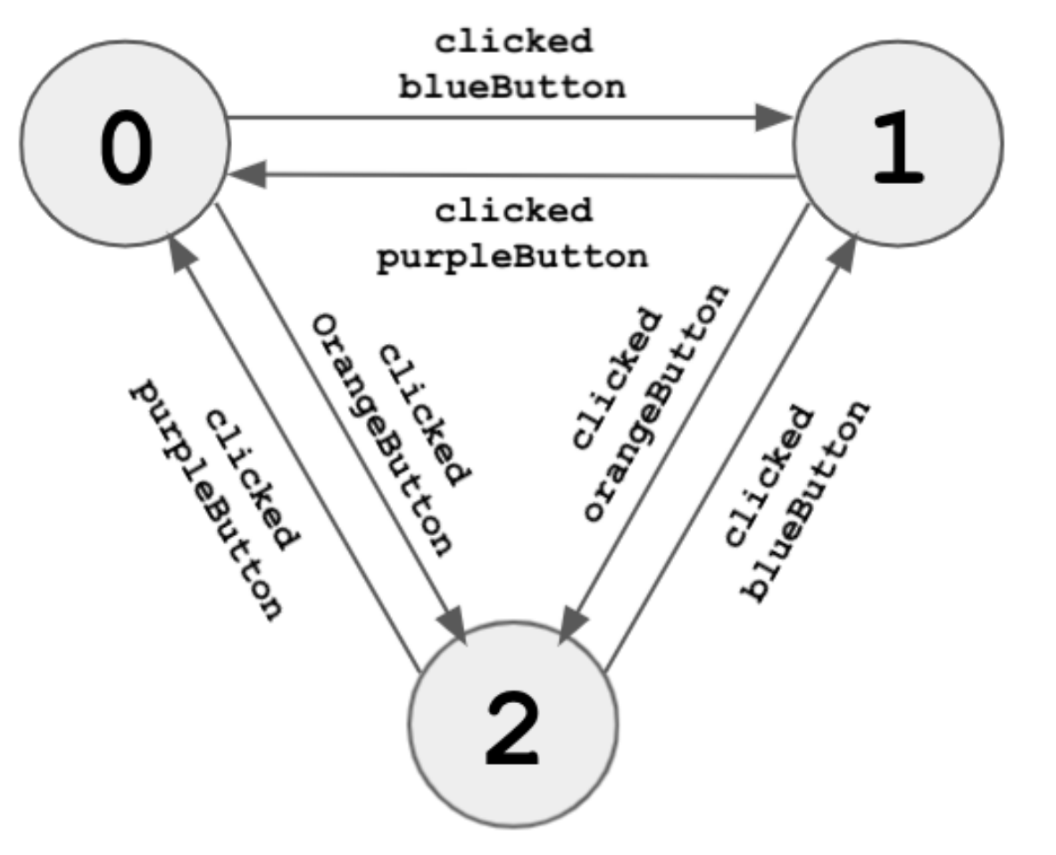


Figure 1: No event matches the `addObj` update function's times, but the "closest" match is the `clicked` event, which co-occurs with `addObj` but also occurs on false positive times. We coerce `clicked` into being a matching event by and-ing it with a predicate involving a new integer latent variable. This variable must take one set of values during the false positive times (indicated by red star), and another set during the true positive times (indicated by green star). Then, the event in the third row matches `addObj`'s times. To define this variable, we must find *transition events* that are true within the intervals between true and false times (false-to-true intervals are black, and true-to-false are gray). These transition events are `clicked` and `intersects`, corresponding to the edges in the automaton diagram in the previous column.

Example Synthesized Automata

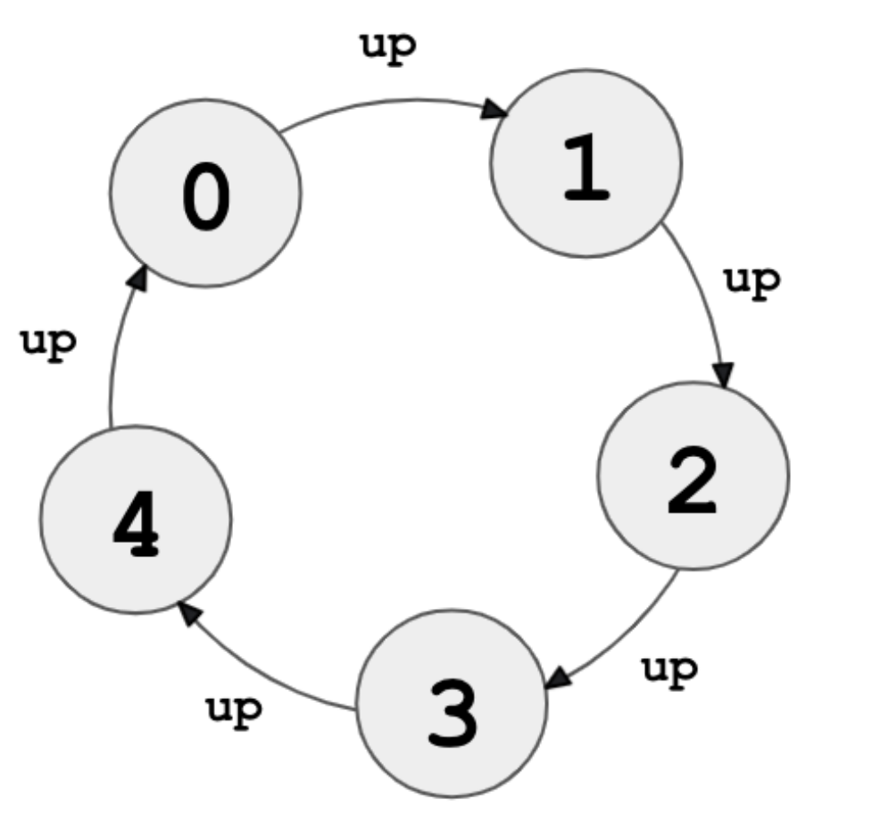
Water Plug: Clicking on an empty square adds a colored square. The square color depends on the last of the three leftmost buttons clicked.





```
// 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
```