Combining Functional and Automata Synthesis to Discover Causal Reactive Programs

Ria A. Das[1], Joshua B. Tenenbaum[2], Armando Solar-Lezama[2], Zenna Tavares[3]

[1]Stanford University, [2]MIT, [3]Basis and Columbia University

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 via observation), 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 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. Further, beyond just learning the values of the unobserved state at static times, reactive synthesis requires learning how the unobserved state dynamically *evolves* over time, in the form of program rules.

Approach

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

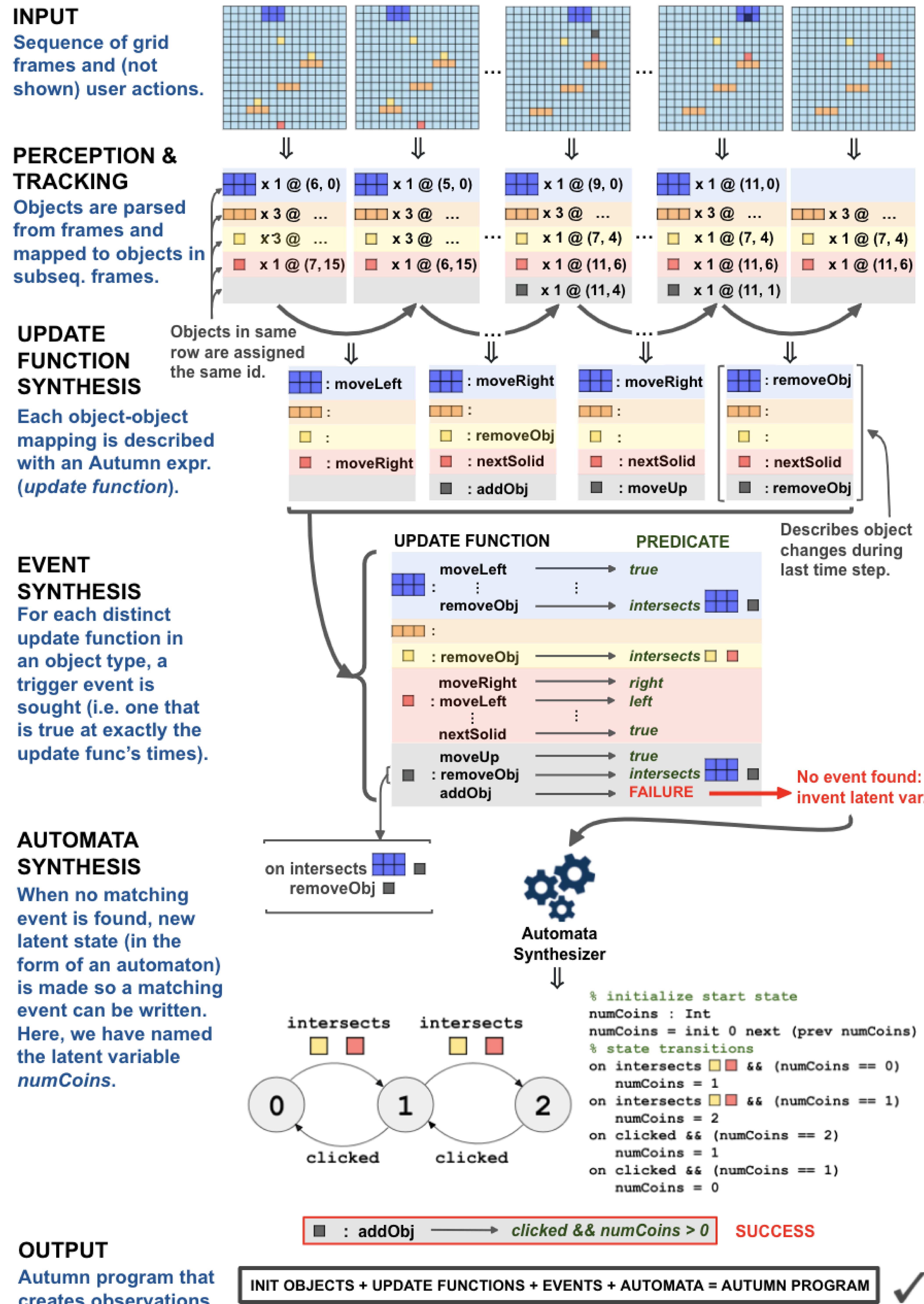
Methodology: We instantiate our algorithm in the domain of time-varying, Atari-like grid worlds, and write programs using a language called Autumn. An Autumn program defines *object* and *latent* (integer) variables, and describes grid-world dynamics using statements of the form `on event update`, where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

Running Example: In the Mario program (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, a bullet (black) is released, 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.

Evaluation

We evaluate our algorithm, named AutumnSynth, on a manually-constructed corpus of 30 Autumn programs that we call the Causal Inductive Synthesis Corpus (CISC), as well as a third-party suite of 27 grid-world-style video games called EMPA. For each benchmark, we manually constructed an input sequence of observed grid frames and user actions. In our preliminary results, we find that we can synthesize **27 of 30** CISC programs and **21 of 27** EMPA programs, where success indicates the synthesized program matches the given observations; see paper for details.

The AutumnSynth Algorithm: An Overview



AutumnSynth Algorithm: Automata Learning

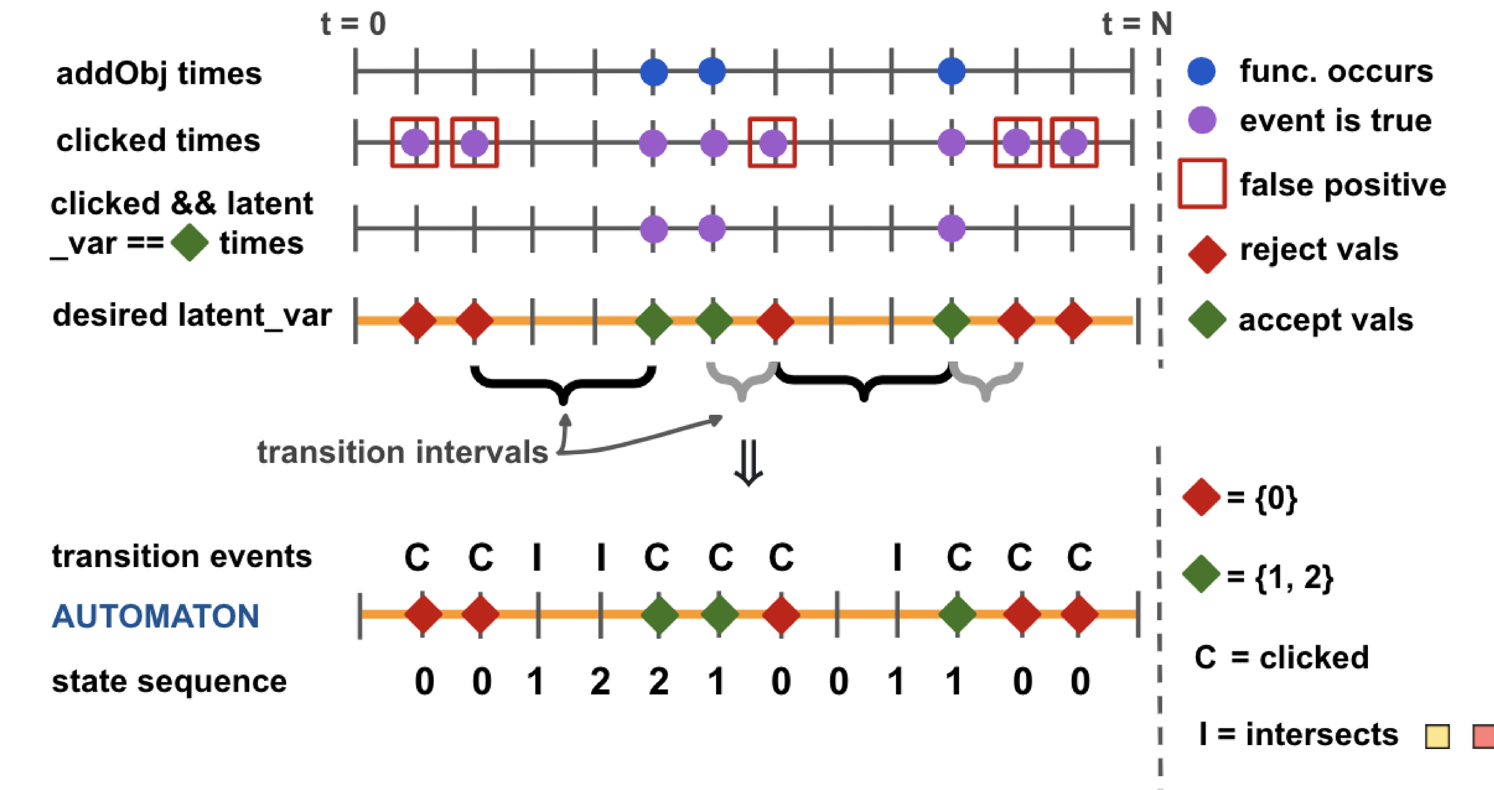


Figure 1: No event matches the `addObj` update function's times, but the "closest" match is the `clicked` event, which co-occurs with `addObj` but also occurs on false positive times. We coerce `clicked` into being a matching event by and-ing it with a predicate involving a new integer latent variable. 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 Benchmark Programs

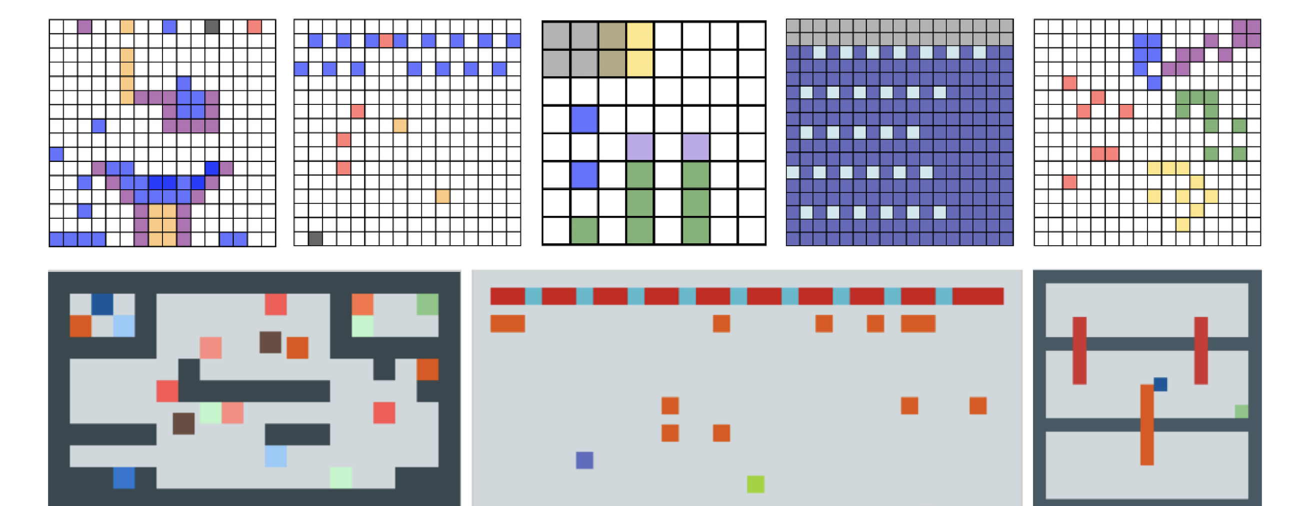


Figure 2: Top: Some CISC programs. Bottom: Some EMPA programs. Clockwise from top-left: Water interacting with a sink, Space Invaders, plants growing, snow blowing in the wind, MSFT Paint, gates that close and block path to goal, oscillating aliens, and portals that transport an agent to different-colored squares.

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.