Combining Functional and Automata Synthesis to Discover Causal Reactive Programs

RIA A. DAS, MIT, USA
JOSHUA B. TENENBAUM, MIT, USA
ARMANDO SOLAR-LEZAMA, MIT, USA
ZENNA TAVARES, Basis, USA and Columbia University, USA

We present a new inductive synthesis algorithm that synthesizes functional reactive programs from observation data. We apply our method to the domain of time-varying, Atari-style grid worlds, and develop a functional reactive DSL called Autumn that can express a rich variety of interesting causal dynamics in these environments. We evaluate our algorithm on a benchmark suite of 30 Autumn programs as well as a third-party corpus of grid-world-style video games. We find that our algorithm synthesizes 27 out of 30 programs in our benchmark suite and 21 out of 27 programs from the third-party corpus, including several programs describing complex latent state transformations, and from input traces containing hundreds of observations. Our results signal the promise of our formulation, and we expect that our algorithm will provide a template for how to integrate functional and reactive synthesis in inductive synthesis contexts more broadly.

Additional Key Words and Phrases: synthesis, automata, reactive, causal

1 INTRODUCTION

In the last decade, the traditional view of program synthesis as a technique for automating programming tasks has broadened due to the observation that programs compactly and interpretably represent a wide variety of structured knowledge, making programming languages powerful *model representations* in artificial intelligence systems [Ellis et al. 2021]. Program synthesis in this context is then not primarily concerned with improving programmer productivity, but instead captures a form of automated model discovery, with a number of recent advances in areas as diverse as learning programs describing biological data [Köksal et al. 2013], synthesizing computer-aided design programs from 3D meshes [Du et al. 2018], discovering phonological rules in linguistics [Ellis et al. 2015; Zuidema et al. 2020], and animal behavior modeling [Tjandrasuwita et al. 2021].

Much of this work at the intersection of program synthesis and AI can be framed as addressing the challenge of *theory induction*: Given an observation, what is the underlying *theory* or *model* that generates or explains that observation? We use *theory* to mean not just formal scientific theories, but also everyday cognitive explanations that humans derive on the fly to explain new observations [Gopnik and Wellman 2012; Ullman and Tenenbaum 2020]. For example, a child who has figured out how a new toy works after a few minutes of play has come up with a *theory* of the toy's mechanism. In particular, *causal theories*, which capture precise causal relationships between observations, are especially valuable as they can be used to predict how a system will react to future stimuli.

Unfortunately, existing methods of program synthesis are not yet suited to capture the rich space of theories that humans can learn from data, be they scientific or intuitive in kind. A critical source of difficulty is that many real-world phenomena are *reactive*, time-varying systems—they change dynamically in reaction to occurring events. However, current methods of inductive program synthesis—synthesizing programs from input-output examples—cannot synthesize non-trivial reactive models. This is because most reactive systems possess hidden, or *latent*, state that cannot be directly observed but nonetheless affects the dynamics of the system. Synthesizing *time-varying latent state*, the key step in learning any interesting reactive model, is a fundamental problem that standard inductive program synthesis techniques were not designed to handle.

Authors' addresses: Ria A. Das, MIT, USA, riadas@mit.edu; Joshua B. Tenenbaum, MIT, USA, jbt@mit.edu; Armando Solar-Lezama, MIT, USA, asolar@csail.mit.edu; Zenna Tavares, Basis, USA and Columbia University, USA.

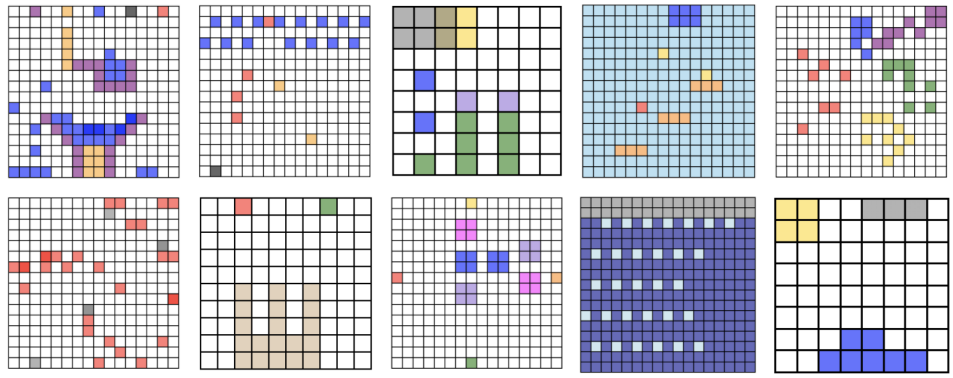


Fig. 1. A sample of Autumn programs. Clockwise from top-left: water interacting with a sink and sink plug a clone of Space Invaders, plants growing under sunlight and water, a simplified implementation of Mario, a simplified clone of Microsoft Paint, a weather simulation, snow falling left or right with varying wind, an alternative gravity simulation, a sand castle susceptible to destruction by water, and ants foraging for food.

By itself, reactivity would not pose a challenge if the state in question was fully observable. In that case, we could synthesize a *transition function* mapping observed input states to observed output states at every point in time, without regard to latent state that might exist as further unseen inputs at each step. There is prior work on synthesizing functions with unseen inputs in the context of *unsupervised learning* [Ellis et al. 2015], but the reactive setting is more challenging because of the need to discover how that latent state *evolves* over time, rather than just its static value.

In addition, there exists an extensive body of work on *reactive synthesis* that seeks to generate reactive programs with latent state, but much of this work focuses on synthesizing finite state programs from temporal logic specifications, rather than examples (e.g. [Bloem et al. 2012] or [Bansal et al. 2018]). Some work has been extended to infinite state systems [Beyene et al. 2014], and even to functional reactive programs [Finkbeiner et al. 2019]. Most recently, there have also been efforts to combine reactive synthesis with syntax-guided synthesis [Choi et al. 2022] to produce large programs with latent state. However, even this recent work takes as input logical specifications, so it does not address the problem of inducing such programs from examples on which we focus.

To address this gap between current inductive program synthesis approaches and the reactive setting, we develop a novel program synthesis algorithm that unites two largely orthogonal traditions within programming languages and formal methods: the *functional synthesis* and *automata synthesis* approaches. Specifically, we show that we can induce reactive programs by splitting synthesis into two procedures, a functional synthesis procedure, and an automata synthesis procedure. The functional synthesis step attempts to synthesize the parts of the program that do not depend on latent state. If functional synthesis fails to synthesize a program component explaining an observed output from observed inputs, our algorithm leverages automata synthesis to induce a *finite state automaton*, where the labels on the automaton transitions are predicates in the underlying domain-specific language used for synthesis. At a high level, based on the specifics of how functional synthesis failed, the automata synthesis procedure *enriches* the original program state with particular new latent structure that then allows functional step to succeed. This process happens modularly for each observed element described in the program, allowing our system to synthesize large programs with unbounded numbers of components, each with their own internal states.

We suspect that our approach to integrating functional and automata synthesis is valuable to a wide breadth of synthesis domains. In this paper, we demonstrate its value by instantiating it in a particular domain of interactive 2D grid worlds. While highly simplified from the real world, this

domain still spans a wide range of dynamic theories of interest in artificial intelligence, cognitive science, and other scientific disciplines, including those familiar from classic Atari-style video games and more recent physics-based games. Specifically, we have developed a functional reactive DSL called Autumn (from *automaton*) that is designed to concisely express the rich variety of causal dynamics within these grids (see examples in Fig. 1). The inductive synthesis problem addressed by our algorithm is: given a sequence of observed grid frames and corresponding user actions (clicks and keypresses), to synthesize the program in the Autumn language that generates the observations. The expressiveness of Autumn means that solving the problem of causal theory induction in the context of Autumn programs will be an important step towards the goal of learning causal theories in cognitive science and AI.

In summary, our paper makes the following contributions:

(1) We present a new functional reactive domain-specific language (Autumn) suitable for expressing and synthesizing non-trivial grid world programs.
(2) We introduce a new algorithm named AutumnSynth, that induces functional reactive programs from observation data. The algorithm has three variants that differ on the mechanism they use to solve the underlying automata synthesis problem.
(3) We introduce a benchmark suite of 30 Autumn programs which we call the *Causal Inductive Synthesis Corpus* (CISC), to spur the development of further algorithms in this space. The programs in this benchmark suite are designed to capture the diversity of time-varying causal models that may be manifested in 2D grids.
(4) We present an empirical evaluation of the scalability and expressiveness of AutumnSynth on both CISC and an externally-sourced dataset of 27 grid-world-style games written in a Python video game framework [Tsividis et al. 2021].

More broadly, we expect AutumnSynth will provide a template for how to integrate functional and reactive synthesis in the context of theory induction. In the rest of the paper, we provide a high-level overview of our work (Section 2), followed by an in-depth description of the algorithm (Sections 3 and 4) and details of the evaluation (Section 5).

2 OVERVIEW

In this section, we briefly describe the Autumn language and AutumnSynth algorithm and walk through a concrete execution of the algorithm on a video-game-inspired example.

2.1 Running example

As a running example, we use a simple program we call Mario, which is inspired by the popular video game. In this program, there is an agent representing Mario, which is rendered as a single red pixel. Mario can move left or right and can jump onto platforms in response to user keyboard commands. Mario can also collect coins, which are a different object type rendered as gold pixels.

As an interesting added twist, when the player clicks on the grid, Mario shoots a bullet upwards, but each bullet costs one coin, so Mario can only shoot if it has collected at least one coin, and if so, then shooting will decrement its coin count by one. Notably, the number of coins that Mario possesses is not displayed anywhere on the grid at any time; it is tracked by a scalar variable in the program. This creates a challenge for any synthesis algorithm trying to infer a program from a sequence of observations because the synthesizer has to infer the existence of this latent state to explain why sometimes clicking on the grid results in a bullet and other times it does not. At the top of the grid, an enemy object continuously moves between the left and right side of the frame and disappears if it is hit by one of Mario's bullets. Figure 2 illustrates a few steps of the game.

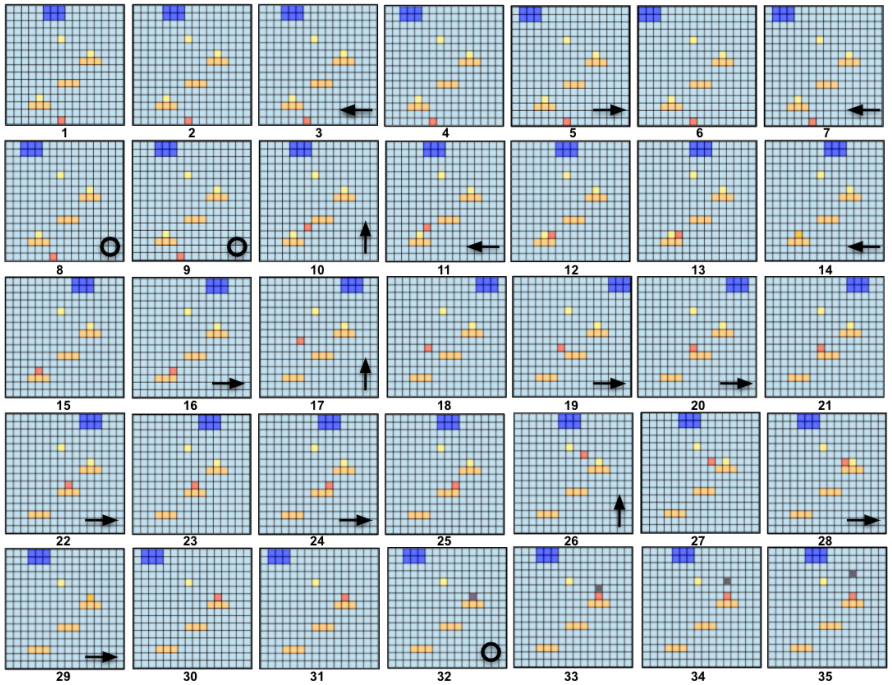


Fig. 2. An observation trace from the Mario program. Black arrows indicate user keypresses and circles indicate user clicks. An agent (red) moves around with arrow key presses and can collect coins (yellow). If the agent has collected a positive number of coins, when the human player clicks, a bullet (gray) is released upwards from the agent's position, and the agent's coin count is decremented. Otherwise, clicking does nothing.

2.2 The Autumn Language

Autumn is a *functional reactive* (FR) language designed to concisely express a rich variety of causal mechanisms in interactive 2D worlds. These mechanisms range from distillations of real-world, everyday causal phenomena, such as water interacting with a sink or plants growing upon exposure to sunlight, to video game-inspired domains such as Atari's Space Invaders or our Mario running example (see Fig. 1 for more examples). The design of Autumn was inspired by prior FR languages such as Elm [Czaplicki and Chong 2013], but differs from those languages in some important respects in order to make the synthesis process more tractable as explained in the rest of this section.

Every Autumn program is composed of four parts: Environment setup, Type definitions, Stream definitions and Event handling. The *Environment setup* defines the grid dimensions and background color for a program. *Type definitions* define object types; each object type has a *shape* represented as a list of 2D positions relative to the object center and each associated with a color, as well as a set of *internal fields* which store additional information about the object (e.g. a Boolean `healthy` field may store an indicator of the object's health). The object type definitions for the Mario program are shown below. In this case, only the Enemy object type has additional state, in the form of a Boolean field indicating whether the enemy's current direction of motion is left or right. The definitions of the other types only include the list of colors and 2D positions that define how the object is rendered. Every instance of an object (e.g. every Mario or every Coin) also has a 2D position without it needing to be declared in the type definition.

```
object Mario ( [(0, 0, red)] )
object Coin ( [(0, 0, gold)] )
object Platform ( [(-1, 0, orange), (0, 0, orange), (1, 0, orange)] )
object Enemy (movingLeft : Bool) ( [(-1, -1, purple), (-1, 0, purple),
                                    (0, -1, purple), (0, 0, purple),
                                    (1, -1, purple), (1, 0, purple)] )
```

The next part of a program consists of *Stream definitions*, which define object instances and other auxiliary values and their evolution over time in the absence of external events. For example, in the Mario program, we have four stream variables: one for Mario, one for the coins, one for the platforms, and one for the number of coins—the invisible latent state that tracks how many coins have been gathered and not used. Each of these streams is defined using the primitive Autumn language construct called *initnext*, which defines a *stream* of values over time using the syntax `var = init expr1 next expr2`. The initial value of the variable (`expr1`) is set with **`init`**, and the value at later time steps is defined using **`next`**. The **`next`** expression (`expr2`) is re-evaluated at each subsequent time step to produce the new value of the variable at that time. Within the **`next`** section of the stream definition, it is possible to access the previous value of the stream using the primitive **`prev`**. For example, below are some of the stream definitions in the Mario program:

```
mario : Mario
mario = init (Mario (Pos 7 15)) next (moveDownNoCollision (prev mario))
coins : List Coin
coins = init (list (Coin (Pos 4 12)) (Coin (Pos 7 4)) (Coin (Pos 11 6)))
        next (prev coins)
bullets : List Bullet
bullets = init (list) next (prev bullets)
numCoins : Int
numCoins = init 0 next (prev numCoins)
```

For the `mario` stream, the **`init`** section initializes the agent, and the **`next`** definition uses a user-defined function `moveDownNoCollision` which specifies that later values of the agent should move down one unit from the previous value whenever that is possible without collision. The `coins` stream illustrates that streams can also be lists of objects, not just individual objects. In the absence of other events, the list of coins will stay the same throughout the game, but shortly we will see the code that will make the list shrink as coins are collected by Mario. Similarly, the initial empty `bullets` list will grow as Mario shoots bullets and shrink if a bullet hits the enemy and is hence removed. Finally, the `numCoins` stream represents the latent state that tracks the number of coins collected by Mario. In general, any value that is not an object will be latent state since it will not be directly observable through the interface. Streams corresponding to latent state can be of primitive types `int`, `string`, or `bool` as well as lists of such values.

The fourth segment of an Autumn program is *Event handling* and is expressed using a construct called *on-clauses* which are expressed via the high-level form

```
on event
   intervention
```

where `event` is a predicate and `intervention` is one or more assignment of the form `var = expr` that override the default next value in the stream defined by the **`next`** clause in the Stream definitions section. For example, the code below shows some of the on-clauses of the Mario game.

```
on intersects (prev mario) (prev coins)
   numCoins = (prev numCoins) + 1
   coins = removeObj coins (-> obj (intersects (prev mario) (prev obj)))
on clicked && ((prev numCoins) > 0)
   numCoins = (prev numCoins) - 1
   bullets = addObj bullets (Bullet ((prev mario).origin))
```

The first on-clause indicates that when Mario intersects with a coin in the list of coins, the coin is removed from the list, and the number of coins is incremented. The second one indicates that when the grid is clicked and the number of coins is positive, the number of coins is decremented and a bullet is added at the current position of Mario. We note also that the coin removal syntax demonstrates that the `prev` function may be used not only to access the previous value of a stream but also the previous values of *individual objects* within a stream that is a list of objects (i.e. the use of `prev obj` instead of `prev coins`). Keeping track of object history in addition to stream history as such allows more fine-grained control over object dynamics. For example, it allows individual objects in a list to be modified or removed without resetting all other list objects to their previous values, as would happen if `removeObj coins` were replaced with `removeObj (prev coins)` above.

One important difference between Autumn and other FR languages like Elm is that on-clauses are evaluated sequentially, with the effect that later on-clauses may update a variable in a way that composes with updates from earlier on-clauses or completely overrides it. We found that for many programs, this led to significantly more concise programs, which made synthesis more efficient.

2.3 Synthesis Overview

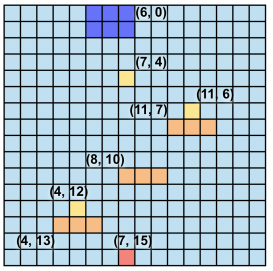
The AutumnSynth algorithm, is an end-to-end synthesis algorithm that takes as input a trace of a program—corresponding to a sequence of grids for a range of time steps and the corresponding user inputs for those steps—together with a library of language components. From these, the algorithm synthesizes an Autumn program using the given components that, when given the observed sequence of user inputs, matches the behavior observed in the trace. The algorithm consists of four distinct steps, each producing a new representation of the input sequence. These steps are:

(1) **perception**: object types and instances are parsed from the observed grid frames;
(2) **object tracking**: links objects in consecutive time steps to distinguish between (1) objects that moved or changed from one time step to the next, (2) objects that were created or destroyed and therefore lack a matching object in the preceding or subsequent timestep respectively;
(3) **update function synthesis**: *update functions*—Autumn expressions describing each object-object mapping from Step 2—are synthesized from the given components; and
(4) **cause synthesis**: Autumn events (predicates) that *cause* each update function from Step 3 are sought, and new latent state in the form of automata is constructed upon event search failure.

We give details for these steps in Section 4, focusing primarily on cause synthesis since that procedure represents the most novel aspect of our work. First, we provide some intuition by briefly describing how these steps are used to synthesize the Mario program from the running example.

2.3.1 Perception. The object perception step first extracts the object types and object instances from the input sequence of grid frames. For the example, the perception phase will identify three different object types: (1) a general single-cell type with a color parameter corresponding to the (red) agent, (yellow) coin, and (gray) bullet objects; (2) a platform type that is a row of three orange cells; and (3) an enemy type that is a rectangle of six blue cells. A list of object instances is extracted from each grid frame in the input sequence, where an object instance describes the object's type, position, and any

field values. For example, the object instances for the first grid frame in Fig. 2 (magnified on the right) are a red single-celled object (agent) at position (7, 15); three yellow single-celled objects (coins) at positions (4, 12), (7, 4), and (11, 6); three platform objects at positions (4, 13), (8, 10), and (11, 7); and an enemy object at position (6, 0). A few points to highlight are that, at this stage, the system does not know that the coins and the agent are different kinds of objects, as opposed to different colors for the same kind of object; also note that for multi-cell objects, the system assigns the object center to be the center of the pixel group. In addition, all Autumn objects are currently partially *transparent*, so occlusion is not a concern for the perception step.



2.3.2 Object Tracking. Next, the object tracking step determines how each object in each grid frame *changes* in the next grid frame. For example, it identifies that the agent object at position (7, 15) in the second grid frame corresponds to the agent object at position (6, 15) in the third grid frame (i.e. it moved left). Intuitively, this step *tracks* the changes undergone by every object across all grid frames. In our current implementation, both Perception and Tracking are heuristic-based.

2.3.3 Update Function Synthesis. This step derives an expression for each object in a given grid frame that describes the change in the object's visible attributes (position and color) in that time step. The expressions are synthesized using the components given as input to the algorithm. In our example, this step identifies that the expression `agent = moveLeft (prev agent)` accurately describes the change undergone by the agent object between the first and second grid frames from Fig. 2. Often, there are multiple such expressions that match any given mapping. For example, the agent's left movement during the first time step might also be described by `agent = moveLeftNoCollision (prev agent)` or `agent = moveClosest (prev agent) Platform`, where the latter indicates movement one unit towards the nearest object of type `Platform`. The update function synthesis step will not try to disambiguate among these options. Instead, it will return a set of possibilities to the subsequent *Cause Synthesis* step, which will be responsible for identifying the correct update function.

2.3.4 Cause Synthesis. Finally, the cause synthesis step searches for an Autumn event that triggers each update function identified in the previous step. For now, we will assume that update function synthesis produced a unique update function for each object at each time step; in Section 3 we will elaborate on the general case where this is not true. With this assumption, the goal is now to explain *why* each function was triggered when it was triggered. Autumn will first attempt to explain the triggering of each update function based on observable events, and then it will synthesize latent state to explain the triggering of any remaining updates that cannot be explained by the observable events alone. To find an Autumn event that triggers a particular update function, the algorithm collects the set of times that the update function is used and enumerates through a space of Autumn events until it finds one that evaluates to true at exactly those times. For example, say that the Mario object undergoes the update function `agent = moveLeft (prev agent)` at times 1, 4, and 5. If the Autumn event `left`, which indicates that a left keypress has occurred, evaluates to true at only those three times, then the on-clause

```
on left
   agent = moveLeft (prev agent)
```

accurately describes that particular update function's occurrence. The search space of Autumn predicates is defined over the *program state*, which consists of the current object instances, latent variables, and user events. At the start of this step in the algorithm, there are not yet any latent

variables in the program state, so the possible events use only the objects and user events (e.g. `clicked`, `clicked mario`, or `intersects bullet enemy`). Lastly, this event-finding process is simplified by the fact that on-clauses may override each other, so perfect alignment between the trigger event and observed update function is not always necessary. For example, even though `mario` does not undergo the update function `mario = moveDownNoCollision (prev mario)` at every time, the trigger event learned for this update function is simply `true`. This is because later on-clauses describing other behaviors like `moveRight` and `moveLeft` *override* the on-clause at appropriate times, so `mario` ends up undergoing `moveDownNoCollision` exactly when desired. Searching for a trigger event in the search space that exactly matches the times of `moveDownNoCollision`, in contrast, may be much more challenging or even impossible. This nuance will be explained in detail in Section 4.

The interesting case in the cause synthesis step is what happens when a matching Autumn event cannot be found for a particular update function. In the Mario example, this happens with the update function `bullets = addObj (prev bullets) (Bullet (mario.origin))`, which describes a bullet object being added to the list of objects named `bullets`. Bullet addition takes place at times 32, 41, and 57, but no event is found that evaluates to true at exactly those times. Since the existing program state does not give rise to any matching events, the algorithm must augment the program state by inventing a new latent variable that can be used to express the desired predicate.

Specifically, the algorithm proceeds by finding the "closest" event in the event space that aligns with the update function. This is the event that *co-occurs* with every update function occurrence, but may also occur during *false positive times*: times when the event is true but the update function does not occur. For bullet addition, this event is `clicked`, as every bullet is added on a click, but some clicks do not add a bullet. Having identified this closest event, our goal is then to construct a latent variable that acts as a finite state automaton that *switches* states between the false positive times and true positive times (i.e. the times when `clicked` is true and the update function occurs). To be precise, the new variable takes one set of values during the false-positive times, and another set during the true positive times. Calling the values taken by the latent variable during true positive times *accept values*, and those during the false-positive times *non-accept values*, the event

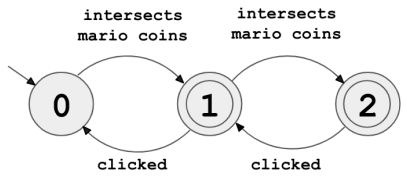
```
clicked && (latentVar in [/* accept values */])
```

perfectly matches the observed update function times. This is because `clicked` is true during a set of false-positive times, and `latentVar` is in *non-accept* values at exactly those times, so bullet addition does not take place, as desired. The full Autumn definition of `latentVar`, including the *transition on-clauses* that change its value over time, is shown in Fig. 3. The variable name `numCoins` is substituted to note the equivalence to a *number of collected coins* tracker. We note that this automaton—and all automata synthesized by our algorithm—are *finite-state* automata, even though it is possible to write an Autumn program describing infinite-state automata (e.g. if the Mario program in Section 2 described coins being added to the program at regular intervals instead of a fixed set of three coins, its counter variable would be infinite-state). In these infinite cases, our method synthesizes a finite-state *approximation* to the ground-truth automaton that suffices at explaining the given finite input trace.

The challenge in constructing this latent variable is learning the transition on-clauses that update the value of the variable at the appropriate times. Note that these transition on-clauses represent *edges* in the *automaton* diagrammed in Fig. 3 (hence the use of the term *accept values* or *states*). We perform the transition learning step as part of a general automaton search procedure, implemented via a SAT solver as well as heuristically, to be discussed in Section 4.

3 PROBLEM FORMALIZATION

In this section, we formally specify the synthesis task solved by the AutumnSynth algorithm.



(a)

```
% 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
```

(b)

Fig. 3. (a) Diagram of automaton representing the numCoins latent variable synthesized for the Mario program. The start value is zero, and the accept values (i.e. the values during which clicked causes a bullet to be added to the scene) are 1 and 2. (b) Description of the numCoins latent variable in the Autumn language. We note that the reason that the automaton states only go up to two collected coins instead of three is because the input sequence provided only demonstrates a maximum of two coins being collected at a time. We discuss this and give an example where the automaton has learned all three accept states instead in Section 5.

3.1 Modeling the Autumn language

We define the AutumnSynth algorithm in terms of a simplified model of the Autumn semantics. In this model, an Autumn program is represented as a tuple $\langle \mathcal{T}, \mathcal{X}, \mathcal{A}, \mathcal{O}, F, Z, X_0 \rangle$, defined below:

- **Object Types**: $\mathcal{T}$ is a set of *object types*, where each object type $t \in \mathcal{T}$ consists of a shape S and a list of additional data fields *data*, i.e. $t = \langle S, data \rangle$. A shape S is a set of cells, where each cell $c \in \mathbb{N} \times \mathbb{N}$ is a 2D position relative to center (0, 0). Each data field $f \in data$ is a a symbol n.
- **States**: Each state $X \in \mathcal{X}$ is a tuple $\langle X_o, X_l \rangle$, where X_o is a set of *objects* and X_l is a set of *latent variables*. Each object $x_o \in X_o$ is a tuple $\langle t, p, d \rangle$ where $t \in \mathcal{T}$ is an object type, $p \in \mathbb{N} \times \mathbb{N}$ is the origin of the object in the 2D grid, and d is the set of data values associated with the data fields of t. Each latent variable $x_l \in X_l$ is a value with integer type. Note that in this simplified model of the language, the different kinds of stream variables (objects, lists of objects, scalars, etc.) have been flattened to a simpler representation where all the objects belong to a single set and all the scalars belong to another. After a program has been synthesized in this representation, the code generator will be responsible for organizing the state into individual streams.
- **Actions**: The action space $\mathcal{A}$ is a set of three types of elements a: (1) arrow key presses (left, right, up, and down); (2) clicks on the observed grid, where each click is associated with a 2D grid position $p \in \mathbb{N} \times \mathbb{N}$; or (3) no action.
- **Observations**: The observations $\mathcal{O}$ are 2D grids of colored cells.
- **Transition Function**: Let $\mathcal{H} = \mathcal{X} \times \mathcal{A}$ be the space of *program histories*, where a program history is simply the current state and action at any given time. The transition function F is defined as a composition of a *next* function $next : \mathcal{H} \to \mathcal{X}$ corresponding to the next function in the stream definition and a set C of m *on-clause functions* $o_i : \mathcal{X} \times \mathcal{H} \to \mathcal{X}$. These functions are described in greater detail below:
  - *The next function.* The *next* function defines the "default" modification to the current set of state variables given history $H \in \mathcal{H}$. Note that in this simplified formalism, all the **next** clauses in the individual stream definitions are collapsed into a single *next* function applied to the entire state X.
  - *On-clause functions.* The on-clause functions are a set of m functions $o_1, \ldots, o_m$ where each o_i is constructed from a tuple $\langle \text{event}_i, \text{update}_i \rangle$. Each $\text{event}_i : \mathcal{X} \times \mathcal{H} \to \{0, 1\}$ is a Boolean predicate over the current program state and action, and each $\text{update}_i : \mathcal{X} \times \mathcal{H} \to \mathcal{X}$ is a

function that modifies the current state X given the same input.
From these event_i and update_i functions, each o_i is then

$$o_i(X, H) = \begin{cases} \text{update}_i(X, H) & \text{if } \text{event}_i(X, H), \\ X & \text{otherwise.} \end{cases}$$

Given these components, we define the transition function $F : \mathcal{H} \to \mathcal{X}$ to be

$$F(H) = o_m(\ldots(o_2(o_1(next(H), H), H)\ldots), H). \tag{1}$$

Note that each o_i has access to both the original state at the end of the previous timestep (which in the code can be accessed through `prev`), as well as to the new state as computed by `next` or by any previous on-clauses. This models the overriding behavior of on-clauses that was defined in Section 2.2.

- **Observation Function**: The deterministic partial observation function $Z : \mathcal{X} \to O$ renders the shape of each object $x_o \in X_o$ at the object's position in the 2D grid. Specifically, for an object $x_o = \langle t, p, d \rangle$ with $t = \langle S, data \rangle$, Z translates the shape S by the position p to obtain the observed rendering of x_o.
- **Start State**: The start state X_0 gives the set of objects and latent variables present at the start of the simulation.

3.2 Inference Task

Given a sequence of observations $(O_1, \ldots, O_T)$ and the corresponding sequence of actions $(A_1, \ldots, A_{T-1})$ from the decision process, our goal is to recover the object types $\mathcal{T}$, initial state X_0, and transition function F that correctly produces the observed data. These three components $(\mathcal{T}, X_0, F)$ specify an Autumn program, although there is still some additional work to do at that point to translate this program from this simplified formalism to the full Autumn syntax, for example by splitting the state and the *next* functions into individual stream definitions as explained in Section 4.6.

In general, the inductive synthesis problem is underdetermined, as there are many programs that will produce the correct observation sequence. Since it is challenging to identify whether a synthesized program is semantically equivalent to the ground-truth program from which the observation data was generated—especially since this generating program may be a black box—we define a *score function* to approximately measure closeness to the ground-truth.

4 SYNTHESIS ALGORITHM

We now give detailed descriptions of the steps of our algorithm introduced in Section 2. We focus on Steps 3 and 4—update function synthesis and cause synthesis—since the object perception and tracking steps use more standard techniques and are not a contribution of our work.

4.1 Step 1: Perception

For each observed grid frame $O_i \in \{O_1, \ldots, O_T\}$, the perception step produces a set $X_{o,i}$ of objects present in the frame, as well as a set of object types $\mathcal{T}$. Each object is a tuple $\langle t, p, d \rangle$ of an object type $t \in \mathcal{T}$, position p, and data values list d, where $t.data$ is either the empty set $\emptyset$ or the singleton set composed of the field $\langle color, \texttt{string} \rangle$. No other data fields beyond the observable color field are identified in this step. Latent data fields may be constructed in Step 4, upon which they are added as a modification to the existing type.

Our current Autumn implementation actually uses two different object parsing algorithms and runs the rest of the synthesis procedure on the result of each. The algorithm then returns the output program from the first parsing for which synthesis succeeds.

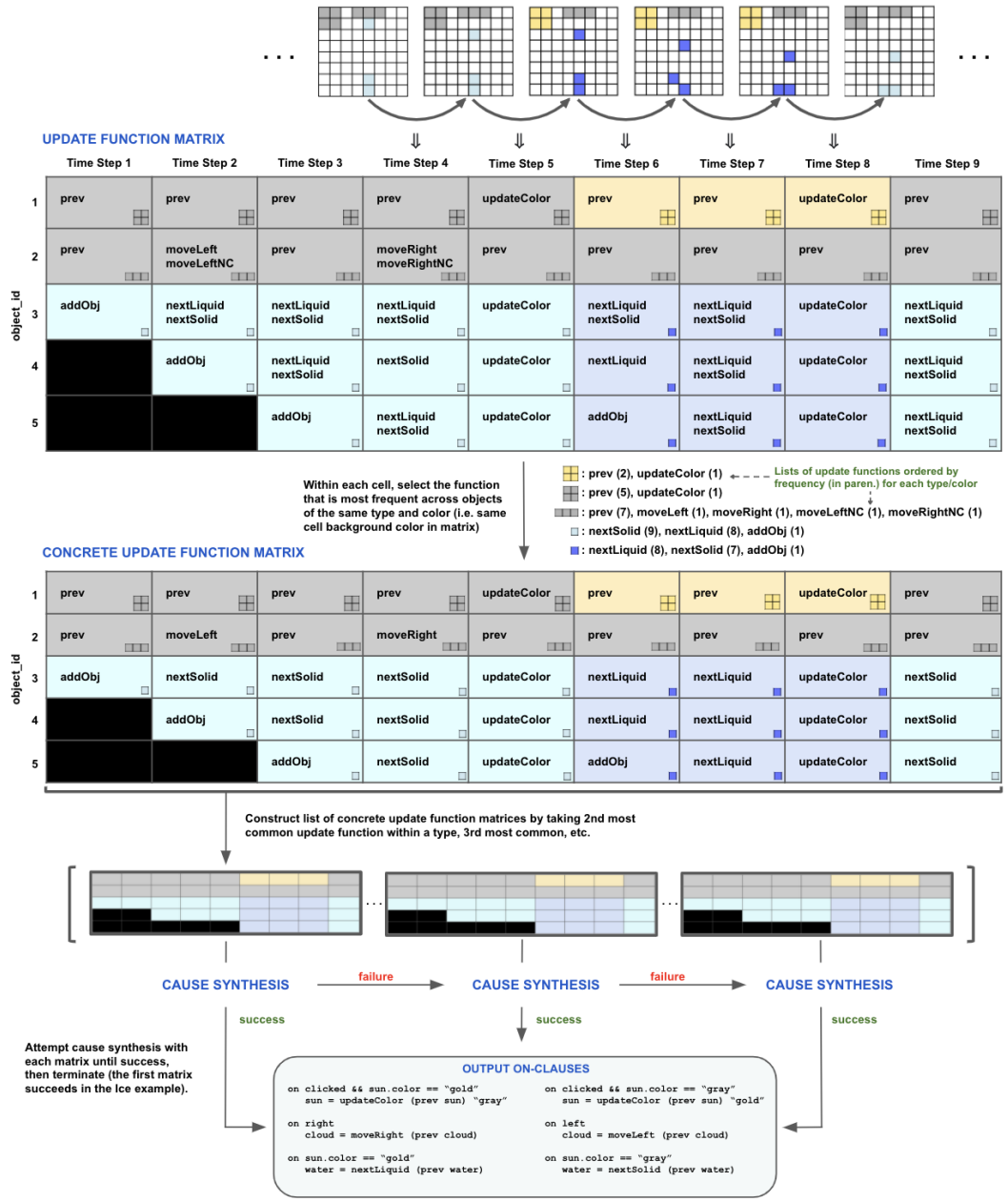


Fig. 4. Update Function Synthesis, demonstrated on the Ice benchmark program. In the ground-truth model, water particles are released from the cloud, and move according to the `nextLiquid` update function when dark blue in color (i.e. melted) and according to the `nextSolid` update function when light blue in color (i.e. frozen). Clicking toggles the sun between gold to gray and the water particles between light and dark blue. Each cell of the update function matrix contains a set of update functions that each describes the change undergone by the object with object_id equal to the row index during a particular time step (column index). A list of concrete update function matrices, with one update function per cell, is extracted via frequency-based heuristic.

The simplest of the two is called *single-cell parsing*, which identifies each colored cell in a grid frame as an individual object. The set of object types is then the set of single-celled shapes each with a particular fixed color observed across the grid frames. The other algorithm is called *multi-cell parsing*, which identifies groups of adjacent cells with the same color as multi-celled objects. This is necessary when the observed grid-frames contain groups of pixels that move together as single objects, but when a frame contains multiple single-celled objects of the same color, this algorithm runs the risk of interpreting them as one object when they are adjacent to each other.

Neither of the two object parsing algorithms is especially sophisticated, but they are sufficient to demonstrate our approach on the benchmarks.

4.2 Step 2: Object Tracking

Object tracking pairs objects in one frame to corresponding objects in the next frame. A *correspondence* m_t is a binary relation over two sets of objects X_t and X_{t+1} from consecutive frames. Intuitively, $(x_1, x_2) \in m_t$ denotes that x_1 and x_2 are the *same* object; $x \in X_{t+1} \setminus \text{domain}(m)$ denotes that x was *added* in the transition; $x \in X_t \setminus \text{codomain}(m)$ denotes that x was *removed*.

The algorithm that constructs the correspondence relations is a heuristic that is based on the assumption that objects are unlikely to move very far in a single time step. Hence, it attempts to maximally assign objects in frame t to their closest objects (by Manhattan distance) in frame $t + 1$. Objects that are remain unassigned this proximity heuristic are deemed to have been added or removed. The algorithm also ensures that no two objects in time step t are mapped to the same object in time step $t + 1$, and vice versa.

4.3 Step 3: Update Function Synthesis

Having tracked the objects in the program through time in Step 2, Step 3 synthesizes an Autumn expression, called an *update function*, that describes each object assignment, addition, or removal in each mapping relation m_i. These update functions are stored in a matrix that we call the *update function matrix*. We describe the construction of this matrix below.

From the sequence $\vec{X} = (X_1, \ldots, X_T)$ of objects produced by Perception, let $\mathcal{P}$ denote a set of unique object identifiers and $id(x) : X_i \to \mathcal{P}$ denote the mapping from objects to their unique identifiers that has two properties: (a) if $(x_a, x_b) \in m_t$ for some time t, then $id(x_a) = id(x_b)$, and (b) if $x_a \in X_t$ and $x_b \in X_t$ are two distinct objects in the same timestep t, then $id(x_a) \neq id(x_b)$.

Our goal is to construct an update matrix M, where $M_{i,t}$ is an Autumn expression defining how an object $x \in X_t$ with $id(x) = i$ transitions between time-steps t and $t + 1$. We will use the shorthand $M_{x,t}$ to mean $M_{id(x),t}$. Specifically, M is a $|\mathcal{P}| \times (T - 1)$ matrix, where each row contains the sequence of update functions undergone by the object with *id* corresponding to that row. To construct the update matrix, our method first constructs an *abstract update matrix* $\hat{M}$, where $\hat{M}_{x,t}$ is a set of candidate Autumn expressions such as `moveLeft` and `nextLiquid`, all of which denote object transformations. $\hat{M}$ is constructed such that each expression in $\hat{M}_{x,t}$ is consistent with $\vec{X}$. Formally, let $\mathcal{L}$ denote a set of Autumn expressions provided as input by the user to the algorithm, then $\hat{M}_{x,t} = \{f \mid f \in \mathcal{L}, (x, f(x)) \in m_t\} \cup Additions_{x,t}$. The final term of $Additions_{x,t}$ accounts for objects added or removed in a particular timestep.

$$Additions_{x,t} = \begin{cases} \{\texttt{addObj}\}, & x \text{ is added at time } t \\ \{\texttt{removeObj}\} & x \text{ is removed at time } t \end{cases}$$

4.3.1 Update Function Filtering. Our ultimate aim is to pair update functions with associated triggering events to generate on-clauses. Prior to this, the abstract update matrix $\hat{M}$, which describes

many possible programs, must be concretized into a concrete update matrix. A concrete update function matrix M is a "filtering" of an abstract update matrix $\hat{M}$ in the sense that $M_{x,t} \in \hat{M}_{x,t}$.

There are combinatorially many concrete matrices corresponding to any given abstract update function matrix, so we follow a set of heuristics to filter and sort the possible matrices. The heuristics are somewhat involved, but in this section, we provide the reader a sense of the main ideas behind them. The heuristics fall into four categories: (1) local filtering, (2) temporal filtering, (3) type consistency, and (4) type update function frequency.

Local filtering corresponds to local heuristics that are applied independently to every cell in $\hat{M}$ to remove low-probability update functions. *Temporal filtering* analyzes individual objects over time and prioritizes update functions that are consistent with what has happened at other times. This is particularly relevant for objects that undergo update functions such as `moveNoCollision` since it is often ambiguous whether the object is actually stationary (i.e. undergoing the update function **`prev`** `obj`) or is attempting to move but is blocked by a collision (i.e. undergoing `moveNoCollision`). When an object stops moving, the fact that it was moving in other time steps makes it more likely that a `moveNoCollision` function was actually involved, compared to if there was not such a history.

The next two categories of heuristics, *type consistency* and *type update function frequency* both build on the intuition that the algorithm should prioritize update functions that occur *more frequently* in $\hat{M}$ across all rows of the abstract matrix with the same object type. The idea is that selecting more frequent update functions will allow the generated code to "maximally share" update functions, resulting in fewer on-clauses for the final program. These heuristics are both illustrated in Figure 4.

Before we define type consistency and type update function frequency filtering, we introduce some notation. First, all objects in the program are grouped by type, so $\Gamma(x)$ is the type of object x. In most cases, the type will correspond to the object type defined earlier, but for objects that change color over time, the synthesizer may introduce distinct types for their different forms. For each type, we define an ordering among the update functions corresponding to that type as follows. Let $a, b \in \hat{M}_{x,t}$ be two update functions; then $a \leq_\tau b$ if and only if $count_\tau(a) \leq count_\tau(b)$ where:

$$count_\tau(u) = \sum_{t \in T, x \in \mathcal{P} \wedge \Gamma(x) = \tau} \mathbf{1}_{\hat{M}_{x,t}}(u)$$

where $\mathbf{1}_{\hat{M}_{x,t}}(u)$ is an indicator function producing a value of 1 if $u \in \hat{M}_{x,t}$ and 0 otherwise.

Based on this type-based ordering, we now define the *type consistency* requirement that all our concrete update matrices should satisfy. A matrix M is type consistent if it satisfies the following requirement:

$$\forall x, y \in \mathcal{P}, t_x, t_y \in T \text{ s.t. } \Gamma(x) = \Gamma(y) = \tau.\ a = M_{x,t_x} \wedge b = M_{y,t_y} \wedge a \neq b$$
$$\rightarrow (a = sup_{\leq_\tau}(\hat{M}_{x,t_x}) \wedge b \notin \hat{M}_{x,t_x}) \wedge (b = sup_{\leq_\tau}(\hat{M}_{y,t_y}) \wedge a \notin \hat{M}_{y,t_y})$$

In other words, type consistency enforces that all objects of the same type apply the same update function at every time step, with the exception that some object x may be different from the others if the update function that was chosen by others is not available for x at a given time step. In that case, x must chose the best update function available to it relative to the order defined earlier ($sup_{\leq_\tau}(\hat{M}_{x,t_x})$).

In the final step, *Type update function frequency* filtering, we first eliminate any matrix that unnecessarily uses any update function not among the top two based on the type update function ordering. More formally if $toptwo_\tau(\hat{M})$ returns the top two update functions based on the type update function order for a given type τ and a given matrix $\hat{M}$, then we filter away any matrix M that does not satisfy:

$$\forall x \in \mathcal{P}, t \in T, M_{x,t} \in toptwo_\tau(\hat{M}) \vee M_{x,t} = sup_{\leq_\tau}(\hat{M}_{x,t})$$

The matrices that pass this filtering are then sorted based on the partial order defined below. Let M^a and M^b be filterings of an abstract matrix $\hat{M}$. Then, $M^a \leq M^b$ iff

$$\forall x \in \mathcal{P}, t \in T. M^a_{x,t} \leq_\tau M^a_{x,t} \text{ where } \tau = \Gamma(x).$$

The end result of update function filtering is a list of update function matrices. All subsequent steps of the algorithm are run independently on each of these candidate matrices until synthesis succeeds.

4.4 Step 4: Cause Synthesis

By this stage in the algorithm, the object types, the object instances, and the possible update functions undergone by each object at every time have been identified. Remaining to be synthesized are the *event predicates* associated with the update functions in on-clauses, and potentially *latent variables*. At a high level, this step enumerates through concrete update function matrices $M_1, M_2, \ldots, M_n$, and searches for events that could have triggered each update function. If search succeeds for a given concrete matrix, the overall algorithm terminates, returning the final program. If it fails on the current concrete matrix, it is repeated on the next concrete matrix until success or until the end of the list is reached, which indicates overall synthesis failure.

To formalize the cause synthesis problem, we introduce the concepts of an *update function trajectory*, *event trajectory*, and a *match* between instances of the two. This step assumes that objects that belong to the same object type are controlled by the same set of on-clauses—if two objects of the same type both undergo `moveLeft` then a single event/on-clause was the cause. In contrast, if two objects undergo `moveLeft` and belong to different types, the method must synthesize a different event associated with each one. Thus, we synthesize events by enumerating through the object types and finding an event for each distinct update function that appears across objects of that type. Since this step of the algorithm is applied to each type independently, for convenience we shall assume that M contains objects of only a single type τ.

4.4.1 Event Space. The event space $\mathcal{E}$ is composed of conjunctions and disjunctions of a finite set of *atomic events*. There are two kinds of atomic events: *global* events and *object-specific* events.

Definition 4.1 (Global Event). A global event is a predicate over the program state, and may switch between occurring and not occurring as the program state evolves. Examples include user events such as `clicked`, `clicked obj1`, and `left` as well as object contact events like `intersects obj1 obj2` and `adjacent obj1 obj2`.

Definition 4.2 (Object-Specific Event). An object-specific event is a predicate on a single object. For example, `obj.color == "red"`. Object-specific events are used to apply update functions selectively to subsets of objects (those that make the predicate true) of a particular type.

The event trajectory for an event $e \in \mathcal{E}$ is its sequence of true/false values over time.

Definition 4.3 (Event Trajectory). For an event $e \in \mathcal{E}$, the event trajectory $E_{x,t}$ is constructed according to the following cases:

(1) Case 1: e is a global event. For all x, i.e., independent of x:

$$E_{x,t} = \begin{cases} 1 & e \text{ is true at time } t, \\ 0 & \text{otherwise.} \end{cases}$$

(2) Case 2: e is an object-specific event.

$$E_{x,t} = \begin{cases} 1 & \text{if e is true for object } x \text{ at time } t, \\ 0 & \text{otherwise.} \end{cases}$$

4.4.2 Update Function Trajectory. Informally, with respect to an update matrix M, the update function trajectory of an update function u is a matrix where each element is 0, 1 or $\frac{1}{2}$ to indicate whether u took place (1) or not (0) for object x at time t. The value $\frac{1}{2}$ represents uncertainty, indicating that u may have taken place but its effect could have been overridden by another update function. As described in Section 2.2, an update function can be overridden when there are other on-clauses that follow it in the Autumn program that are triggered at the same time. For synthesis, there is a choice of how to order on-clauses. With a similar rationale to the ordering of concrete matrices in the previous Section, our method orders update functions according to the ordering relation $\leq_\tau$, which favors update functions that occurred more frequently.

Definition 4.4 (Update Function Trajectory). Given a concrete update matrix M and an update function u, the update function trajectory $U_{x,t}$ is defined as:

$$U_{x,t} = \begin{cases} 1 & \text{if } M_{x,t} = u, \\ \frac{1}{2} & \text{if } M_{x,t} = u' \text{ where } u' \neq u \text{ and } u \leq_\tau u' \\ 0 & \text{otherwise.} \end{cases}$$

In other words, more frequent update functions appear earlier in the Autumn program than less frequent ones, and hence are overridden by those less frequent update functions when both are triggered at once.

4.4.3 Matching. Finally, an event is said to match an update function if their corresponding event and update function trajectories are the same over all objects at all times. If there is any ambiguity due to the overriding behavior, we tend towards being permissive in calling it a match. That is, the method counts any instance where $U_{x,t}$ is $\frac{1}{2}$ as a match, regardless of the event.

Definition 4.5. An event trajectory E *matches* an update function trajectory U if for all x, t, $E_{x,t} = U_{x,t}$, where for $a, b \in \{0, 1, \frac{1}{2}\}$, $a = b$ is true if a or b is $\frac{1}{2}$, and otherwise defined in the standard way.

If a matching event trajectory cannot be found for a particular update function trajectory, the algorithm moves on to the automata synthesis step, which attempts to augment the existing program state in such a way that a matching event may be written.

4.5 Step 4b: Automata Synthesis

Failure to find an event trajectory that matches an update function trajectory suggests that the domain of the event—the program state and known objects—may be missing something. That is, there may be some latent state, which if known, would allow our method to discover a matching event. The automata synthesis step discovers this latent state.

The input to the automata synthesis step is a set of update function trajectories, one for each unmatched update function from the previous step. The goal of the automata synthesis procedure is to construct the simplest latent state automaton that enables us to write matching latent-state-based event predicates. For ease of exposition, we will begin by describing the automata synthesis procedure for the scenario in which there is exactly one unmatched update function for which a latent-stated-based predicate must be constructed. We will then describe the extension to the more general scenario of multiple unmatched update functions.

4.5.1 Problem Formulation. To start, we formulate the problem of latent state synthesis within the classic formulation of automata synthesis given input-output examples, which aims to determine the minimum-state automaton that accepts a given set of accepted input strings (positive examples) and

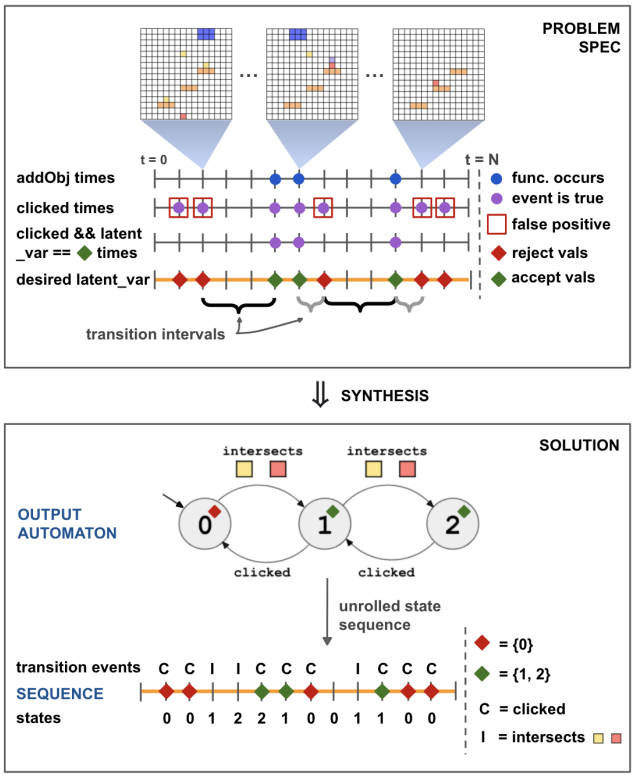


Fig. 5. Bird's-eye view of the automata synthesis problem, using the example of the Mario program. The bullet addition update function, indicated by `addObj`, does not have a matching trigger event. The closest event is `clicked`, which co-occurs with bullet addition but also is true at false positive times. We seek a latent variable that is true at one set of times (accept values) and false at another set of times (reject values), so that the conjunction of `clicked` and that latent variable perfectly matches `addObj`'s times. As shown in the solution, this latent variable initially has value zero, and changes to one then two on agent-coin intersection, and changes back down on clicks.

rejects a given set of rejected input strings (negative examples). In our scenario, we can construct positive and negative input "strings" from the sequence of program states.

To construct positive and negative examples, we consider the set of *prefixes*—sub-sequences of the program-state sequence, starting from the first position—that has, as their last element, a program state where the *optimal co-occurring event* is true. The optimal co-occurring event is the event that co-occurs with the update function in question and has the minimum number of false-positive times, i.e. times when the event is true but the update function does not occur. In the Mario example, this co-occurring event is `clicked`. Our method partitions the set of program state sequence prefixes into those that end with a program state in which the update function took place and those in which it did not. The former set is the set of positive examples and the latter is the set of negative examples for use in automata synthesis.

This construction of positive and negative input strings is motivated by the fact that, if there existed a latent state automaton that fit this specification, then the event

```
co_occurring_event && (latentVar in [/* accept values */])
```

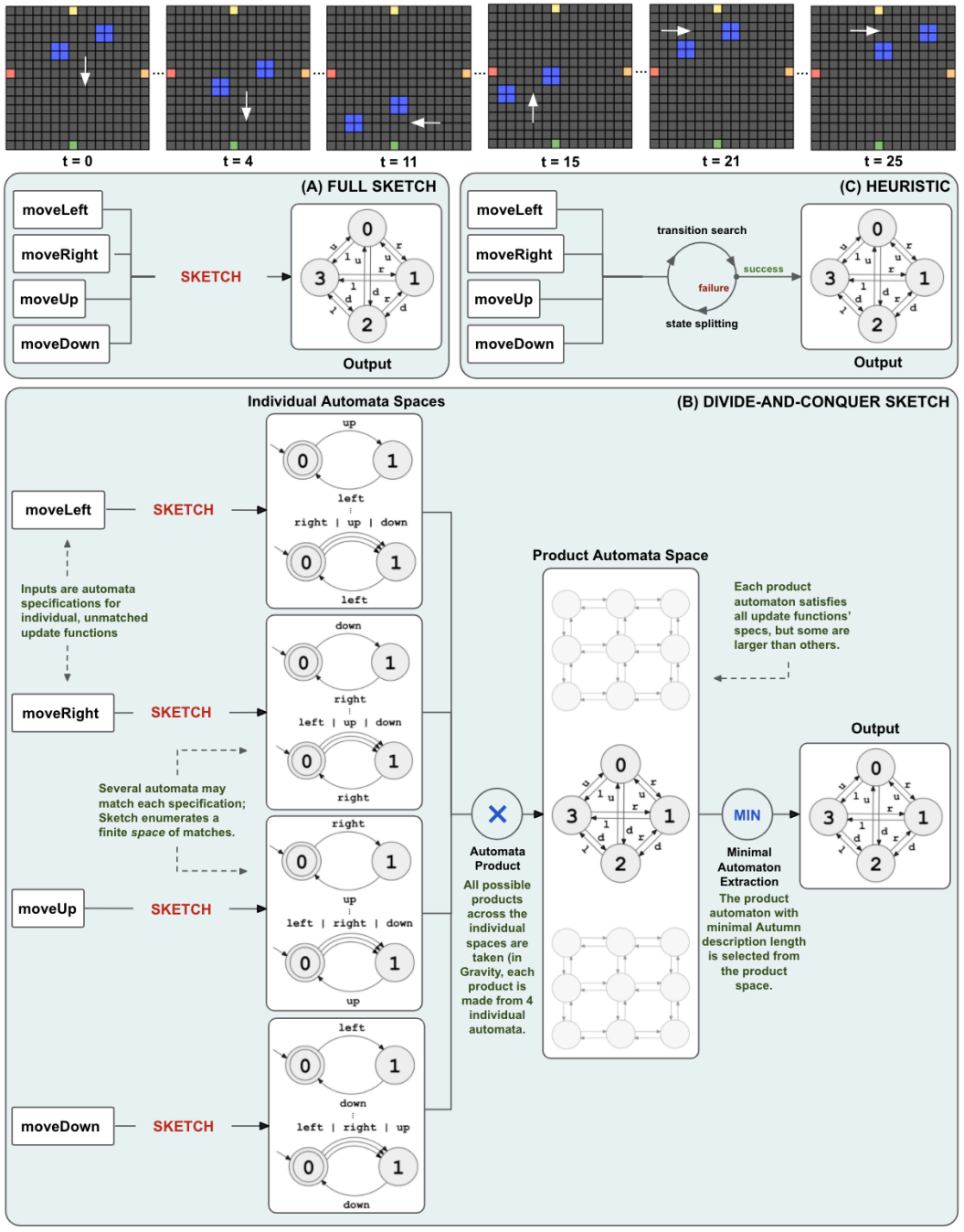


Fig. 6. Three variant methods for automata synthesis, shown for the Gravity I benchmark program. The blue blocks move left, right, up, or down depending on the button last clicked. The transition label `left` abbreviates `(clicked leftButton)`, etc.

would be a perfect match for the update function. This is because the co-occurring event is true during a set of false positive times with respect to the update function trajectory, and the latent automaton is in rejecting states at exactly those times (since those times correspond to the rejected program state prefixes). Thus, finding such an automaton would mean we would have an event that matches the update function under consideration.

4.5.2 Multiple unmatched update functions. The general setting has multiple unmatched update functions. In this scenario, each unmatched update function specifies its own inductive automata synthesis problem—a set of positive and negative input strings—that if solved will give rise to a matching latent-state-based predicate.

One solution to this "multi-automata" synthesis problem is to construct a distinct latent automaton (variable) that satisfies each update function. However, a smaller number of latent variables is often sufficient to explain all the update functions. In fact, the *product* of all the individual update function automata is a single automaton that satisfies all specifications, up to changing the accept states for each update function. However, taking the product of the simplest automata satisfying individual update functions—where we define *simplest* to mean that automaton with the fewest number of states—does not necessarily produce the simplest product automaton. This is because it is possible that larger automata for individual update functions will multiply to form the minimal product automaton instead. Thus, optimizing each update function's automaton individually and multiplying is not sufficient.

We now discuss three distinct algorithms for solving this inductive automata synthesis problem: Full Sketch, Divide-and-Conquer Sketch, and Heuristic. Our current implementation synthesizes a single latent state automaton that satisfies all unmatched update functions within each object type, as opposed to a single automaton for the entire program (i.e. across all object types), which mirrors the latent state structure found in most real Autumn programs in practice.

4.5.3 Algorithm 1: Full Sketch. In the Full Sketch approach, the complete multi-automata synthesis problem (for each object type) is encoded as a Sketch [Solar-Lezama 2013] problem. In other words, Sketch is tasked with identifying the minimal automaton that accepts each update function's language, as specified by the observed examples, up to changing just the accept states. As an example, consider the Autumn program named Gravity I shown in Figure 6. The blue blocks continuously move left, right, up, or down depending on which of the four colored buttons was last pressed. A matching event cannot be found for any of the four update functions `moveLeft`, `moveRight`, `moveUp`, or `moveDown`, so their update function trajectories are fed to the Sketch solver to produce the 4-state automaton shown in Figure 6a. This new latent variable then allows a matching predicate to be written for each of the four update functions: `true && latentVar == 1`, `true && latentVar == 2`, `true && latentVar == 3`, and `true && latentVar == 4`, where the optimal co-occurring event is `true`.

4.5.4 Algorithm 2: Divide-And-Conquer Sketch. Rather than attacking the full multi-automata synthesis problem, Divide-And-Conquer Sketch tasks Sketch with solving each update function's automata synthesis problem *individually*, and then combines those solutions together via product. The intuition behind this approach is that synthesizing an automaton matching *all* update functions at once may face scalability challenges, but finding an automaton matching a single update function, which is likely smaller, may be easier. As described previously, the smallest automaton satisfying a single update function may not give rise to the smallest product, so the Divide-and-Conquer algorithm identifies a small *set* of automata matching each update function instead. It then takes the product over all update functions' automata sets, and computes the minimal automaton from that product space. We illustrate this algorithm again with the Gravity I example (Figure 6b). The

algorithm first identifies a set of automata that solve the automata synthesis problems corresponding to the four unmatched update functions. Note that each of these automata have just two states instead of the full 4-state solution found in the Full SAT approach. Next, it computes all automata *products* over these four automata sets, and takes the minimal automaton from this product set, which is the 4-state solution seen previously.

4.5.5 Algorithm 3: Heuristic. Despite the simplicity of the Sketch-based formulations of automata synthesis, their scalability to problem settings with large automata is unclear, due to the scaling limitations of SAT solvers. As such, we also implemented a heuristic algorithm that synthesizes an automaton satisfying a set of update function trajectories via a series of greedy updates to an initial automaton (Figure 6c). At a high level, this approach begins with an automaton with a small number of states, and repeatedly *splits* states into two based on a heuristic related to the search for transition events. More precisely, the algorithm begins by searching for transition events (edges) that result in an automaton that produces a particular initial state sequence that has few distinct states. If the transition search fails, one of the original states is split into two, and the transition search is repeated. This process continues until a satisfying automaton is identified. Our heuristic algorithm bears some similarity to *counterexample-guided abstraction refinement*, in that it iteratively *refines* a specification in response to errors while trying to satisfy it; see Appendix A.2 for further detail.

4.5.6 Non-Determinism Handling in AutumnSynth. Having described the final cause synthesis step, we briefly comment on support for writing *nondeterministic* programs in Autumn. Autumn provides a library function called `uniformChoice`, which selects one element uniformly at random from a non-empty list. Using this function, many interesting causal probabilistic Autumn programs can be written. However, inferring the probability distribution described by a probabilistic Autumn program adds a new level of complexity, so our current synthesis algorithm is focused only on synthesizing *deterministic* Autumn programs.

We make one minimal, small-scoped exception to this, however: We allow use of `uniformChoice` at the *update function* level of an on-clause, but not at the *event* level. Explicitly, if no deterministic Autumn program is found by the synthesizer, which means cause synthesis fails on every concrete update function matrix M identified through the update function synthesis step, the algorithm will try to construct new concrete matrices using `uniformChoice`-based update functions. Currently, the algorithm only allows these random update functions to have the form `addObj (uniformChoice [ \* list of positions *\ ])`. For example, it is possible that the set of possible update functions in the unfiltered matrix for a certain object x at a certain time t is

$$M_{x,t} = \{\ \texttt{addObj (Bullet (Position 5 5))},\ \texttt{addObj (Bullet (uniformChoice (map (-> obj obj.origin) objects)))}\ \},$$

if (5, 5) is the location of an object in the list. Hence, a matching event might be found for a `uniformChoice`-based update function even if not found for deterministic update functions, so this limited form of nondeterminism in Autumn programs is supported by the synthesizer.

4.6 Code Generation

We implemented the Autumn language and AutumnSynth algorithm in Julia. The interpreter and library functions of the language are expressed in about 1,600 lines of Julia code, while the synthesizer is about 18,000 lines. To construct the correct Autumn syntax describing the object types $\mathcal{T}$, initial state X_0, and transition function F determined by AutumnSynth, we first express the stream definitions as follows. For each object type $t \in \mathcal{T}$, we define a list variable where the **`init`** value contains the values of all objects $x \in X_0$ with type t, and the **`next`** value is simply the

default **prev** expression (i.e. synthesized Autumn programs do not use the **next** clause, keeping all updates in on-clauses instead). After these stream definitions, the on-clauses are expressed in order from most-frequent to least-frequent for each object type $t \in \mathcal{T}$. Precisely, for each on-clause o_i in $F(H) = o_m(...(o_2(o_1(next(H), H), H)\dots), H)$, the on-clause o_i is the ith on-clause to appear in the Autumn code when read from top to bottom. These stream definitions and on-clauses are inserted following the standard grid size definition, background definition (we currently change all background colors to white, for simplicity), and definitions of the object types $\mathcal{T}$ at the start of the program to form the complete synthesized output.

5 EVALUATION

To evaluate AutumnSynth, we constructed a suite of 30 Autumn programs, called the Causal Inductive Synthesis Corpus (CISC), and also evaluated against a preexisting corpus of grid-world video games written in Python. We evaluated the following questions:

(1) How expressive is the Autumn language for modeling grid world environments?
(2) Does the AutumnSynth algorithm scale to interesting programs and long input traces?
(3) Are the synthesized programs able to generalize to new scenarios as opposed to just memorizing the input trace?

5.1 Benchmarks

5.1.1 The Causal Inductive Synthesis Corpus. Descriptions of each of the 30 benchmark programs in the Causal Inductive Synthesis Corpus are given in Fig. 10 in the Appendix. Of these models, 24 possess latent state and hence require the automata synthesis step of our algorithm, whereas the remaining six models do not possess latent structure and thus only test the functional synthesis component. The largest latent automaton present across the benchmark models has 11 states and 20 edges (Count V in the figure), while the largest number of latent automata in a single benchmark is four (Water Plug), where there is one global variable and three object types that each contain one object-specific latent field. Stills from some of the CISC benchmark programs are displayed in Fig. 1.

5.1.2 The Exploration, Modeling, and Planning Agent Corpus. In addition to the CISC benchmark, we also ran the AutumnSynth algorithm on a subset of the Exploration, Modeling, and Planning Agent (EMPA) benchmark suite [Tsividis et al. 2021]. The EMPA suite consists of 27 distinct grid-world games, each of which contains two to five levels, making for a total of 90 different levels across all games. In our evaluation, we used the first level of each of the 27 games, so our final evaluation set contained 27 EMPA programs. Unlike CISC, EMPA models are not natively written in Autumn, which makes them a suitable benchmark for measuring how effective Autumn is at capturing general grid-world dynamics. Some EMPA stills are shown in Fig. 11 in the Appendix.

5.2 Scalability and Performance

We now discuss our experiments on CISC and EMPA evaluating the scalability of AutumnSynth.

5.2.1 CISC. For each of the 30 CISC programs, we manually constructed an input sequence of user actions and corresponding observations. Our objective when curating these input traces was to demonstrate all of the dynamics encoded by the model so that the synthesizer would be forced to compute a solution capturing all aspects of the grid environment. We ran the three AutumnSynth variants on these manually curated input traces for each of the 24 latent-state-containing programs, and ran just the Heuristic on the 6 non-latent-state-based programs since the three algorithms differ only in the latent state synthesis step. For the purpose of this section, we declared a success

on a benchmark program if the synthesizer produced an Autumn program that generated the input sequence of observations given the input sequence of actions, even if the program was not semantically equivalent to the ground-truth benchmark. The results are shown in Fig. 7.

Notably, all three algorithm variants—Sketch, D&C Sketch, and Heuristic—are able to synthesize a program for most of the benchmark problems, with the Heuristic algorithm solving the most with 27 out of 30. The runtimes for the Heuristic algorithm range from just two minutes for some of the smaller benchmarks that have few on-clauses and short input traces (e.g. Disease with seven on-clauses and a 22-frame input trace) to up to 9 to 13 hours for larger benchmarks with many on-clauses or very long traces (e.g. Mario with 16 on-clauses or Sokoban with a 243-frame input trace). Unlike the Heuristic, both of the Sketch-based versions of the algorithm time out after 24 hours on several of the benchmarks. This is because those programs require large latent automata that in particular have many *hidden states*, which are additional accept states corresponding to a state-based update function. We provide further analysis of this result in Appendix A.1.1.

The fact that AutumnSynth is able to synthesize most of the CISC benchmarks, including those with many on-clauses and large latent automata, as well as from fairly long input traces with hundreds of frames, demonstrates the scalability of the algorithm.

5.2.2 EMPA. As in the CISC evaluation, we manually constructed an input observation trace for each of the 27 programs in the EMPA suite and ran the AutumnSynth algorithm on those traces. Unlike CISC programs, EMPA games have a higher frame rate (20 frames per second versus 3 frames per second) that results in longer input traces and are further played on grids with significantly larger dimensions (e.g. 300 pixels by 110 pixels for the Aliens game versus the 16 pixels by 16 pixels of most CISC games), both of which cause longer synthesis runtimes. In addition, stylistic differences between EMPA games and CISC games required us to slightly modify the event and update function spaces used in the AutumnSynth algorithm in order to synthesize Autumn programs modeling EMPA environments, as well as modify some of the lower-level heuristics used for the object perception, object mapping, and update function synthesis steps of the algorithm. We describe these modifications in greater detail in Appendix A.1.2.

One difference between the corpora is that the vast majority (19 out of 27) of the EMPA games are non-deterministic, compared to very few of the CISC programs. Further, the type of random behavior present in EMPA is different from that in CISC: While the only non-determinism in CISC appears at the update function level (i.e. update functions may use the `uniformChoice` function but all trigger events in on-clauses are deterministic), EMPA programs also have a few kinds of dynamics triggered by *random events*. For example, enemy objects in EMPA shoot out rockets with some probability at every time step, a feature not supported by the original AutumnSynth algorithm tuned to the CISC benchmark. To account for this difference, we slightly modified the cause synthesis step of AutumnSynth to assign a single, arbitrary random event to certain types of update functions (e.g. bullet addition) for which a deterministic event cannot be found, instead of synthesizing latent state to explain those update functions. The details of this modification to the algorithm in order to better suit the new domain of EMPA programs is described in the Appendix. We emphasize that while AutumnSynth is flexible enough to handle different domains, such lower-level, domain-specific tweaks are generally needed.

The results of our evaluation are shown in Fig. 8. We find that AutumnSynth synthesizes a solution for 21 out of the 27 EMPA programs. As in the CISC evaluation, we run all three variants of the AutumnSynth algorithm on the latent-state-based benchmarks—which compose 9 out of the total 27—while only running the Heuristic version on the non-latent-state-based models. Since the majority of EMPA benchmarks display nondeterministic behavior, we had to manually inspect those programs to check that they matched the given input trace instead of performing

an automatic check. Given the differences with respect to the original CISC programs that helped guide the development of the algorithm, it is notable that AutumnSynth manages to synthesize models of many of the EMPA games, including some with very large numbers of on-clauses and latent states, such as Aliens, which has 14 latent states and 37 on-clauses.

5.3 Quality of Synthesized Models

Next, we describe our experiments that help quantify the *quality* of the programs synthesized by AutumnSynth. Phrased differently, we are interested in verifying that the synthesized programs generalize reasonably well from the given input trace, as it is generally always possible for an inductive synthesis engine to simply produce a program that regurgitates the examples it was fed. To measure the generalization performance of synthesized programs, we perform two kinds of experiments: (1) We construct new input traces from the benchmark programs and compute how often the synthesized and benchmark programs produce the same output observation sequences on these new traces, and (2) we run AutumnSynth on *multiple input traces* that are strung together, to show that an exact semantic match to the ground truth can be synthesized given enough data.

5.3.1 CISC. For each deterministic CISC program, we constructed a test set of additional user action traces, and evaluated the synthesized program on each trace. (For the three non-deterministic models in CISC, the randomness in the models makes checking exact matches with the benchmark programs on new traces impossible. Hence, we simply inspected the synthesized programs and determined that they exactly matched their ground-truth programs, since they were small enough to do so.) We measured the fraction of each new observation sequence until the frame where it diverged from the ground-truth observation sequence (i.e. 1 if the sequences were exactly the same), and averaged these values to produce the percentages shown in the final column of Fig. 7. All but four—Magnets, Sokoban, Mario, and Coins—of the synthesized programs matched the corresponding ground-truth benchmark program on all of the test set traces. For Magnets and Sokoban, the reason behind their divergence was that some details of the model dynamics were not demonstrated in the input trace fed to the synthesizer. For example, a certain kind of diagonal motion that the blue magnet object can undergo was simply not shown in the input, so the synthesizer did not learn an on-clause describing it.

An incomplete input trace is also the reason that the Mario and Coins solutions generalized differently, but for those benchmarks, it is actually *impossible* to show all the dynamics of the model with just a single trace. Precisely, both the Mario and Coins models involved collecting coins that can be used to shoot bullets. Since there is a finite number of coins, it is impossible to show in one trace both that collecting all of the n coins in the model allows the agent to shoot n bullets in a row before clicking does nothing, and *also* show that collecting $k < n$ coins allows the agent to shoot just k bullets before clicking does nothing. Without this detail, the automata synthesis algorithm will not produce the ground-truth automaton, so we construct a few independent traces showing these dynamics. The AutumnSynth algorithm is trivially extended to take multiple traces as input by concatenating the traces into a single long trace and informing the synthesizer of the connecting times so it does not try to learn on-clauses describing those reset moments. The results of this experiment, performed for a subset of the five programs discussed, are shown in Fig. 9. It is clear that AutumnSynth scales to these extremely long inputs and produces the previously elusive ground-truth programs.

5.3.2 EMPA. Since the majority of the EMPA benchmarks contain non-deterministic behavior, we elected to skip the test set accuracy experiment, and only performed the multi-trace synthesis experiment with EMPA models. This experiment was particularly relevant for this benchmark, because EMPA games possess both win conditions and lose conditions, and it is impossible to

demonstrate both conditions in a single trace. This means the synthesizer cannot learn a full model of the game's dynamics with just one trace. As a result, the synthesized programs for several of the EMPA benchmarks discussed in the previous section exhibit generalization errors related to not being shown a certain way to win or die. For example, the model of how the moveable agent dies in a game often describes that intersecting just one type of enemy object causes the agent to die when in reality there are several enemy types that cause agent death.

Hence, for a small sample of EMPA programs, we constructed several traces to show that the full ground-truth model could be reached given enough data. Each additional trace demonstrated a new aspect of the benchmark model not seen in the previous trace, and we ran the synthesizer until all on-clauses in the synthesized output were correct. The results are shown in Fig. 9, where the accuracy is measured in terms of how many on-clauses were fully correct in the synthesized program. An interesting observation from this experiment is that we often made these new traces in the style of counterexample-guided synthesis (CEGIS): The synthesizer would produce an output that was slightly incorrect given n traces, so we would add an $(n+1)$-th trace to the input to correct this error, and then find that the newly synthesized program was still incorrect in some way, and repeat this process until the desired program was produced. This compatibility of AutumnSynth with CEGIS opens up interesting future directions related to online program synthesis.

6 RELATED WORK

There is an extensive literature on the synthesis of reactive programs from temporal specifications. Early work such as that of Pnueli and Rosner [Pnueli and Rosner 1990] involved doubly exponential algorithms with little possibility of practical use. However, in 2012, Boelm et al. [Bloem et al. 2012] showed that this style of synthesis could be made tractable by restricting it to a subset of temporal logic known as GR(1). However, a lot of that work has been limited to synthesizing finite state automata from temporal specifications and is not obvious how to apply them to inductive synthesis.

There is also an extensive literature on synthesizing finite state models from examples, starting with the seminal work of Dana Angluin [Angluin 1987]. This work relied on an ongoing interaction with an oracle to provide examples until a correct automaton was found. The initial algorithm has served as a basis for a number of more sophisticated methods and has been used extensively for automated model creation. There are now several mature tools for induction of finite state machines from examples [Combe et al. 2010]. For example, LearnLib [Raffelt and Steffen 2006] is a popular automata synthesis library that has been used in a number of applications. Vaandrager [Vaandrager 2017] provides a survey of recent methods and applications of automata learning.

In recent years, there has been an effort to improve upon the expressiveness of finite automata by learning *Register Automata* (RA). These automata extend the basic FSA formalism by allowing a fixed number of registers in addition to the finite set of states. The registers can be updated during transitions, but the set of operations on these registers is limited to assignments and equality comparisons. LearnLib incorporates Register Automata synthesis [Howar et al. 2012], and the use of Register Automata has expanded the range of applicability of these techniques.

Abstraction is another technique that has been applied to learn more expressive functions. For example, Cho et al. [Cho et al. 2010] demonstrate the use of automata inference together with abstraction to discover botnet command and control protocols. Aarts et al. [Aarts et al. 2015] have further generalized the combination of abstraction and automata synthesis to express complex protocol descriptions. Even with these extensions, however, none of these systems match the expressiveness of Autumn, which can generate programs with unbounded lists of objects, each with their own finite state, and with the expressiveness afforded by syntax guided synthesis.

	Model Name	# of A.	Max # of A. S.	Max # of A. T.	# of O.C.	Input Length (Frames)	Output Length (Program Lines)	Heuristic Runtime (min)	Sketch Runtime (min)	D&C Sketch Runtime (min)	Test Set Accuracy
No Latent State	Ants	0	0	0	3	17	22	207.6	N/A	N/A	N.D.
	Chase	0	0	0	7	27	32	12.2	N/A	N/A	N.D.
	Magnets	0	0	0	12	183	41	110.9	N/A	N/A	20.1%
	Space Invaders	0	0	0	27	55	84	750.0	N/A	N/A	N.D.
	Sokoban	0	0	0	12	243	43	818.6	N/A	N/A	76.5%
	Ice	0	0	0	10	27	45	3.1	N/A	N/A	100%
Latent State	Lights	1	2	2	4	24	36	2.5	2.7	5.2	100%
	Disease	1	2	2	7	22	31	2.4	3.4	3.6	100%
	Grow	1	2	2	11	95	49	185.9	235.7	412.3	100%
	Grow II	1	2	2	11	95	-	×	×	×	×
	Sandcastle I	1	2	2	7	32	33	3.0	3.2	4.7	100%
	Sandcastle II	1	2	2	7	32	-	×	×	×	×
	Bullets	2	4	12	4	53	93	10.8	26.4	⊥	100%
	Gravity I	1	4	12	9	19	38	2.3	2.5	3.1	100%
	Gravity II	2	4	12	14	24	50	2.9	3.6	6.1	100%
	Gravity III	1	9	24	32	27	83	2.1	5.8	⊥	100%
	Gravity IV	1	8	56	17	43	54	2.7	3.1	7.2	100%
	Count I	1	3	4	6	22	31	1.9	2.6	3.8	100%
	Count II	1	5	8	10	39	39	2.0	3.0	8.4	100%
	Count III	1	7	12	14	69	47	2.8	⊥	⊥	100%
	Count IV	1	9	16	18	109	55	3.9	⊥	⊥	100%
	Count V	1	11	20	22	149	63	4.4	⊥	⊥	100%
	Double Count I	1	5	8	12	94	43	2.3	3.1	25.1	100%
	Double Count II	1	9	16	20	156	59	3.3	⊥	⊥	100%
	Wind	1	3	4	9	23	43	21.0	26.9	35.7	100%
	Paint	1	5	5	10	27	39	2.5	2.7	12.9	100%
	Mario	2	3	6	16	81	59	526.9	598.6	×	79.9%
	Water Plug	4	3	6	16	64	53	89.9	⊥	⊥	100%
	Mario II	2	4	6	16	81	┤	×	×	×	×
	Coins	1	11	20	10	168	57	45.7	×	⊥	78.5%

Fig. 7. Results from running AutumnSynth on the CISC benchmark suite. ⊥ indicates timeout after 24 hours. The column header abbreviations signify the following: # of A. → # of Automata, Max # of A. S. → Max. # of Automaton States, Max # of A. T. → Max. # of Automaton Transitions, # of O.C. → # of On-Clauses. N.D. in the rightmost column stands for *non-deterministic*, indicating that those programs display the limited scope of non-determinism described in Section 4.5.6, and hence it is challenging to measure their closeness to the ground-truth benchmark via a test set. We further note that the output length was computed on the Heuristic-synthesized programs, and that the test set accuracies were computed by running the Heuristic-synthesized programs as well. This is because the Heuristic algorithm produced the most generalizable programs, though the Sketch algorithm matched it for the benchmarks that it also solved.

7 CONCLUSION

We have presented a new functional reactive domain-specific language (Autumn) suitable for expressing and synthesizing non-trivial grid world programs, and a new algorithm (AutumnSynth) that induces functional reactive programs in this language from observation data. We have empirically evaluated our algorithm on a new benchmark suite of 30 Autumn programs that we call the Causal Inductive Synthesis Corpus (CISC), as well as on a third-party dataset of 27 grid-world-style games written in Python. Our evaluation shows that AutumnSynth is able to synthesize long programs describing complex dynamics and latent state, and is able to do so from long input traces containing hundreds of frames. Looking ahead, we expect AutumnSynth will provide a template for how to integrate functional and reactive synthesis in other theory induction domains.

	ID	Model Name	# of A.	Max # of A. S.	Max # of A. T.	# of O.C.	Input Length (Frames, Sec)	Output Length (Program Lines)	Heuristic Runtime	Sketch Runtime	D&C Sketch Runtime
No Latent State	1	Antagonist	0	0	0	4	186, 9.3s	49	1.1h	N/A	N/A
	2	Avoid George	0	0	0	6	100, 5.0s	49	1.3h	N/A	N/A
	3	Bait	0	0	0	4	87, 4.4s	47	12.5m	N/A	N/A
	4	Bees and Birds	0	0	0	4	66, 3.3s	42	13.6m	N/A	N/A
	5	Boulder Dash	0	0	0	-	-	-	⊥	N/A	N/A
	6	Butterflies	0	0	0	3	53, 2.7s	38	40.2m	N/A	N/A
	7	Chase	0	0	0	-	-	-	×	N/A	N/A
	8	Closing Gates	0	0	0	4	159, 8.0s	47	1.8h	N/A	N/A
	9	Explore/Exploit	0	0	0	2	222, 11.1s	33	10.9m	N/A	N/A
	10	Helper	0	0	0	4	326, 16.3s	42	1.2h	N/A	N/A
	11	Jaws	0	0	0	8	192, 9.6s	59	1.8h	N/A	N/A
	12	Preconditions	0	0	0	3	83, 4.2s	42	4.6m	N/A	N/A
	13	Push Boulders	0	0	0	-	-	-	×	N/A	N/A
	14	Relational	0	0	0	5	177, 8.9s	45	30.9m	N/A	N/A
	15	Sokoban	0	0	0	3	189, 9.5s	38	17.9m	N/A	N/A
	16	Surprise	0	0	0	6	211, 10.6s	54	1.4h	N/A	N/A
	17	Water Game	0	0	0	5	57, 2.9s	52	15.4m	N/A	N/A
	18	Zelda	0	0	0	4	142, 7.1s	47	12.6m	N/A	N/A
Latent State	19	Aliens	3	14	20	37	318, 15.9s	114	21.3h	⊥	⊥
	20	Corridor	-	-	-	-	-	-	×	×	×
	21	Frogs	-	-	-	-	-	-	⊥	⊥	⊥
	22	Lemmings	3	4	12	15	356, 17.8s	77	6.3h	7.5h	⊥
	23	Missile Command	1	4	12		168, 8.4s	-	×	×	×
	24	My Aliens	1	2	2	23	127, 6.4s	57	1.6h	1.4h	⊥
	25	Plaque Attack	3	11	11	32	83, 4.2s	107	3.5h	⊥	⊥
	26	Portals	2	2	2	14	245, 12.3s	85	10.5h	10.1h	⊥
	27	Survive Zombies	2	12	11	30	138, 6.9s	90	2.9h	⊥	⊥

Fig. 8. Results from running AutumnSynth on the EMPA benchmark suite. ⊥ indicates timeout after 30 hours. The column header abbreviations have the same meanings as in Fig. 7.

	Model	Metrics	1 Trace	2 Traces	3 Traces	4 Traces	5 Traces	6 Traces	7 Traces
CISC	Magnets	Trace Len.	183	325					
		Runtime	32.8m	4.7h					
		Accuracy	20.1%	100%					
	Mario	Trace Len.	81	152	198				
		Runtime	36.6m	3.1h	13.8h				
		Accuracy	79.9%	87.1%	100%				
	Coins	Trace Len.	168	235	308	393	481	568	676
		Runtime	13.9m	35.1m	79.2m	3.0h	5.6h	9.6h	15.8h
		Accuracy	78.5%	82.2%	86.8%	90.9%	93.8%	97.5%	100%
EMPA	Sokoban	Trace Len.	118	288	405				
		Runtime	5.7m	14.3m	22.2m				
		Accuracy	2/3	2/3	3/3				
	Water Game	Trace Len.	152	172	257				
		Runtime	14.6m	18.0m	30.4m				
		Accuracy	4/6	5/6	6/6				

Fig. 9. Results from running AutumnSynth on multiple traces taken from a sample of CISC and EMPA benchmarks that are strung together.

REFERENCES

Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. 2015. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods Syst. Des.* 46, 1 (2015), 1–41. https://doi.org/10.1007/s10703-014-0216-x

Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

Suguman Bansal, Kedar S. Namjoshi, and Yaniv Sa'ar. 2018. Synthesis of Asynchronous Reactive Programs from Temporal Specifications. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 367–385. https://doi.org/10.1007/978-3-319-96145-3_20

Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A constraint-based approach to solving games on infinite graphs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 221–234. https://doi.org/10.1145/2535838.2535860

Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. 2012. Synthesis of Reactive(1) designs. *J. Comput. Syst. Sci.* 78, 3 (2012), 911–938. https://doi.org/10.1016/j.jcss.2011.08.007

Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. 2010. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 426–439. https://doi.org/10.1145/1866307.1866355

Wonhyuk Choi, Bernd Finkbeiner, Ruzica Piskac, and Mark Santolucito. 2022. Can reactive synthesis and syntax-guided synthesis be friends?. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 229–243. https://doi.org/10.1145/3519939.3523429

David Combe, Colin de la Higuera, and Jean-Christophe Janodet. 2010. Zulu: An Interactive Learning Competition. In *Finite-State Methods and Natural Language Processing*, Anssi Yli-Jyrä, András Kornai, Jacques Sakarovitch, and Bruce Watson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–146.

Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 411–422. https://doi.org/10.1145/2491956.2462161

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: automatic conversion of 3D models to CSG trees. *ACM Trans. Graph.* 37, 6 (2018), 213. https://doi.org/10.1145/3272127.3275006

Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2015. Unsupervised Learning by Program Synthesis. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 973–981. https://proceedings.neurips.cc/paper/2015/hash/b73dfe25b4b8714c029b37a6ad3006fa-Abstract.html

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*. 835–850.

Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2019. Synthesizing functional reactive programs. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, Richard A. Eisenberg (Ed.). ACM, 162–175. https://doi.org/10.1145/3331545.3342601

Alison Gopnik and Henry M Wellman. 2012. Reconstructing constructivism: causal models, Bayesian learning mechanisms, and the theory theory. *Psychological bulletin* 138, 6 (2012), 1085.

Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. 2012. Inferring Canonical Register Automata. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 251–266.

Ali Sinan Köksal, Yewen Pu, Saurabh Srivastava, Rastislav Bodík, Jasmin Fisher, and Nir Piterman. 2013. Synthesis of biological models from mutation experiments. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 469–482. https://doi.org/10.1145/2429069.2429125

Amir Pnueli and Roni Rosner. 1990. Distributed Reactive Systems Are Hard to Synthesize. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*. IEEE Computer Society, 746–757. https://doi.org/10.1109/FSCS.1990.89597

Harald Raffelt and Bernhard Steffen. 2006. LearnLib: A Library for Automata Learning and Experimentation. In *Fundamental Approaches to Software Engineering*, Luciano Baresi and Reiko Heckel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 377–380.

Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Megan Tjandrasuwita, Jennifer J Sun, Ann Kennedy, Swarat Chaudhuri, and Yisong Yue. 2021. Interpreting expert annotation differences in animal behavior. *arXiv preprint arXiv:2106.06114* (2021).

Pedro A. Tsividis, Joao Loula, Jake Burga, Nathan Foss, Andres Campero, Thomas Pouncy, Samuel J. Gershman, and Joshua B. Tenenbaum. 2021. Human-Level Reinforcement Learning through Theory-Based Modeling, Exploration, and Planning. https://doi.org/10.48550/ARXIV.2107.12544

Tomer D Ullman and Joshua B Tenenbaum. 2020. Bayesian models of conceptual development: Learning as building models of the world. *Annual Review of Developmental Psychology* (2020).

Frits W. Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95. https://doi.org/10.1145/2967606

Willem Zuidema, Robert M French, Raquel G Alhama, Kevin Ellis, Timothy J O'Donnell, Tim Sainburg, and Timothy Q Gentner. 2020. Five ways in which computational modeling can help advance cognitive science: Lessons from artificial grammar learning. *Topics in cognitive science* 12, 3 (2020), 925–941.

A APPENDIX

A.1 Additional Evaluation Details

A.1.1 Further CISC Analysis. We briefly discuss the reason behind why the Sketch-based AutumnSynth variants timed out on a number of the CISC benchmarks. Those benchmark programs require large latent automata that, in particular, have many *hidden states*, which are additional accept states corresponding to a state-based update function. More precisely, every update function triggered by a latent-state-based event must have at least one accept state in the automaton. For this reason, the number of states in the automaton must be at least the number of update functions, so the Sketch solver begins its search for a satisfying automaton by searching among automata with this minimum state count. If a solution is not found, Sketch will keep incrementing this state count until it finds a correct automaton. Hence, when the actual number of states in the desired automaton is much larger than the number of update functions (e.g. of the 11 states in the Count V automaton, 8 states are hidden), the underlying SAT solver does not terminate quickly, resulting in a timeout. This also explains why the Sketch-based algorithms time out on a few of the EMPA benchmarks. The Heuristic algorithm avoids these sinks by means of some domain-specific tricks, though we emphasize that the algorithm is not complete and will not work on every input.

A.1.2 Modifications to AutumnSynth for the EMPA Domain. In Section 5.2, we described that lower-level modifications to AutumnSynth were made before applying the algorithm to EMPA. In this section, we provide further detail about these changes to give a sense of the nature of the modifications that may be needed to apply AutumnSynth to new domains.

With respect to update function synthesis, we introduced a few new heuristics to the update function filtering algorithm in order to account for some of the nuances of EMPA games. For example, unlike CISC programs, many EMPA programs are characterized by objects moving at regular time intervals of 5, 12, 16, etc. time steps. Knowing this fact about the domain, we apply some heuristics that check for time-regularity-based motion in the update function matrix, so that the concrete matrix produced does not describe objects to be moving at off-times (e.g. by assigning `prev obj` instead of a motion-causing update function at a time when an object should not be moving, if there is ambiguity). The standard frequency-based filtering technique is applied after the modifications made by this regularity-based heuristic.

In addition, regarding cause synthesis, we added a few new library functions that capture some common behaviors in the EMPA suite, which were originally more cumbersome to express in Autumn. For example, we added a single function that captures "pushing" behavior, which originally was expressed via a combination of a few different events in the event search space. This was useful because many objects in EMPA games can push other objects, and variations of this pushing behavior, which may have required a trigger event combining too many atomic events to be searched for in a reasonable amount of time, can now be identified fairly quickly.

Lastly, we also provide a concrete example of how we modified the event search process to handle the random events in EMPA games. Specifically, if a trigger event cannot be found for an update function and the update function has a particular form, such as being an object addition, that is typically caused by a random event, we assign the update function the trigger event,

```
(uniformChoice (list 1 2 3 4 5 6 7 8 9 10)) == 1,
```

regardless of the actual underlying probability distribution. In other words, we only claim to binarily identify that a particular update function is non-deterministic versus deterministic, rather than infer an approximately correct probability distribution describing the behavior.

A.1.3 Event Search. We currently implement search through the space of conjunctions and disjunctions of atomic events using *Z*3. We enumerate through events in order of increasing numbers of atomic events contained within them, beginning with conjunctions and disjunctions of two atoms, three atoms, etc. up to four atoms. We also include one five-atom form that appears fairly frequently, the disjunction of five atoms, though we do not consider other combinations for the sake of performance.

A.1.4 Note about Object Perception Algorithms. We mention that in our current implementation, we use a list of colored pixel positions as the representation of the observed grid frames given as input to the AutumnSynth algorithm, instead of raw images. The significance of this slightly more processed representation is that, if two objects overlap at one pixel, the synthesizer does not need to figure out from that pixel's color and transparency value that there are really two overlapping colors there. Instead, the input will already include two elements with the same x-y coordinates and color, e.g. $\{(x, y, color), (x, y, color)\}$. This detangling of pixels with overlap into their individual components can be trivially performed by storing a mapping between (1) all RGBA values formed via overlaps of a finite number of colors, and (2) the lists of colors that compose them. We decided to bypass implementing this more complete object parsing algorithm because perception is not a contribution of our work.

In addition, in our slightly modified implementation for synthesizing EMPA, we currently further skip the object perception step by taking as input an *object-ordered* list of colored pixels saved from a web interface for EMPA games, unlike the CISC implementation. More precisely, every object in the EMPA suite is 10 pixels by 10 pixels, and the input list of pixels for any frame can be split into groups of pixels corresponding to each object simply by dividing the list into sub-lists of size 900. In the case of one benchmark program, Closing Gates, we refine this object representation further by applying the multi-cell object parsing algorithm defined in Section 4.1 on top of the default 10 by 10 object units. This produces new object types composed of several 10 by 10 cells (e.g. forming a type with dimensions 40 by 10). We bypassed the full object perception step in this way to focus on the more interesting later steps of the algorithm in our evaluation, but it is certainly possible to develop complete object perception algorithms that operate on EMPA frames.

A.2 Automata Synthesis via Abstraction Refinement

In this section, we provide some further intuition about our heuristic algorithm for automata synthesis, since it is one of the more unusual contributions of our work. In particular, we describe the algorithm as being closely related to *counterexample-guided abstraction refinement*. This is because a more *abstract*, small-state automaton specification is iteratively *refined* in response to errors until a concrete automaton satisfying the specification is found. We call the specification a *partial state sequence*. Informally, it may be viewed as the current guess for the *unrolled state trajectory* of the desired automaton, or the sequence of state values it underwent over time.

Definition A.1 (Partial State Sequence). Let $\mathcal{S}$ be the space of possible automata describing the occurrence of a set $\mathcal{U}$ of unmatched update functions with the same co-occurring event c. A partial state sequence $P_{\mathcal{S}}$ describing $\mathcal{S}$ is a time-ordered list of tuples, where each tuple contains a time and an integer state hypothesized to be the state of the desired latent automaton at that time.

At initialization of the automata synthesis procedure, the partial state sequence contains only elements corresponding to the times where the co-occurring event c is true. Each of these times is labeled with 1 through n if one of the n update functions $u \in \mathcal{U}$ took place at that time, or $n + 1$ if none of the update functions took place at that time. For example, the partial state sequence corresponding to a possible 10-frame observation trace from the Gravity model is

$$P_S = \left[\begin{Bmatrix} t{=}1 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}2 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}3 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}4 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}5 \\ s{=}3 \end{Bmatrix}, \begin{Bmatrix} t{=}6 \\ s{=}4 \end{Bmatrix}, \begin{Bmatrix} t{=}7 \\ s{=}4 \end{Bmatrix}, \begin{Bmatrix} t{=}8 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}9 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}10 \\ s{=}2 \end{Bmatrix}\right],$$

where the state labels $s = 1$ through $s = 4$ correspond to the four update functions `moveLeft`, `moveRight`, `moveDown`, and `moveUp`, and no times have the label $n + 1 = 5$ because one of the four update functions took place at every co-occurring event (`true`) time. Similarly, the initial partial state sequence for the bullet addition update function in the Mario program is

$$P_S = \left[\begin{Bmatrix} t{=}1 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}2 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}5 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}6 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}7 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}9 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}10 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}11 \\ s{=}2 \end{Bmatrix}\right],$$

where the state labels $s = 1$ corresponds to bullet addition times and the state labels $s = 2$ corresponds to times when the co-occurring event c (`clicked`) was true but no bullet was added.

From the partial state sequence, a set of *transition ranges* is extracted, which specify the search for *transition events* that label the edges in the final automaton.

Definition A.2 (Transition Range). A transition range is a set of time ranges corresponding to a change in state value between consecutive elements of a partial state sequence P_S. It is specified by a (1) start state, (2) end state, and (3) set of (start time, end time) pairs corresponding to the time periods in which that particular state change occurred.

For example, the transition ranges corresponding to the Gravity partial state sequence above are

(1) $(s = 1, s = 2)$ taking place during the time ranges $(t = 3, t = 4)$ and $(t = 9, t = 10)$,
(2) $(s = 2, s = 3)$ taking place during time range $(t = 4, t = 5)$,
(3) $(s = 3, s = 4)$ taking place during time range $(t = 5, t = 6)$, and
(4) $(s = 4, s = 1)$ taking place during time range $(t = 7, t = 8)$.

Similarly, the transition ranges corresponding to the Mario partial state sequence are

(1) $(s = 2, s = 1)$ taking place during time ranges $(t = 2, t = 5)$ and $(t = 7, t = 9)$, and
(2) $(s = 1, s = 2)$ taking place during time ranges $(t = 6, t = 7)$ and $(t = 9, t = 10)$.

For each transition range, the automata synthesis procedure searches for an event in a pre-specified space of transition events that occurs only during the specified time ranges. If such a matching event is found for each transition range, we say that the partial state sequence is *satisfied*, and the procedure terminates. For example, the events `clicked rightButton`, `clicked upButton`, `clicked downButton`, and `clicked leftButton` match the four transition ranges for Gravity respectively, and hence form the labels of the latent automaton's edges.

However, if no matching event is found for a particular transition range, the following process is performed: First, the *closest* transition event, or the event that takes place during each time range and occurs outside the time ranges as few times as possible, is selected. Second, for each undesired time that this closest transition event takes place, the state value at that time in the partial state sequence—or the nearest time in the sequence before that time, if it does not appear in the sequence—is changed to a new, unused value. This causes the transition event

```
closest_transition_event && (latentVar == start_state),
```

where `start_state` is the start value of the transition range, no longer takes place at the extra times. To provide a concrete example, this state specialization process takes place in the Mario program when solving the state transition $(s = 1, s = 2)$, which occurs during time ranges $(t = 6, t = 7)$ and $(t = 9, t = 10)$. No transition event takes place within only those ranges, since the closest event `clicked` also takes place at the additional time $t = 5$. Changing the state value at time $t = 5$, however, so that the new partial state sequence is

$$P'_S = \left[\begin{Bmatrix} t{=}1 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}2 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}5 \\ s{=}3 \end{Bmatrix}, \begin{Bmatrix} t{=}6 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}7 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}9 \\ s{=}1 \end{Bmatrix}, \begin{Bmatrix} t{=}10 \\ s{=}2 \end{Bmatrix}, \begin{Bmatrix} t{=}11 \\ s{=}2 \end{Bmatrix}\right],$$

makes it so that the event `clicked && (latentVar == 1)` satisfies the new specification for the state change $(s = 1, s = 2)$, which now only occurs during $(t = 6, t = 7)$. In other words, by *splitting* the original state $s = 1$ into two states $s = 1$ and $s = 3$, the transition range $(s = 2, s = 1)$ is solved. Having refined the partial state sequence in this way, the algorithm proceeds by repeating transition search and state specialization for the rest of the transition ranges, until a transition event is found for each transition range without need for additional specialization. We employ a few additional variations of the error-driven state splitting technique described here to get this procedure to converge to the desired solution. In the Mario program, the final partial state sequence $P^f_{\mathcal{S}}$ ends up being

$$P^f_{\mathcal{S}} = \left[\left\{\begin{matrix}t{=}1\\ s{=}2\end{matrix}\right\}, \left\{\begin{matrix}t{=}2\\ s{=}2\end{matrix}\right\}, \left\{\begin{matrix}t{=}3\\ s{=}1\end{matrix}\right\}, \left\{\begin{matrix}t{=}5\\ s{=}3\end{matrix}\right\}, \left\{\begin{matrix}t{=}6\\ s{=}1\end{matrix}\right\}, \left\{\begin{matrix}t{=}7\\ s{=}2\end{matrix}\right\}, \left\{\begin{matrix}t{=}9\\ s{=}1\end{matrix}\right\}, \left\{\begin{matrix}t{=}10\\ s{=}2\end{matrix}\right\}, \left\{\begin{matrix}t{=}11\\ s{=}2\end{matrix}\right\}\right],$$

where the additional $s = 1$ state is created because no transition event could be found to match the state change $(s = 2, s = 3)$. The transition events associated with this final state sequence are then

(1) $(s = 2, s = 1) \longrightarrow$ `intersects (prev mario) (prev coins)`,
(2) $(s = 1, s = 3) \longrightarrow$ `intersects (prev mario) (prev coins)`,
(3) $(s = 3, s = 1) \longrightarrow$ `clicked`, and
(4) $(s = 1, s = 2) \longrightarrow$ `clicked`.

We note that the state labels are slightly different from those depicted in the diagram of this automaton in Fig. 3. In the actual synthesized automaton with states and labels described above, the state $s = 2$ corresponds to Mario having collected 0 coins, the state $s = 1$ corresponds to Mario having collected 1 coin, and the state $s = 3$ corresponds to Mario having collected 2 coins. For the diagram, we permuted and decremented these labels by 1 so that they exactly lined up with the number of collected coins, just for ease of understanding; the two are semantically equivalent.

B ADDITIONAL FIGURES

	Model Name	Description
	Ants	Ants foraging for randomly generated food particles.
	Chase	Agent evading randomly generated enemies.
	Magnets	Two magnets displaying attraction/repulsion.
	Space Invaders	A clone of Atari Space Invaders.
	Sokoban	A clone of Sokoban.
	Ice	Water particles behaving like solids vs. liquids.
Latent State	Lights	Clicking turns on/off a set of lights.
Latent State	Disease	Sick particles infect healthy particles.
Latent State	Grow I	Flowers grow upon water addition and sunlight.
Latent State	Grow II	Same as above, but plant stems grow longer.
Latent State	Sandcastle I	Water causes sand particles to turn liquid from solid.
Latent State	Sandcastle II	Same as above, but buttons match water/sand colors.
Latent State	Bullets	Agent that can shoot bullets in four directions.
Latent State	Gravity I	Blocks move according to four gravity directions.
Latent State	Gravity II	Same as above, except colors of added blocks rotate.
Latent State	Gravity III	Blocks move according to nine gravity directions.
Latent State	Gravity IV	Same as Gravity I, except there are eight gravities.
Latent State	Count I	Weighted left/right movement, with two weights.
Latent State	Count II	Weighted left/right movement, with four weights.
Latent State	Count III	Weighted left/right movement, with six weights.
Latent State	Count IV	Weighted left/right movement, with eight weights.
Latent State	Count V	Weighted left/right movement, with ten weights.
Latent State	Double Count I	Weighted left/right/up/down, with four weights.
Latent State	Double Count II	Weighted left/right/up/down, with eight weights.
Latent State	Wind	Snow falls left, down, or right based on wind state.
Latent State	Paint	A simplified clone of MSFT Paint, with five colors.
Latent State	Mario	A Mario-style agent collects coins and shoots enemy.
Latent State	Mario II	Same as above, but enemy has two lives, not just one.
Latent State	Coins	Agent can collect 10 coins which convert to bullets.
Latent State	Water Plug	Water interacts with a sink and removable sink plug.

Fig. 10. Descriptions of the 30 benchmark programs in the Causal Inductive Synthesis Corpus.

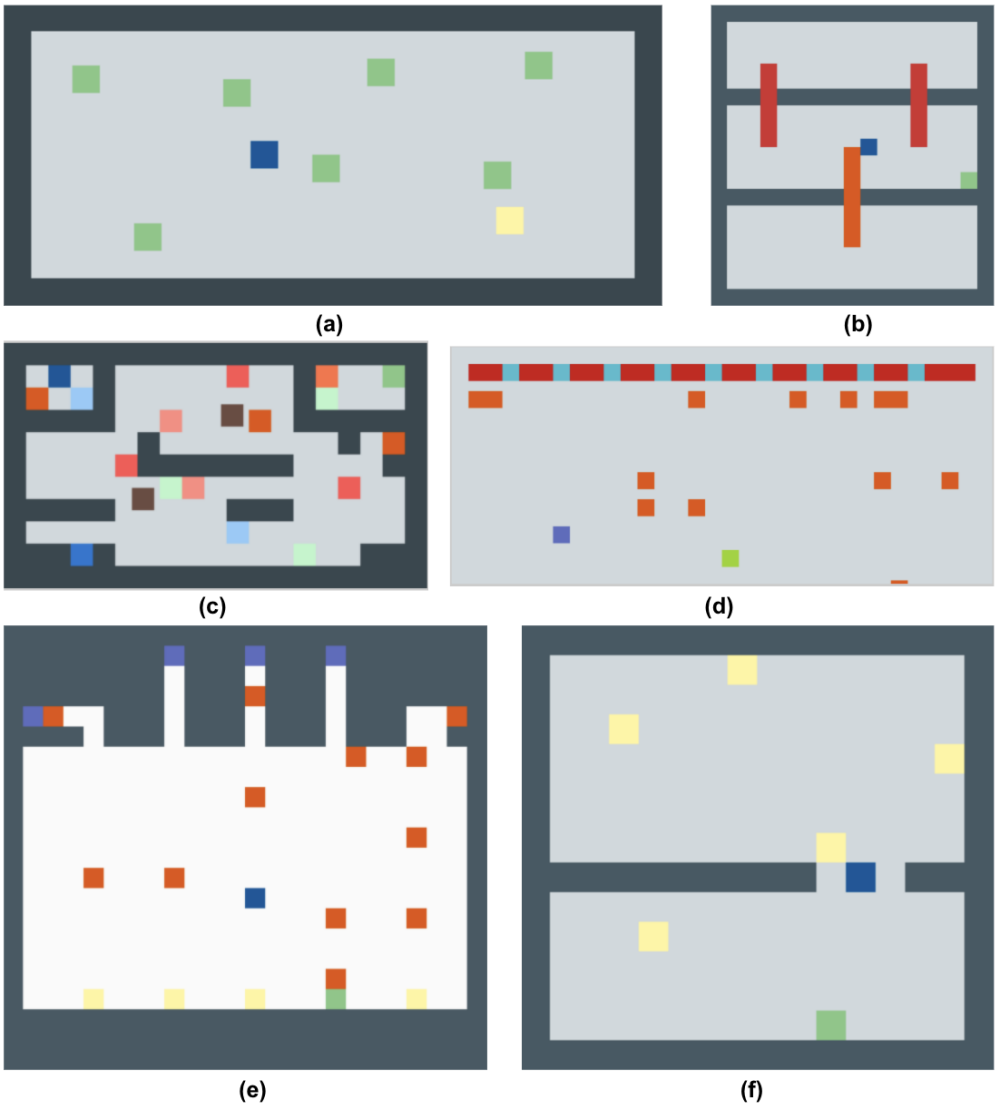


Fig. 11. Stills from a sample of programs in the EMPA suite, resized to fit neatly into the figure. (a) Avoid George, where the dark blue agent must avoid the yellow enemy, which chases it and the randomly moving green objects. (b) Missile Command, in which the dark blue agent must get to the green goal before the gates close. (c) Portals, in which some blocks teleport the agent to other blocks. (d) My Aliens, in which the agent collects orange and is killed by purple objects. (e) Plaque Attack, in which the agent can shoot at orange enemies before they reach the yellow goals. (f) Bees and Birds, where the randomly moving yellow objects can kill the enemy before it reaches the green goal.

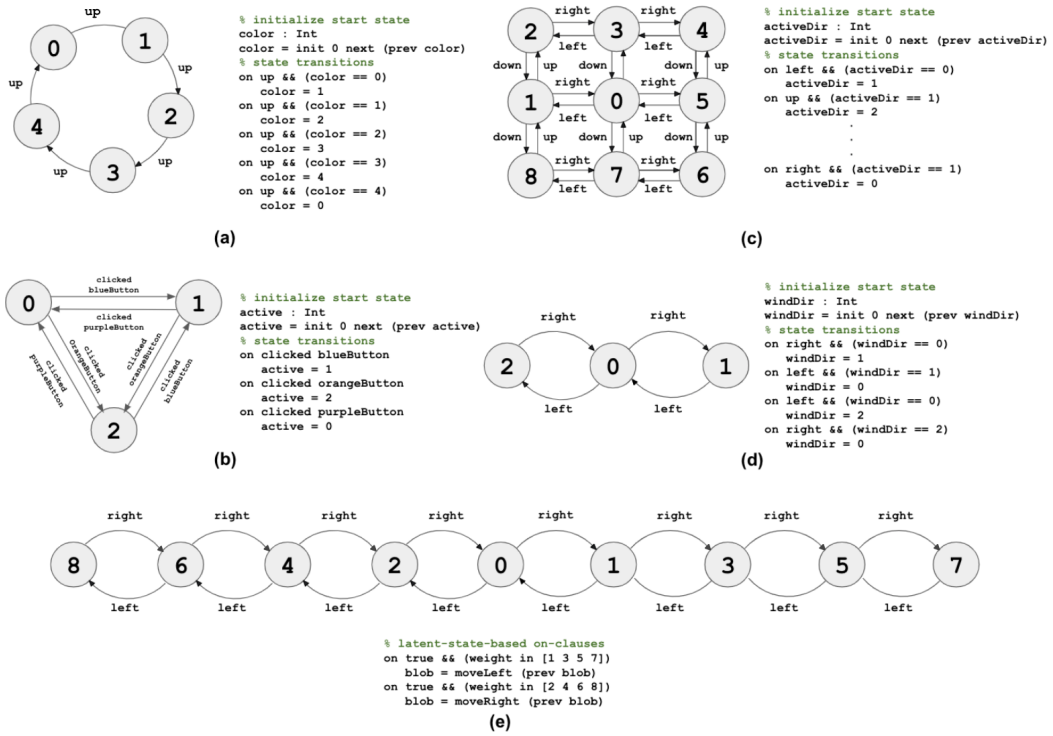


Fig. 12. Sample latent state automata synthesized by AutumnSynth on the CISC domain. **(a)** Paint model. Each state corresponds to a different color, indicating the color of the block added when a user clicks on an empty grid square. Pressing up cycles through the colors. **(b)** Water Plug model. Clicking one of three colored buttons changes the color of the block added when a user clicks an empty grid cell to the color of the button. **(c)** Gravity III model. Each state corresponds to one of the nine directions of motion formed by crossing three possible x-directions (-1, 0, 1) with y-directions (-1, 0, 1). **(d)** Wind model. Snow particles fall downward, left-diagonally, and right-diagonally, depending on the wind state that changes with left/right arrow keys. **(e)** Count IV model. Instead of giving the Autumn language description for this automaton, we show the on-clauses for the update functions that depend on the latent variable instead. Here, a particle moves left if the total number of left presses is greater than the total number of right presses up to a maximum difference of 4. It moves right according to a similar rule, and is stationary in state zero.