Synthesis of Reactive Programs with Structured Latent State

Ria A. Das
MIT CSAIL
riadas@mit.edu

Joshua B. Tenenbaum
MIT BCS
jbt@mit.edu

Armando Solar-Lezama
MIT CSAIL
asolar@csail.mit.edu

Zenna Tavares
Columbia University
zt2297@columbia.edu

Abstract

The human ability to efficiently discover causal theories of their environments from observations is a feat of nature that remains elusive in machines. In this work, we attempt to make progress on this frontier by formulating the challenge of causal mechanism discovery from observed data as one of *program synthesis*. We focus on the domain of time-varying, Atari-like 2D grid worlds, and represent causal models in this domain using a programming language called AUTUMN. Discovering the causal structure underlying a sequence of observations is equivalent to identifying the *program* in the AUTUMN language that generates the observations. We introduce a novel program synthesis algorithm, called AUTUMNSYNTH, that approaches this synthesis challenge by integrating standard methods of synthesizing functions with an *automata synthesis* approach, used to discover the model's latent state. We evaluate our method on a suite of AUTUMN programs designed to express the richness of the domain, and our results signal the potential of our formulation.

1 Introduction

Children are born scientists [4, 5, 9]. Without being told the rules, they can figure out how a new toy or video game works—a full causal theory of which stimuli cause which changes—after just minutes of observation. Such a data-efficient and flexible ability to discover causal models has yet to be demonstrated in an artificial agent. In this paper, we seek to address this gap by framing the problem of discovering a causal model from observed data as one of *program synthesis*. We represent a causal model as a program in a domain-specific language (DSL), and as such are able to exploit two key advantages of programs: their ability to express complex designs very concisely, and that they can often be synthesized from small input data. In this work, we focus specifically on the domain of time-varying mechanisms in 2D Atari-like grid worlds, and design a functional reactive language called AUTUMN to succinctly represent a wide variety of interesting causal phenomena within these worlds (Figure 1). The input to an AUTUMN program is a sequence of user events (either clicking anywhere on the grid, pressing an arrow key, or no event), one per time step. The output is a corresponding sequence of grid frames, each a partially observed representation of the underlying AUTUMN program state. Our goal is to build a program synthesis engine which, given the output sequence of observed frames and the associated user events, produces the best program in the AUTUMN language that generates the observations.

Critically, despite the natural fit of programs as a model representation for causal discovery, the AUTUMN domain does introduce a complication not handled by most existing program synthesis algorithms. To understand this aspect, we first note that, though the overall AUTUMN synthesis

35th Conference on Neural Information Processing Systems (aiplans 2021), Sydney, Australia.

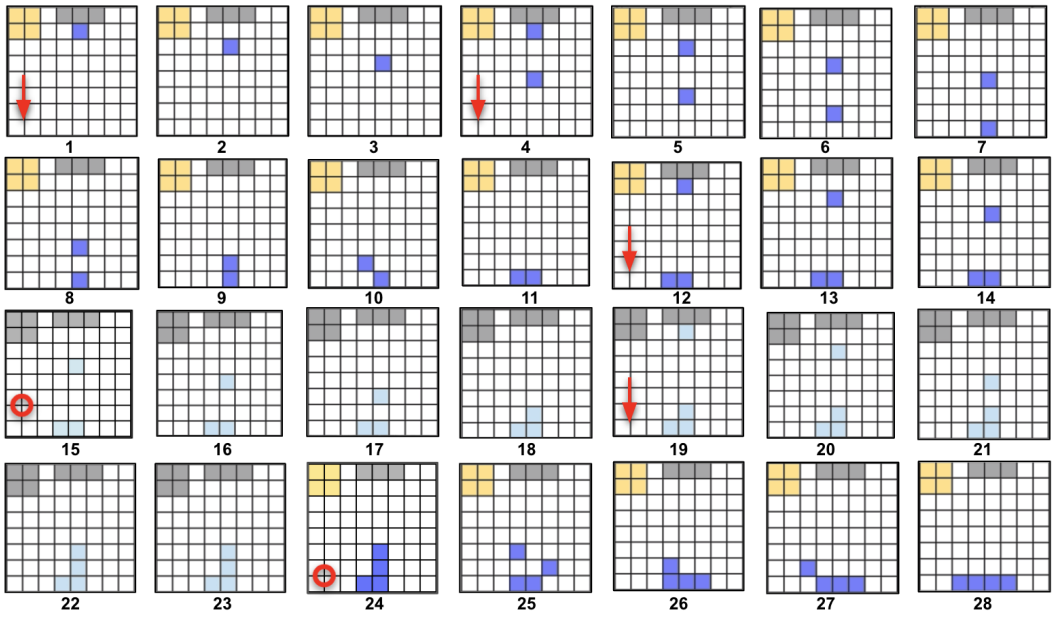

Figure 1: Sequence of grid frames from the Ice program. At times 1 and 4, the user presses down (red arrow), releasing a blue water particle from the gray cloud. The water moves down to the lowest possible height, moving to the side (time 10) if necessary to reach this height. The user presses down again at time 12, and then clicks anywhere (red circle) at time 15. The click causes the sun to change color and the water to turn to ice, which *stacks* rather than tries to reach the lowest height. A down press at time 19 releases another ice particle from the cloud. Finally, a click at time 24 changes the sun color back to yellow and turns the ice back to water, which again seeks the lowest possible height.

problem is to construct a program that takes a sequence of inputs to a sequence of outputs, the setting is also *reactive*: The output sequence is constructed *one time step at a time* from the user event at that time and the *program state* that summarizes what has happened in the program over all the previous time steps. Thus, the program (function) that we wish to synthesize is one that takes as input the current program state and user event, and produces as output a *modified* program state corresponding to the next time. However, while most methods for functional synthesis from input-output data assume that both the inputs and outputs are fully observed [2], we never directly observe the full AUTUMN program state. Instead, we only have access to the grid frames, which are partial views of the underlying program states produced by a *rendering function*. Further, while there exist synthesis approaches that reconstruct hidden elements from partially observed inputs, our problem requires that we learn ***how*** these latent elements are modified over time, instead of just *what* they are at one time. Specifically, a full AUTUMN program state consists of a set of *objects*, each storing a 2D position and shape along with potentially some internal (latent) data fields, as well as global latent elements that are not tied to any particular object. The values of these time-varying latent variables may pivotally impact the rendered grid objects, but they are never directly exposed themselves. As such, part of the AUTUMN synthesis challenge becomes reconstructing the appropriate values and dynamics of the *latent state* in the generating program.

For concreteness, we give an example of this additional fold in our problem formulation via an AUTUMN program called "Mario" (Figure 2). In Mario, the agent (red) moves around with arrow key presses and can collect coins (yellow). If the agent has collected a positive number of coins, on a click event, a bullet (purple) is released upwards from the agent's position, and the agent's coin count is decremented. The number of coins that the agent possesses is not displayed anywhere on the grid at any time, so the only way to write an AUTUMN program that captures this behavior is to define an *internal* or *latent variable*, which tracks the number of coins (bullets) possessed by the agent. This involves both setting the variable's initial value, as well as learning functions that dictate when (on what stimulus) and how (increment, decrement, etc.) that value will change. Notably, though a simple exercise for a human, this identification of latent state represents an elevation of the standard program

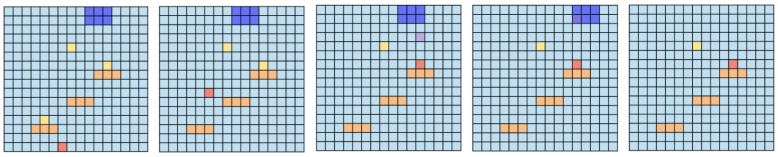

Figure 2: Stills (ordered but not consecutive) from the Mario model. The red agent initially cannot shoot bullets (purple), but after jumping up and collecting coins (yellow), shoots on a user click. The agent can no longer shoot when all its bullets are used up. A bullet kills the blue enemy.

synthesis paradigm to a new regime. While much work exists on learning latent state representations, including structured representations like those in AUTUMN [6, 8, 10], the integration of dynamical latent state discovery methods with standard methods of synthesizing functions is novel, as far as we know.

To solve the AUTUMN latent state learning problem, we first clarify that we can reasonably parse the visible elements of the program state—the object shapes and positions—directly from the grid frames, but cannot do the same for *invisible* elements, which include internal object-specific and global fields like `numCoins`. We address this challenge by making a key insight: The invisible state in AUTUMN programs can be viewed as finite state automata (FSMs), where the labels on the automata transitions are simply AUTUMN predicates (over the visible *and* invisible state). As such, we design our synthesis algorithm for AUTUMN programs, called AUTUMNSYNTH, to be a fusion of standard methods of functional synthesis with techniques drawn from the largely orthogonal area of *automata synthesis*. At a high level, AUTUMNSYNTH first attempts to synthesize small component functions in the full generating program without constructing new invisible state. Upon failure of this attempt, the method uses a novel automata synthesis algorithm to enrich the original program state with new, compact latent elements, which then enable the previous functional synthesis step to succeed.

The most relevant prior work to our problem setting are approaches that synthesize reactive models as finite state machines. These approaches, however, cannot handle model state as complex as our framing, in which objects may be created and destroyed and may each have their own hidden state in addition to coordinates [11]. Further, these approaches often begin with specifications in the form of temporal logic formulas instead of an observed sequence [7]. The other line of related work is that of functional synthesis approaches that can synthesize transformations on very complex data structures, but have no concerns about discovering latent state, e.g. work by Ellis et. al. on inferring graphics programs from hand-drawn images [3].

We give more details about the AUTUMNSYNTH algorithm as well as the AUTUMN language in Section 2. Then, in Section 3, we describe the results of evaluating our synthesis algorithm on a benchmark dataset constructed to highlight the diversity of the AUTUMN domain. We note that, in the rest of the paper, we will use the term *latent state* to specifically refer to *invisible state*, though, technically, the full program state is latent since it is all viewed through a rendering function.

2 The AUTUMNSYNTH Algorithm

The AUTUMN language was designed to concisely express a rich variety of causal mechanisms in interactive 2D grid worlds. The language is *functional reactive*, indicating that it augments the standard functional language definition with primitive support for temporal events. The key elements of an AUTUMN program are (1) object type definitions, (2) object instance and latent variable definitions, and (3) *on-clauses*. The most interesting component to synthesize are these on-clauses, which describe the causal dynamics of the model via a set of statements with the syntax `on event update`, where `event` is a predicate and `update` is a modification to an object that overrides the object's default behavior. See Appendix B for details and sample AUTUMN programs.

Synthesizing the correct AUTUMN program from observed data involves determining the object types, object instance and latent variable definitions, and on-clauses described previously. The AUTUMNSYNTH algorithm, as an end-to-end synthesis algorithm taking images as input, consists of four distinct steps, each producing a new representation of the input sequence. These steps are (1) ***perception***, in which objects are parsed from the observed grid frames; (2) ***object tracking***, which

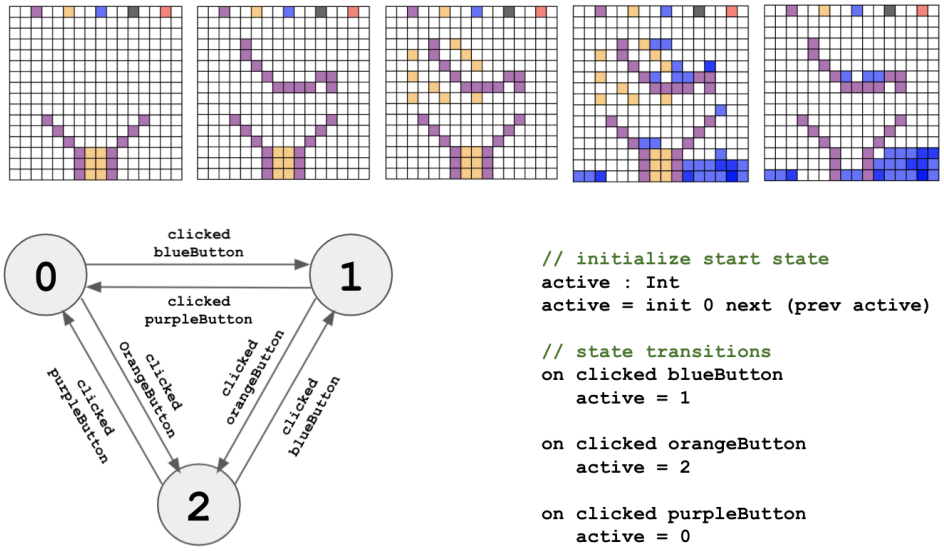

Figure 3: The latent state automaton learned for the Water Plug program. The last clicked of the top-row purple, orange, and blue buttons dictates which colored cell is added upon clicking free positions. *Top*: A sequence of frames (with time jumps) from the model. *Left*: Diagram of the learned state machine. *Right*: The AUTUMN description of the structure (we renamed the raw synthesized output variable names with more meaningful names for simplicity of exposition).

involves assigning each object in a frame to either (a) an object in the subsequent frame, deemed to be its transformed image in the next time, or (b) no object, indicating that the object was removed in the next time; (3) ***update function synthesis***, in which AUTUMN expressions, called update functions, describing each object-object mapping from Step 2 are determined; and (4) ***event synthesis***, in which AUTUMN predicates that trigger each update function from Step 3 are sought. Event synthesis also involves synthesizing invisible state in the form of automata, in the case that an appropriate predicate that triggers a given program update cannot be constructed from the existing program state structure. Specifically, when we cannot find an AUTUMN predicate that causes a particular object update, we augment the program state with a new latent variable taking a particular series of values over time, where this latent variable may be used to construct a predicate matching the trigger times of the update. On-clauses that perform updates (value changes) to this latent variable correspond to *transitions* in the latent variable's automaton diagram (Figures 3 and 4). See Appendix C for details.

3 Experiments

We curated a benchmark suite of 26 AUTUMN programs to evaluate our synthesis algorithm, and stills from a subset of these programs are displayed in Appendix A. We summarize statistics about

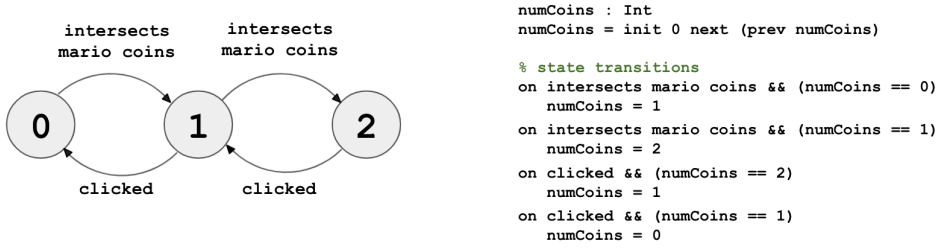

Figure 4: The latent state automaton learned for Mario (with similar variable renaming as above).

the synthesis procedure for each of these models in Table 1 in Appendix A. Though these results are still preliminary, our algorithm is able to synthesize a majority (24 out of 26) of these programs up to the success criterion defined in the Appendix, and is also able to discover interpretable latent state automata underlying these programs. For example, the automata learned for the Water Plug model (described in the Appendix) and the Mario model are shown in Figures 3 and 4.

References

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[2] Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9165–9174, 2019.

[3] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and J. Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *NeurIPS*, 2018.

[4] Alison Gopnik. Scientific thinking in young children: Theoretical advances, empirical research, and policy implications. *Science*, 337(6102):1623–1627, 2012.

[5] Alison Gopnik and Laura Schulz. Mechanisms of theory formation in young children. *Trends in cognitive sciences*, 8(8):371–377, 2004.

[6] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[7] Parthasarathy Madhusudan. Synthesizing reactive programs. In Marc Bezem, editor, *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings*, volume 12 of *LIPIcs*, pages 428–442. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.

[8] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

[9] Laura Schulz. The origins of inquiry: Inductive inference and exploration in early childhood. *Trends in cognitive sciences*, 16(7):382–389, 2012.

[10] Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022):1279–1285, 2011.

[11] Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, January 2017.

Appendix

A Full Results

Model Name	Input Length (Frames)	On-Clause Count	Contains Latent State?	Synthesized?
PARTICLES	22	2	No	Yes
ANTS	24	3	No	Yes
CHASE	42	7	No	Yes
ICE	27	10	No	Yes
LIGHTS	24	2	No	Yes
MAGNETS	53	7	No	Yes
SPACE INVADERS	42	11	No	Yes
SOKOBAN	25	7	No	Yes
DISEASE	22	7	**Yes**	Yes
GROW	40	-	**Yes**	**No**
SANDCASTLE	32	-	**Yes**	**No**
BULLETS	54	18	**Yes**	Yes
GRAVITY I	19	9	**Yes**	Yes
GRAVITY II	24	14	**Yes**	Yes
GRAVITY III	27	32	**Yes**	Yes
GRAVITY IV	48	18	**Yes**	Yes
COUNT I	22	6	**Yes**	Yes
COUNT II	39	10	**Yes**	Yes
COUNT III	69	14	**Yes**	Yes
COUNT IV	109	18	**Yes**	Yes
DOUBLE COUNT I	156	10	**Yes**	Yes
DOUBLE COUNT II	94	18	**Yes**	Yes
WIND	21	9	**Yes**	Yes
PAINT	27	10	**Yes**	Yes
WATER PLUG	42	10	**Yes**	Yes
MARIO	81	19	**Yes**	Yes

As this work is still ongoing, we currently define a *success* in our evaluation if the synthesized program produces an output sequence that is *consistent* with the observed data, which means that it produces the correct observed sequence upon being evaluated on the given input user event sequence. In other words, we declare success even if the synthesized program does not exactly match the ground-truth generating program. In practice, we find that many of the synthesized programs are *almost* exactly the ground-truth program, except in cases of ambiguity in the observation sequence. For example, it is often the case that multiple events are a match for triggering an update function, such that an output program using any of them would technically match the observations. The way

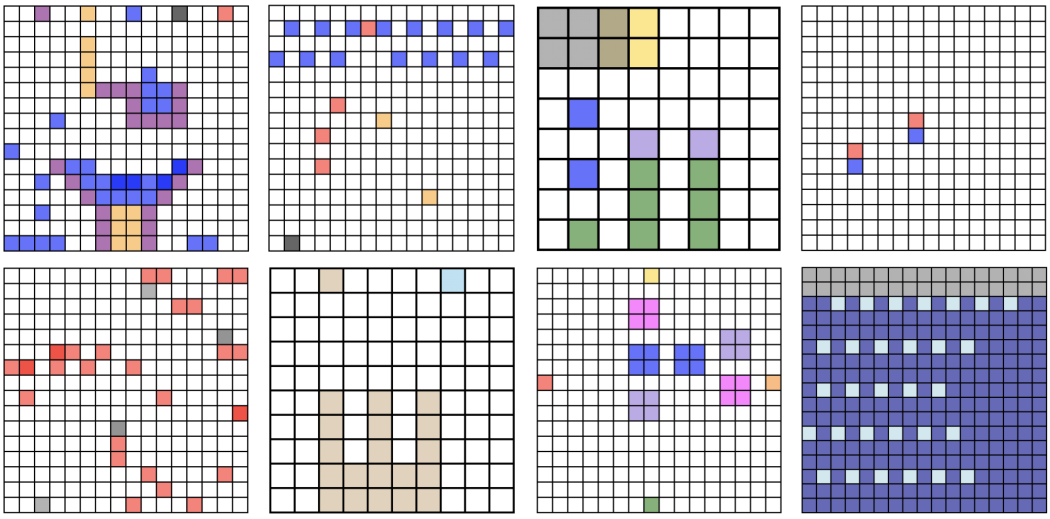

Figure 5: Stills from a selection of AUTUMN programs in the evaluation suite. From top-left to bottom-right: water interacting with a sink, a clone of Space Invaders, plants growing under sunlight and water, magnets, ants foraging for food, water destroying a sandcastle, an alternative gravity simulation, and snow falling in windy conditions.

to disambiguate between multiple matches is to check the hypothesis program's evaluation against additional, independent input sequences, or against a forecast of the existing input sequence into the future. Further, defining a *scoring function* that assigns all technically correct solutions a value related to how "good" that program is (perhaps according to a prior distribution over the space of programs) would also allow us to select from multiple options. These strategies remain part of our future work.

In addition, we have not yet fully standardized our evaluation, in that we presently modify both the event and update function search spaces between models to lessen the impact of ambiguity and improve performance, along with some low-level modifications to the algorithm. We give some additional details about these aspects of our evaluation in Appendix D, because many are better understood having read the more detailed description of the synthesis algorithm in Appendix C. We also note that we selected input observation sequences for each evaluation model manually, in a way that we knew was compatible with our heuristic-based automata synthesis algorithm. We have not yet fully studied the degree of manual input curation necessary for our synthesis procedure to succeed, though this (along with algorithm modifications to alleviate this limitation) remains an important priority for future work. Finally, while we take care to emphasize to the reader that our results are still preliminary and should not be the basis of overzealous speculation, the fact that the complex dynamics, and especially latent state automata, that underlie these models is being captured in some form by our procedure is exciting.

B The AUTUMN Language

Every AUTUMN program is composed of four parts (Figures 6 and 7). The first part defines the grid dimensions and background color. The second part defines *object types*, which are simply structs which define an object *shape*, or a list of 2D positions 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 third part defines *object instances*, which are concrete instantiations of the object types defined previously, as well as *latent variables*, which are values with type `int`, `string`, or `bool`. Object instances and latent variables are defined using a primitive AUTUMN language construct called `initnext`, which defines a *stream* of values over time via 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 the present

time. Further, the previous value of a variable may be accessed using the primitive `prev`, e.g. `prev var`. Indeed, the `next` expression frequently utilizes the `prev` primitive to express dependence on the past. For example, the definition of the Mario object in the example program from the introduction is `mario = init (Mario (Position 7 15)) next (moveDownNoCollision (prev mario))`, indicating that later values of Mario should move down one unit from the previous value whenever that is possible without collision.

Finally, the fourth segment of an AUTUMN program defines what we call *on-clauses*, which are expressed via the high-level form

```
on event
   intervention,
```

where `event` is a predicate (Boolean expression) and `intervention` is a variable update of the form `var = expr`, or multiple such updates. As suggested by the name *intervention*, an on-clause represents an *override* of the default modification to a variable that is defined in the `next` clause. In particular, when the `event` predicate evaluates to true, the new value of the variable `var` at that specific time is computed by evaluating the associated `intervention` instead of the standard `next` expression. Each on-clause may contain multiple update statements for different variables, and a single program may contain multiple on-clauses. In the latter scenario, the 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. In the rest of the discussion, we use the term *update function* to mean the same as *intervention*.

C The AUTUMNSYNTH Algorithm

We describe each of the four steps of the AUTUMNSYNTH algorithm below, with greatest space given to the fourth step of event and latent state synthesis, since that procedure represents the most novel aspect of our work. We note first that, for simplicity, we elect to define each object in our synthesized program with the trivial `next` expression, `prev obj`, and instead express default object behavior using the "default" on-clause

```
on true
   update_function.
```

When this on-clause is first in the list of on-clauses, it acts as the default behavior that takes place when later on-clauses evaluate to false and hence do not override it. Using this equivalent form in our synthesized programs simplifies our discussion by allowing us to describe synthesis as just involving determining object types, *initial* (not next) object and latent values, and on-clauses.

C.1 Step 1: Perception

In the perception step, each frame in the observation sequence is parsed into a set of object variables. Each object variable is characterized by a *shape*, which is a list of 2D grid positions relative to $(0, 0)$ that are each associated with a color, as well as an *origin*, which indicates the location of the object in the grid frame (the positions of the shape are translated by the origin to obtain the final rendering of each object). We use two different object parsing algorithms, each of which produces a different representation. We perform the rest of the synthesis procedure atop both of these parsing results one at a time, and take the output program from the first parsing using which the procedure succeeds.

The first parsing algorithm ("multi-cell") is based on a breadth-first search pixel crawler, which identifies groups of adjacent cells with the same color as multi-celled objects. This approach currently supports only uniform-colored objects instead of individual objects composed of multiple colors. Extending this algorithm to handle more diverse object renderings is an area of future work. Object types are extracted from the union of the parsed object sets over all frames by finding shapes that contain the same 2D positions, though not necessarily the same colors. Shapes that support multiple colors are described by object types that have a custom field $\langle color, \texttt{string} \rangle$, which allows individual instances of the object type to specify a particular color. The second parsing algorithm ("single-cell") simply identifies each colored cell in a frame as an individual object, with the set of object types being the set of single-celled shapes each with a particular fixed color.

```
-- define sand and water types
object Sand dry:Bool {(Cell 0 0 (if dry then "gold" else "darkgold"))}
object Water {(Cell 0 0 "blue")}

-- dry sand moves like a solid (down), wet sand like a liquid (down or side)
sand : List Sand
sand = init (list (Sand (Pos 2 10)) (Sand (Pos 2 9)) /* . . . */ (Sand (Pos 6 6)))
    next (updateObj (prev sand)
                    obj -> if obj.dry
                              then nextSolid obj
                              else nextLiquid obj)

-- water moves like a solid (down)
water : List Water
water = init ()
     next (updateObj (prev water)
                     obj -> nextLiquid obj)

-- water and sand buttons
sandButton = init (Sand (Pos 2 0)) next (prev sandButton)
waterButton = init (Water (Pos 5 0)) next (prev waterButton)
active = init "sand" next (prev active)

-- sand becomes wet when adjacent to water
on adjacent sand water
  sand = updateObj (prev sand)
                   obj -> update obj dry false
                   obj -> adjacent obj water

-- clicking a free position adds the active type to that position
on clicked && (isFree click) && active == "sand"
  sand = addObj (prev sand) (Sand true (.. click position))

on clicked && (isFree click) && active == "water"
  water = addObj (prev water) (Sand true (.. click position))

-- clicking a button makes it active
on clicked sandButton
  active = "sand"

on clicked waterButton
  active = "water"
```

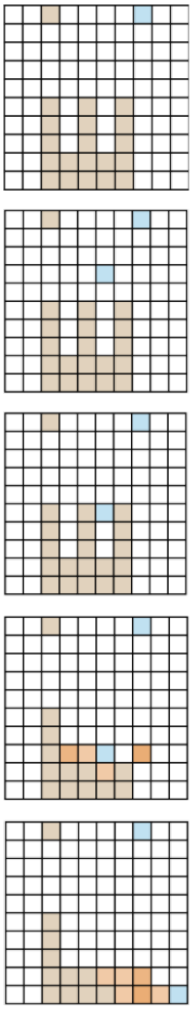

Figure 6: AUTUMN program for the Sandcastle model, with frames (ordered, but with time jumps in between) taken from a sample evaluation. The last of the top two buttons clicked dictates whether a sand particle or water particle is added upon clicking a free position. If water is adjacent to sand, it changes the `dry` field of the sand, so that it changes color and behaves as a liquid (i.e. moves to the lowest reachable height) as opposed to stacking.

```
-- define button and particle types
object Button color:String {(Cell 0 0 color)}
object Vessel {(Cell 0 0 "blue")}
object Plug {(Cell 0 0 "blue")}
object Water {(Cell 0 0 "blue")}

-- define button instances
vesselButton = init (Button "purple" (Pos 2 0)) next (prev vesselButton)
plugButton = init (Button "orange" (Pos 5 0)) next (prev plugButton)
waterButton = init (Button "blue" (Pos 8 0)) next (prev waterButton)
removeButton =  init (Button "black" (Pos 11 0)) next (prev removeButton)
clearButton = init (Button "red" (Pos 14 0)) next (prev clearButton)

-- define particle instances (lists)
vessels : List Vessel
vessels = init (list (Vessel (Pos 6 15)) /* . . . */ (Vessel (Pos 12 10)) )
    | | | | next (prev vessels))

plugs : List Vessel
plugs = init (list (Plug (Pos 8 15)) /* . . . */ (Plug (Pos 8 13)) )
    | | | next (prev plugs)

water : List Water
water = init (list)
    | | | next (updateObj (prev water) (-> obj (nextLiquid obj))))

-- define active particle (invisible state)
activeParticle : String
activeParticle = init "vessel" next (prev activeParticle)

-- clicking a particle button changes activeParticle (automaton transitions)
on clicked vesselButton
  | activeParticle = "vessel"
on clicked plugButton
  | activeParticle = "plug"
on clicked waterButton
  | activeParticle = "water"

-- clicking a free (uncolored) position adds an active particle there
on clicked && (isFree click) && (activeParticle == "vessel")
  | vessels = addObj (prev vessels) (Vessel (click.position))
on clicked && (isFree click) && (activeParticle == "plug")
  | plugs = addObj (prev plugs) (Plug (click.position))
on clicked && (isFree click) && (activeParticle == "water")
  | water = addObj (prev water) (Water (click.position))

-- clicking black button removes all plug particles
on clicked removeButton
  | plugs = removeObj (prev plugs) (-> obj true)

-- clicking red button removes all particles of any type
on clicked clearButton
  | vessels = removeObj (prev vessels) (-> obj true)
  | plugs = removeObj (prev plugs) (-> obj true)
  | water = removeObj (prev water) (-> obj true)
```

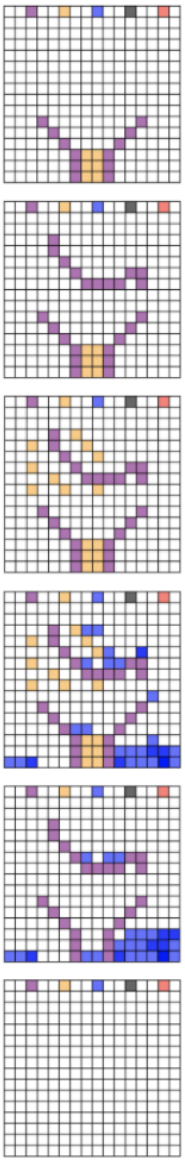

Figure 7: AUTUMN program describing the Water Plug model. In the first frame, the purple structure at the bottom is a vessel, and the orange structure is a plug that does not let water pass into the vessel. Excluding the top row of buttons, purple squares are vessel particles, orange squares are plug particles, and blue squares are water particles. Clicking an uncolored (free) position adds a particle to that position, where the type of particle depends on which of the top-left three buttons was clicked last. The right-side frames are in order (from top to bottom) but with time jumps: the user events during these jumps are the following: 1-2: clicking several free positions (new purple); 2-3: clicking top orange button then several free positions (new orange); 3-4: clicking top blue button then several free positions (new blue, though water moves down rather than being stationary); 4-5: clicking black button (orange removed); 5-6: clicking red button (all removed).

C.2 Steps 2 and 3: Object Tracking and Update Function Synthesis

Together, the second and third steps in the synthesis procedure answer the question, "What does each object do at each time step?" Concretely, this means identifying the *update function* undergone by each object in each frame to produce the object's rendering in the subsequent frame. The first element of answering this question is *object tracking* (Step 2), which involves assigning each object in a frame either to (1) an object in the subsequent frame, which is considered to be the transformed image of the object after the time step, or (2) no object, which means that the object has been removed after the time step. Multiple objects may not map to the same object in the subsequent frame, and further, objects in the subsequent frame without a pre-image in the previous frame are deemed to have been just *added* to the program in the current time. The algorithm that performs this mapping is based upon a heuristic that embodies the following prior assumption about object motion in AUTUMN: Objects are unlikely to move very far in a single time step. As such, the tracking algorithm performs assignments based on a proximity metric that tries to maximally assign objects in one frame to their closest objects (with the same object type) in the next frame.

Having determined which objects in a frame become which objects in the next, the *update function synthesis* procedure (Step 3) computes an AUTUMN expression, the update function, that describes every object-object, object-`null`, and `null`-object mapping. The `null` object simply represents a non-existent object, so an object-`null` mapping indicates object removal and a `null`-object mapping indicates object addition. To identify a matching update function, the procedure simply enumerates through a space of update function expressions, such as `obj = moveLeft obj` or `obj = nextLiquid obj`. Since it is often the case that there are multiple update functions that correctly describe a single mapping, we define a simple heuristic (based on how common an update function is across objects of a type) to select one update function from a set of correct options. At the end of the update function synthesis procedure, the synthesized update functions may be visualized in a matrix depiction, which we call the *update function matrix*. In the update function matrix, the rows represent object_id's, where objects are assigned the same object_id if one is transformed into the other over time, and the columns display the update function undergone by each object at each time.

While the above procedure is effective in some scenarios, we have actually found that the ambiguity in deciding which update function should be chosen for each object-object mapping has made using a slightly *relaxed* version of our heuristic more successful. In this case, rather than producing a single update function matrix as output, the update function synthesis procedure actually produces a *list* of possible matrices, each with a different combination of update functions across cells. We perform the rest of the synthesis procedure with each matrix until the first successful output program is produced.

C.3 Step 4: Event Synthesis

By this stage in the synthesis process, the object types, the object instance definitions, and the 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* that are necessary for the appropriate events to exist. At a high level, the event synthesis step answers the simple question, "*Why* does each object do what it does in each time step?"

To synthesize events, we first define a finite set of AUTUMN predicates, which roughly embodies a prior about what types of events are likely to be triggers of changes in the grid world. We call these predicates *atomic events*, because we ultimately enumerate both through the events themselves as well as *conjunctions* and *disjunctions* of those atoms when searching for a matching event. The atomic event set includes *global events*, including user events like `clicked`, `clicked obj1`, and `leftPress` as well as object contact events like `intersects obj1 obj2` and `adjacent obj1 obj2`, among other forms. These stand in contrast to the other type of event in the atomic event set, called an *object-specific event*, which takes different values for *distinct object_id's* in addition to distinct times. These events are effectively implemented as functions in a filter operation; for example, the event `obj.color == ''red''` is true for an object if the object is contained in the filtered list

```
filter (obj -> (obj.color == ''red'')) objects,
```

where `objects` denotes the set of all objects at the current time. We note that while the evaluation of a global event over time consists of a single vector of true/false values (one per time), the full evaluation of an object-specific event consists of a set of such vectors, one per distinct object_id.

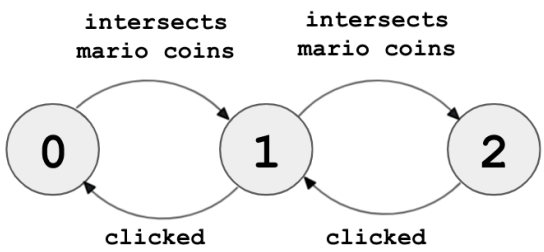

Figure 8: The latent state automaton for `numCoins` learned in the Mario example. Note that the learned automaton is the finite version of a more general Mario program, in which Mario can collect an uncountably infinite number of coins. Since we feed our synthesis procedure a finite trace, we cannot expect to learn the infinite automaton directly. In future work, however, we aim to capture this generalization via an *abstraction* step. Further, we note that the state numbers are simply *labels*, though they do happen to align with the contextual meaning of coin count. In particular, we could permute the three labels 0, 1, and 2 and still possess an automaton that explains the observed data. In fact, we actually did permute the labels of the raw automaton produced by our synthesizer to create this diagram, for ease of understanding. The raw output automaton had the 2 and 1 labels flipped.

Next, we describe the set of update functions for which we must find associated events. In our setting, we make the assumption that objects that belong to the same object type are all controlled by the same set of on-clauses. This means that if two objects both undergo the update `moveLeft` and the objects have the same object type, then a single event (on-clause) caused both of them to undergo the update. In contrast, if two objects undergo `moveLeft` and belong to different object types, we must synthesize a different event associated with each one, since a different on-clause caused each object type's update. 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.

Lastly, for each update function under consideration, we construct what is called an *update function trajectory*, which is a set of vectors $v \in \{-1, 0, 1\}^T$ that describes the times when the update function took place versus did not take place (T is the length of the observation sequence). There is one vector for each object_id with the object type under consideration. Each vector position is 1 if the update function took place at that time for that object_id, 0 if it did not take place, and -1 if it *may* have taken place but could have been *overridden* by another update function. This third scenario is interesting, and arises because we structure synthesized AUTUMN programs so on-clauses with update functions that are more frequent in the observed sequence are ordered before on-clauses with less frequent update functions. Thus, those later on-clauses with always override the earlier ones. With respect to event search, an event is a match for an update function if it is true for every time and object_id for which the update function trajectory vector is 1, and false whenever it is 0. The event may be either true or false when the corresponding update function trajectory value is -1.

Notably, if the number of unique vectors in an update function trajectory is 1, then the matching event may be a global event, because there is no variance based on object-specific features. Otherwise, if there is more than one unique vector in the trajectory, then the matching event must be an object-specific event, since the evaluated vector depends on the particular object_id. It is possible that a matching event may not be found in either of these cases, which signals that we must enrich the program state with new elements that were not used in the original event space. For simplicity, in the following section about latent state synthesis, we focus only on the case where the unmatched update function trajectory contains a single unique vector. This setting is called *global latent state synthesis*; the alternative setting, called *object-specific latent state synthesis*, is a straightforward extension.

C.3.1 Discovery of Latent State: Automata Synthesis

The input to the automata synthesis step is an update function trajectory composed of a single vector $v \in \{-1, 0, 1\}^T$. The goal of the procedure is to design the simplest latent state automaton that enables us to write a latent-state-based event predicate that matches v.

This problem statement is best illustrated with an example. Consider the Mario program from the introduction. As discussed, the bullet addition function does not have an event that perfectly matches its trajectory of occurrence versus non-occurrence (update function trajectory). Indeed, the event `clicked` co-occurs with every occurrence of a bullet addition, but there are times when `clicked` is true but no bullet addition takes place. These are the times when the number of coins possessed by Mario is zero, but there is no way to express this event using the existing program state representation, in which the only variables are the object variables. The latent state automaton described below alleviates this expressiveness limitation:

```
% initialize latent variable
numCoins : Int
numCoins = init 0 next (prev numCoins)
% define dynamics of variable
on intersects (prev mario) (prev coins)
   numCoins = (prev numCoins) + 1
on clicked && ((prev numCoins) > 0)
   numCoins = (prev numCoins) - 1
```

The automaton diagram associated with this AUTUMN description is shown in Figure 4. With this new latent variable `numCoins`, the event

```
clicked && ((prev numCoins) > 0)
```

suddenly is a perfect match for the update function trajectory vector v for bullet addition, so that we can write the following on-clause:

```
on clicked && ((prev numCoins) > 0)
   bullets = addObj (prev bullets) (Bullet (prev mario).origin).
```

Having developed a concrete example, we return to the challenging part of the problem, which is actually synthesizing the `numCoins` latent variable (along with its associated on-clauses) from the observed data alone. We do this by first framing our problem with respect to the classic formulation of automata synthesis given input-output examples. Classically, the problem of inductive automata synthesis is to determine the minimum-state automaton that accepts a given set of accepted input strings (positive examples) and rejects a given set of rejected input strings (negative examples) [1]. In our scenario, these positive and negative input "strings" may be determined from the sequence of program states (one per time) corresponding to the observation sequence. In particular, we consider the set of *prefixes* (sub-arrays starting from the first position) of the program state sequence that have, as their last element, a program state where the *optimal co-occurring event* is true. The optimal co-occurring event is defined to be 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`. We then partition 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 take place. The former set is the set of positive examples and the latter is the set of negative examples in our automata synthesis problem.

This definition of positive and negative input strings may be understood by considering the fact that, if there existed a latent state automaton that fit this specification, then the event

```
co_occurring_event && (latent_var in [/* accepting state labels */])
```

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.

To synthesize automata, we first recognize that a single automaton may provide the latent structure necessary to define the matching predicate for *multiple* distinct update functions, instead of just

one. We thus employ a heuristic algorithm that first *groups* update functions into sets for which one automaton will be sought. The algorithm then starts with a small automaton (i.e. small number of states), and tries to determine *transition events* that result in the appropriate program state sequences being accepted and rejected, for each update function in the group. If no such transition event can be found that perfectly partitions the input sets into positive and negative, then the number of states in the automaton is increased by *splitting* one of the existing states into two states. This process of searching for transition events and dividing a state into two on failure is repeated until success.

D Additional Evaluation Details

The in-progress nature of our submission manifests in the fact that our evaluations across different models are not yet fully standardized. Specifically, we modify the event and update function search spaces (i.e. by adding and/or removing some elements) between models to try and minimize scenarios of ambiguity, along with other low-level modifications to the algorithm. In particular, since our enumeration procedure through the event predicate space is currently just blind enumeration, we only compute conjunctions and disjunctions of atom expressions up to a depth of 2 atoms for performance reasons. As a result, we occasionally seed the event space with events that themselves contain conjunctions and disjunctions, so that combinations with greater depth may be effectively constructed with just depth-2 enumeration. This is necessary for models where the events have more than 2 atoms. We intend to use more sophisticated enumeration procedures, e.g. SyGuS solvers, to overcome these performance bottlenecks in the final version of the algorithm. Standardization of these variations are currently in progress and a priority for future work.

Lastly, instead of completely raw images, we currently send a slightly more processed version of the images to the synthesis engine as input, namely a list of 2D pixel positions with colors. The significance of this 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 (all Autumn renderings are partially transparent) 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 all RGBA values formed via overlaps of a finite number of colors, and the lists of colors that compose them. We will implement this procedure in the final version of the algorithm.

E Sample Synthesized Program

Below is an example raw output from the synthesis procedure, where we have added new line and space characters in the on-clause expressions to aid readability.

E.1 Ice Output

```
(program
  (= GRID_SIZE 8)
  (= background "white")
  (object ObjType1 (: color String) (list (Cell 0 -1 color) (Cell 0 0
    color) (Cell 1 -1 color) (Cell 1 0 color)))
  (object ObjType2  (list (Cell -1 0  "gray" ) (Cell 0 0  "gray" ) (
    Cell 1 0  "gray" )))
  (object ObjType3 (: color String) (list (Cell 0 0 color)))

  (: obj1 ObjType1)
  (: obj2 ObjType2)

  (: addedObjType1List (List ObjType1))
  (: addedObjType2List (List ObjType2))
  (: addedObjType3List (List ObjType3))

  (= obj1 (initnext (ObjType1  "gold"  (Position 0 1)) (prev obj1)))
  (= obj2 (initnext (ObjType2  (Position 4 0)) (prev obj2)))

```

```
(= addedObjType1List (initnext (list) (prev addedObjType1List)))
(= addedObjType2List (initnext (list) (prev addedObjType2List)))
(= addedObjType3List (initnext (list) (prev addedObjType3List)))


(: time Int)
(= time (initnext 0 (+ time 1)))

(on clicked (= obj1 (updateObj (prev obj1) "color" "gold")))
(on (& clicked (== (.. (prev obj1) color) "gold"))
    (= obj1 (updateObj (prev obj1) "color" "gray")))
(on left (= obj2 (moveLeft (prev obj2))))
(on right (= obj2 (moveRight (prev obj2))))
(on true
    (= addedObjType3List
       (updateObj addedObjType3List
          (--> obj (nextLiquid (prev obj)))
          (--> obj true))))

(on (== (.. (prev obj1) color) "gray")
    (= addedObjType3List
       (updateObj addedObjType3List
          (--> obj (moveDownNoCollision (prev obj)))
          (--> obj true))))

(on clicked
    (= addedObjType3List
       (updateObj addedObjType3List
          (--> obj (updateObj (prev obj) "color" "blue"))
          (--> obj true))))

(on (& down (== (.. (prev obj1) color) "gray"))
    (= addedObjType3List
       (addObj addedObjType3List
          (ObjType3  "lightblue"  (move (.. obj2 origin) (Position 0
  1))))))

(on (& clicked (== (.. (prev obj1) color) "gold"))
    (= addedObjType3List
       (updateObj addedObjType3List
          (--> obj (updateObj (prev obj) "color" "lightblue"))
          (--> obj true))))

(on (& down (== (.. (prev obj1) color) "gold"))
    (= addedObjType3List
       (addObj addedObjType3List
          (ObjType3  "blue"  (move (.. obj2 origin) (Position 0 1)))
 ))))
```