Combining Functional and Automata Synthesis to Discover Causal Reactive Programs

Ria A. Das
MIT

Joshua B. Tenenbaum
MIT

Armando Solar-Lezama
MIT

Zenna Tavares
Columbia University

Abstract

Note: This is a working document for an ongoing project. The discussed results represent the current stage of the work, and are subject to change as we continue to develop the methods.

ACM Reference Format:
Ria A. Das, Joshua B. Tenenbaum, Armando Solar-Lezama, and Zenna Tavares. 2022. Combining Functional and Automata Synthesis to Discover Causal Reactive Programs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In the last decade, the traditional view of program synthesis as a technique for automating programming tasks has expanded with the growth of the following hypothesis: Programs, with their unique ability to compactly and interpretably represent a wide variety of structured knowledge, may also be an important *model representation* in artificial intelligence (AI) systems. Recent work has demonstrated the potential of using programs as a modeling mechanism in a number of domains, such as learning rule-based programs describing biological data and synthesizing computer-aided design (CAD) programs from 3D drawings.

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. 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.

While there are many possibilities for the choice of theory representation in AI systems, programs offer the benefits that they can often be synthesized from small data (sample-efficiency) and that their concise, modular form often gives them strong generalization properties. These features have made program synthesis especially popular in cognitive AI as a route to building artificial agents that learn theories from observation as effectively as humans.

Despite the promise of formulating theory induction as program synthesis, existing methods of program synthesis are not yet suited to capture the richness of the space of theories that humans can learn from data, be it scientific or casual. One critical limitation is that many real world phenomena are *reactive*, time-varying systems, which update in *reaction* to new inputs at every time. However, current methods of inductive program synthesis—synthesizing programs from input-output examples—cannot synthesize non-trivial reactive models. This is because *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.

Specifically, most existing inductive program synthesis approaches are purely *functional*, meaning that both the inputs and outputs are fully observed, and the task is to construct a *function* taking one to the other. In other words, there are no concerns about identifying latent state, as the inputs and outputs are fully known. In a few other cases, inductive synthesis has also been applied to tackle the setting of *unsupervised learning*, in which hidden (latent) state representations are learned from partially observed inputs. However, neither of these method classes attempt to solve the full latent state learning problem that underlies the reactive setting. There, not only *what* the latent state representation is for every input (time point) must be learned, as is the case in unsupervised learning, but also *how* that latent state *evolves* over time must be identified, in the form of programmatic rules.

For concreteness, we consider the simple yet rich domain of Atari-style, time-varying 2D grid worlds (Figures 1, 2, and 3), which demonstrates these shortcomings of inductive program synthesis. This particular domain is of great interest in the AI and cognitive science communities, drawing its relevance from the fact that humans are able to learn *causal theories*—full explanations of which stimuli *cause* which changes

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

in the environment—of grid worlds incredibly quickly, a feat yet to be replicated by AI.

In the Mario-style game in this domain that is shown in Figure 1, 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. Notably, the number of coins that the agent possesses is not displayed anywhere on the grid at any time, so the only way to write a program that models this behavior is to define an *unobserved* or *latent variable*, which tracks the number of coins (bullets) possessed by the agent. In other words, there is no way to express *why* bullet addition takes place using just the current visible state of the program: the objects (with their locations and shapes) and current user action (click, key press, or none). Instead, we must define an *invisible* variable that can distinguish between two grid frames that are visually equivalent, but in which the agent has collected different numbers of coins (zero vs. some). Synthesizing this latent variable involves both identifying 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. Crucially, learning this dynamical latent state-based program from observations alone (a sequence of grid frames and user actions) is not feasible with standard techniques.

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 communities within programming languages: the *functional synthesis* and *automata synthesis* communities. Specifically, we show that we can inductively synthesize 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 observation, the automata synthesis procedure is called. The automata synthesis procedure is so named because the time-varying latent state in a reactive system can be viewed as a *finite state automaton*, where the labels on the automaton transitions are predicates in the underlying domain-specific language (DSL) used for synthesis (Figure 4). At a high level, based on the specifics of how the functional synthesis step failed, the automata synthesis procedure *enriches* the original program state with particular new latent structure (e.g. a time-varying latent variable like number of coins) that then allows that functional step to succeed.

By combining functional and automata synthesis techniques, our approach expands the horizon of synthesis problems that can be solved by either method alone. In particular, while the functional synthesis community has demonstrated impressive performance at synthesizing complex functional transformations from input-output data, the applicability of their techniques is limited by the fact that they cannot synthesize state-based models, including reactive systems, which are plentiful in the real world. On the other hand, the automata synthesis community has seen great success at synthesizing finite-state automata or *transition systems* from traces, but their methods do not scale to domains with intricate functional data transformations or very large numbers of states (which are often more compactly represented using program abstractions).

We suspect that this concept of 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 the particular domain of 2D Atari-style grid-worlds. We develop a DSL called Autumn (from *automaton*) that is designed to concisely express a variety of causal dynamics within these grids. 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. Since Autumn programs encode causal dynamics, this synthesis problem is one of *causal theory induction*, and is important in both cognitive science and AI. These fields aspire to the goal of developing an artificial agent that can learn causal theories as well as humans can, for which our hybrid functional-automata synthesis approach offers a potential route.

Our synthesis algorithm, named AutumnSynth, has three variant implementations, each differing in the algorithm used to perform automata synthesis from observed data. Two of these algorithms rely on the Sketch system to discover a minimal latent state automaton from examples, while the third algorithm is a heuristic that greedily searches through the space of automata. We construct a benchmark suite of 31 Autumn programs designed to express the diversity of time-varying causal models that may be manifested in 2D grids, and evaluate our algorithm implementations on this benchmark. Though subject to change as the work progresses, in our preliminary results, we find that our heuristic algorithm outperforms both Sketch implementations in both accuracy—it solves the majority of the benchmarks—and runtime—taking seconds to a few hours—especially on benchmarks with large automata, signaling the promise of our formulation. In sum, we make the following contributions:

(1) a novel inductive program synthesis algorithm that learns causal reactive programs from observation data (AutumnSynth);
(2) a guiding example of how to design synthesis algorithms that integrate functional and automata synthesis, enabling synthesis of programs beyond the scope of either alone; and
(3) a benchmark dataset of Autumn programs to spur the development of further algorithms in this space.

2 Overview

In this section, we briefly describe the AUTUMN language and AUTUMNSYNTH algorithm, and walk through a concrete execution of the algorithm on the Mario program described in the introduction.

2.1 The AUTUMN Language

AUTUMN was designed to concisely express a rich variety of causal mechanisms in interactive 2D grid worlds (Figure 2). These mechanisms range from distillations of real-world, everyday causal phenomena, such as water interacting with a sink, plants growing upon exposure to sunlight, or an egg breaking upon being dropped, to video game-inspired domains such as Atari's Space Invaders. The language is *functional reactive* — it augments the standard functional language definition with primitive support for temporal events.

Every AUTUMN program is composed of four parts (Figure 3). 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 that time. Further, the previous value of a variable may be accessed using the primitive `prev`, e.g. `prev var`. The `next` expression frequently utilizes the `prev` primitive to express dependence on the past. For example, the definition of the agent object in the Mario program from the introduction is `agent = init (Agent (Position 7 15)) next (moveDownNoCollision (prev agent))`, indicating that later values of the agent 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 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*.

2.2 Synthesis Example

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 object types and instances are parsed from the observed grid frames;
2. ***object tracking***, which involves assigning each object in a frame to either (1) an object in the subsequent frame, deemed to be its transformed image in the next time, or (2) 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 found; and
4. ***cause synthesis***, in which 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, with greatest space given to the step of 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 (Figure 4).

2.2.1 Perception.

The object perception step first extracts the object types and object instances from the input sequence of grid frames. The object types are (1) a general single-cell type with a string 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 are a red single-celled object (agent) at position (7, 15); three yellow single-celled objects (coins) at positions

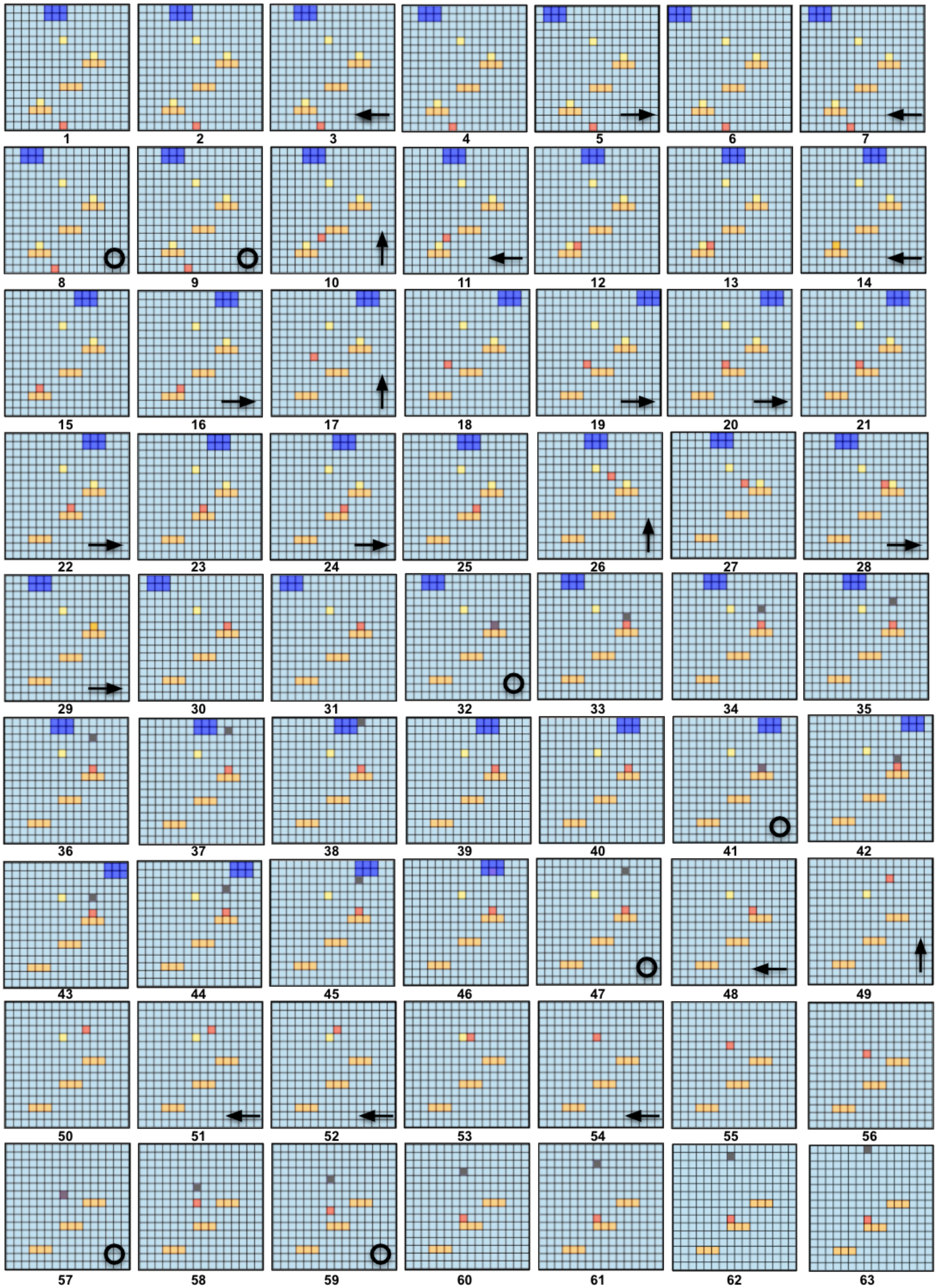


Figure 1. An observation trace from the Mario program. Black arrows indicate user keypresses and circles indicate user clicks.

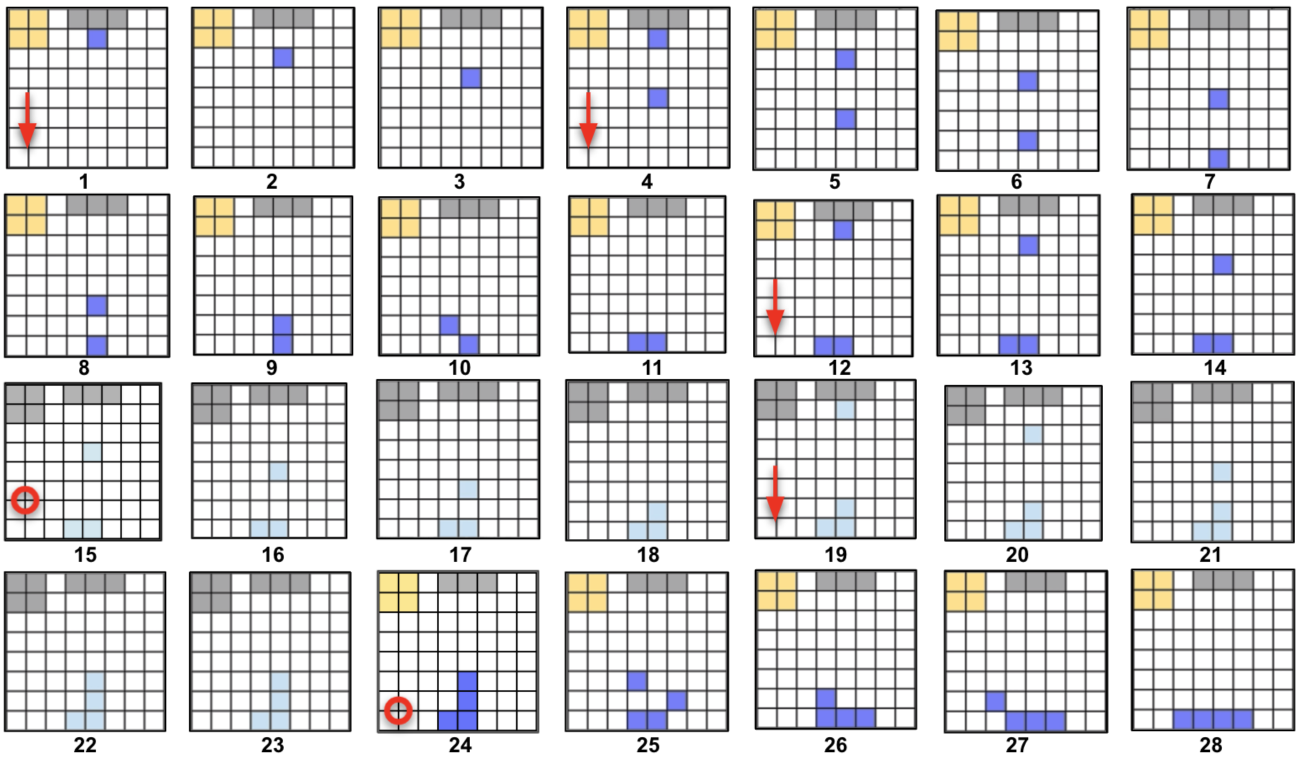


Figure 2. Sequence of grid frames from the Ice program. At times 1 and 4, the user presses down (red arrow), releasing a blue water particle from the gray cloud. The water moves down to the lowest possible height, moving to the side (time 10) if necessary to reach this height. The user presses down again 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.

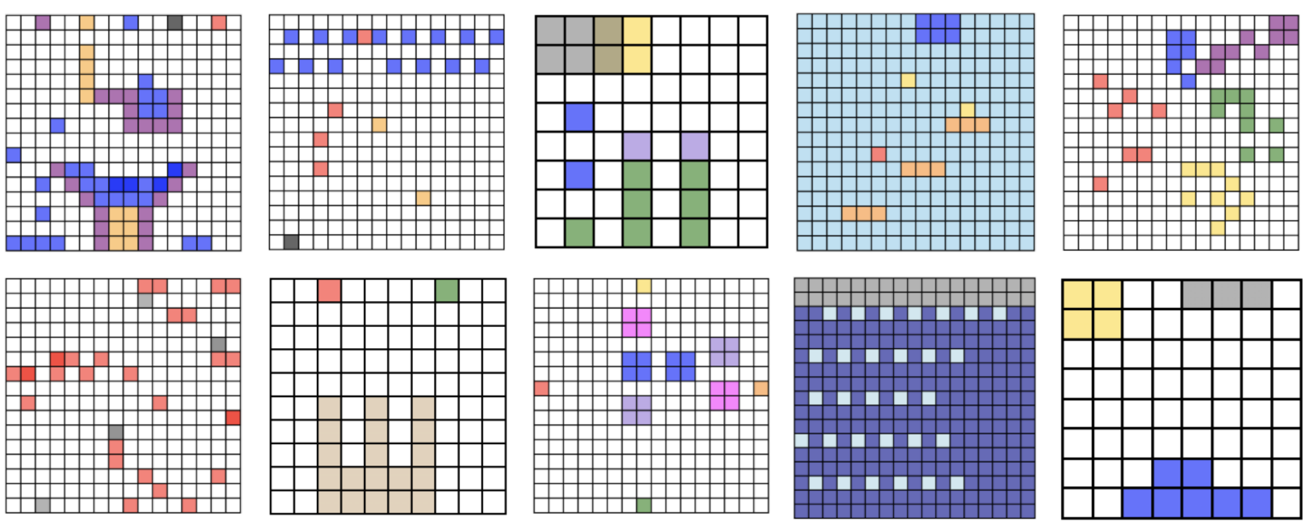
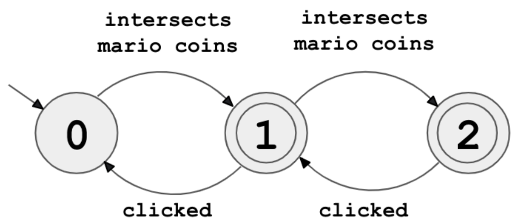


Figure 3. A sample of Autumn programs. Clockwise from top-left: water interacting with a sink and sink plug a clone of Space Invaders, plants growing under sunlight and water, a simplified implementation of Mario, a simplified clone of Microsoft Paint, a weather simulation, snow falling left or right with varying wind, an alternative gravity simulation, a sand castle susceptible to destruction by water, and ants foraging for food.



(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)

Figure 4. (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.

(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).

2.2.2 Object Tracking. Next, the object tracking step determines how each object in each grid frame *changes* to become a new object 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.

2.2.3 Update Function Synthesis. In the third step of update function synthesis, for each mapping between an object in one grid frame and an object in the next that is determined in Step 2, an Autumn expression is sought that describes that object-object mapping. For 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. 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 collects a set of these possibilities for each object mapping. Ultimately, one update function is selected as the single description for each object-object mapping during the final step of cause synthesis.

2.2.4 Cause Synthesis. Finally, the cause synthesis step searches for an Autumn event or predicate that triggers each update function identified in Step 3. For now, we will assume that we have already selected a single update function that matches each object-object mapping from the set of all possible update functions that do so; we will explain how we perform this selection process in Section 3. To find an Autumn event that triggers a particular update function, we collect the set of times that the update function takes place, and enumerate through a space of Autumn events until we find one that evaluates to true at each of those times. For example, say that the agent object in Mario 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 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 agent`, or `intersects bullet enemy`). Lastly, this event-finding process is complicated slightly by the fact that on-clauses may *override* each other, so perfect alignment between trigger event and observed update function is not always necessary. This nuance will be explained 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 (Position agent.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, we augment the program state by inventing a new latent variable that can be used to express the desired predicate.

Specifically, we proceed 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 when a click takes place, but some clicks do not add a bullet (specifically, at times 8, 9, 47, and 59). 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 of values during the true positive times. Calling the values taken by the latent variable during true positive times *accept values*, and those taken 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 Figure 4. The variable name `numCoins` is substituted to note the equivalence to a *number of collected coins* tracker.

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 Figure 5 (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 Formulation

Having provided a high-level description of the operation of our synthesis algorithm, we now formalize the full inductive synthesis problem for which our approach produces approximate solutions.

4 The Algorithm

We now give detailed descriptions of the steps of our algorithm introduced in Section 2. We focus on the latter two steps of the algorithm—update function synthesis and cause synthesis—referring the reader to the Appendix for full details of the object perception and tracking steps (Step 1 and 2), since they use more standard techniques and are not a central contribution of our work.

4.1 Step 3: Update Function Synthesis

Together, the object tracking (Step 2) and update function synthesis (Step 3) steps in the synthesis procedure answer the question, "What does each object *do* at each time step?" Object tracking first determines which objects in a grid frame become which objects in the next grid frame, as well as which objects were just added to or removed from the grid frame, across the full observation sequence. Then, the update function synthesis procedure computes an Autumn expression, the update function, that describes every object-object mapping. This includes update functions describing object addition and removal, which are represented as mappings with a `null` or non-existent object: a `null`-object mapping indicates object addition and an object-`null` mapping indicates object removal. These update functions will eventually become part of the on-clauses in the final output program.

To identify a matching update function, the procedure simply enumerates through a fixed, finite space of update function expressions, such as `obj = moveLeft obj` or `obj = nextLiquid obj`. Some of these update function options are simple translations, like `moveLeft obj` and `move obj -2 0`, while others are more *abstract* options that describe multiple concrete translations under different circumstances. For example, the `nextLiquid` function causes an object to move down when there is no object below it (i.e. there is no chance of collision), and to the left or right if there is an object below but there exists a path to a lower height in the left or right direction. There are typically multiple update functions in the space that describe any given object assignment, so the procedure collects all of these possibilities.

At the end of this process, the synthesized update functions may be visualized in a matrix depiction, which we call the *update function matrix* (Figure 5). 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 represent times in the observation sequence (in increasing order). Each cell in the update function matrix contains the set of possible update function expressions corresponding to that particular object_id at that particular time, or more precisely, those possibly undergone by the object *between* the frame at that time and the frame at the next time.

Ultimately, rather than a set of update functions for each object_id at each time, we want a single update function. This is because we will eventually search for Autumn predicates that evaluate to true at the times that each update function takes place, to form the on-clauses of the final synthesized program. Different choices for the single update function in each cell in the update function matrix changes the sets of times at which matching predicates must be true. For example, say that the sets of possible update functions undergone by an object in a three-grid-frame observation sequence are { `moveLeft` }, { `nextLiquid`, `moveLeft` }, and { `nextLiquid`,

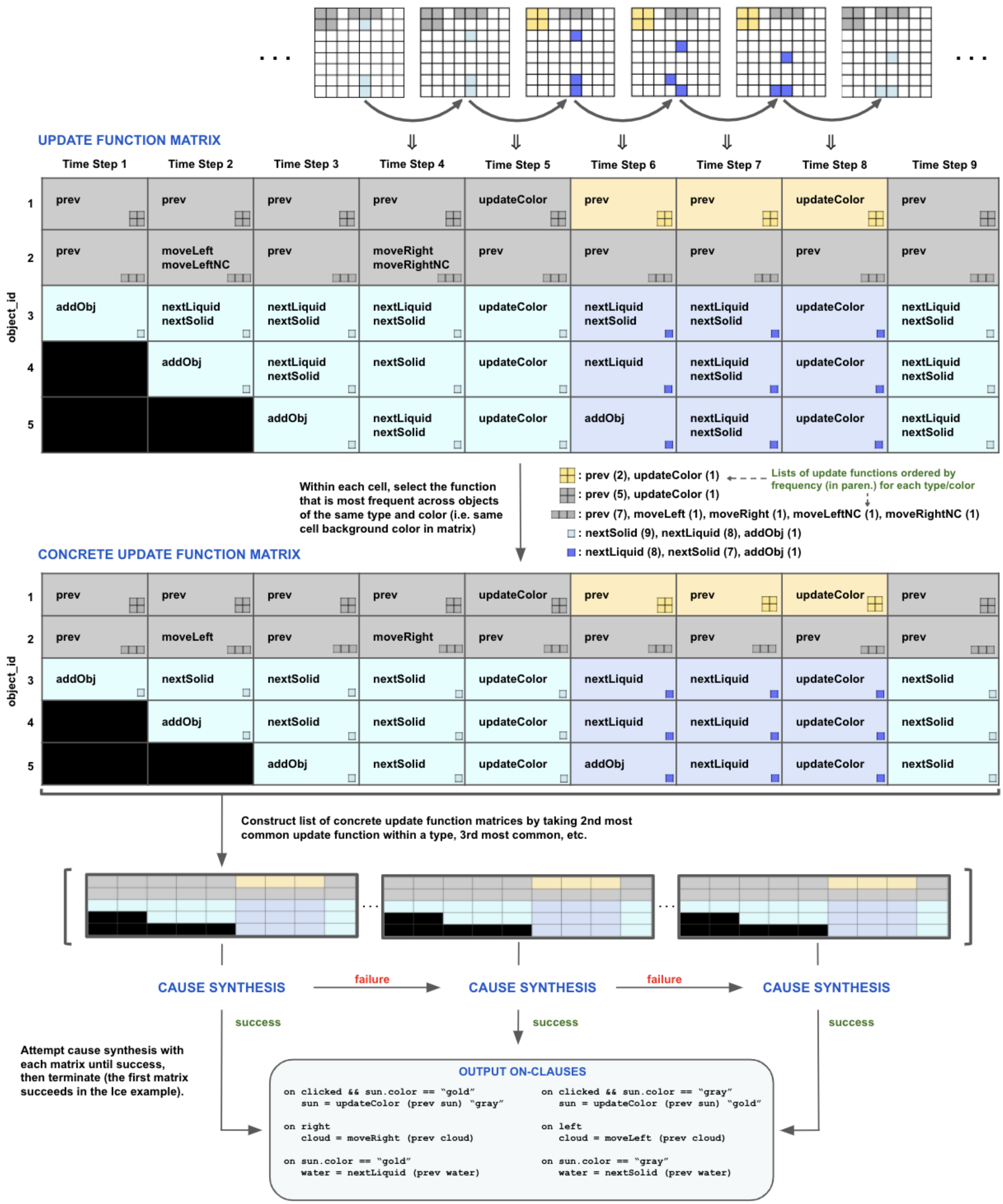


Figure 5. Update Function Synthesis. Each cell of the update function matrix contains a set of update functions that each describes the change undergone by the object with object_id equal to the row index during a particular time step (column index). A list of concrete update function matrices, with one update function per cell, is extracted via frequency-based heuristic.

`moveLeft` }. It is possible that there exists an event that is true at exactly the times 1, 2, and 3, which means that selecting `moveLeft` in all three matrix cells gives rise to a matching event. However, it is also possible that no event exists that is true exactly at time 1 or exactly at times 2 and 3, so the sequence of single update functions `moveLeft`, `nextLiquid`, `nextLiquid` does not produce matching events. Though a latent state automaton may possibly be constructed that alleviates this latter event search failure, automata search may also fail. Thus, the selection of a single update function in each cell of the update function matrix can make or break the success of the later cause synthesis step. Further, there might be multiple such selections that ultimately result in the success of the full synthesis procedure, but not every produced output program will be the optimal solution described in Section 3.

To handle this uncertainty with regard to which single update function selection within each matrix cell will allow matching events to be found for all update functions, we take the following approach. Let a *concrete update function matrix* be a "filtering" of the original matrix that contains just one option in each cell from the original options. There are a combinatorially large number of concrete matrices corresponding to any given full update function matrix. We select a small fixed set of concrete matrices from this large space using a heuristic that selects a single update function within a cell based on that update function's *frequency* across all rows of the matrix with the same object type. More frequent update functions across an object type are more likely to be selected than less frequent ones. The intuition behind this heuristic is that selecting more frequent update functions *minimizes* the number of distinct update functions within the concrete matrix for which corresponding events must be found. This can be viewed as trying to "maximally share" update functions across the cells of the matrix, resulting in an overall output program with fewer on-clauses if the cause synthesis step succeeds. This procedure is summarized in Figure 5; full details are given in the Appendix.

4.2 Step 4: Cause Synthesis

By this stage in the synthesis process, the object types, the object instance definitions, 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* that are necessary for the appropriate events to exist. At a high level, this step proceeds by enumerating through each concrete update function matrix in the list identified in the previous step, and searching for events and latent state that explain each distinct update function. If this process succeeds for a given concrete matrix, the overall algorithm terminates, returning the final program. If this process fails on the current concrete matrix, it is repeated on the next concrete matrix in the list until success or until the end of the list is reached, which indicates overall synthesis failure.

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 `left` 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.

Next, we describe the set of update functions for which we must find associated events in a given concrete update function matrix. 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 will 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 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 rest of the section, 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.

4.3 Step 4b: Automata Synthesis

The input to the automata synthesis step is a set of update function trajectories, one for each unmatched update function from the previous step. Each update function trajectory is a single vector $v \in \{-1, 0, 1\}^T$. The goal of the automata synthesis procedure is to construct the simplest latent state automaton that enables us to write latent-state-based event predicates matching each v. 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.

To start, we frame our overall 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). 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* (subarrays 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

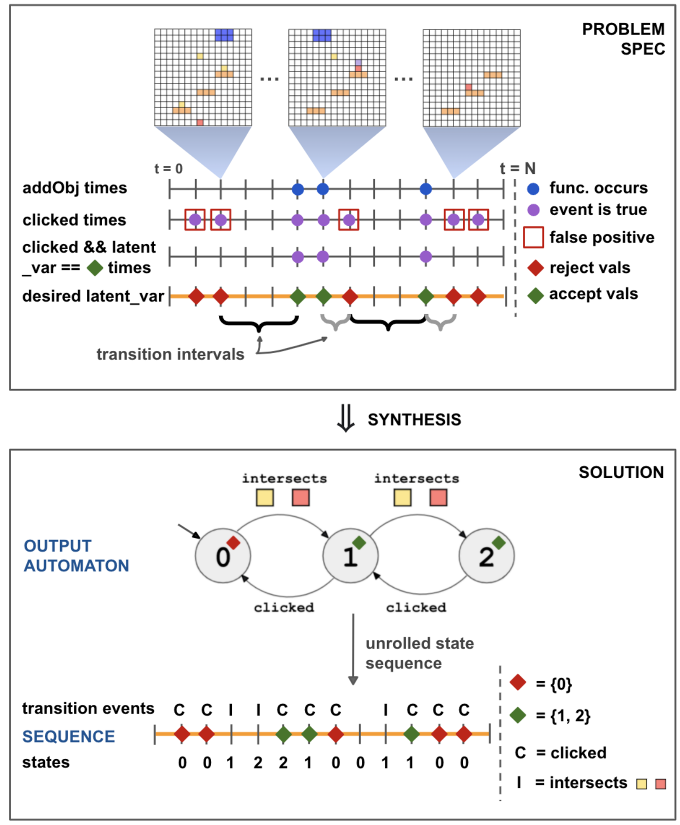


Figure 6. Bird's-eye view of the automata synthesis problem, using the example of the Mario program. The bullet addition update function, indicated by `addObj`, does not have a matching trigger event. The closest event is `clicked`, which co-occurs with bullet addition but also is true at false positive 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.

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

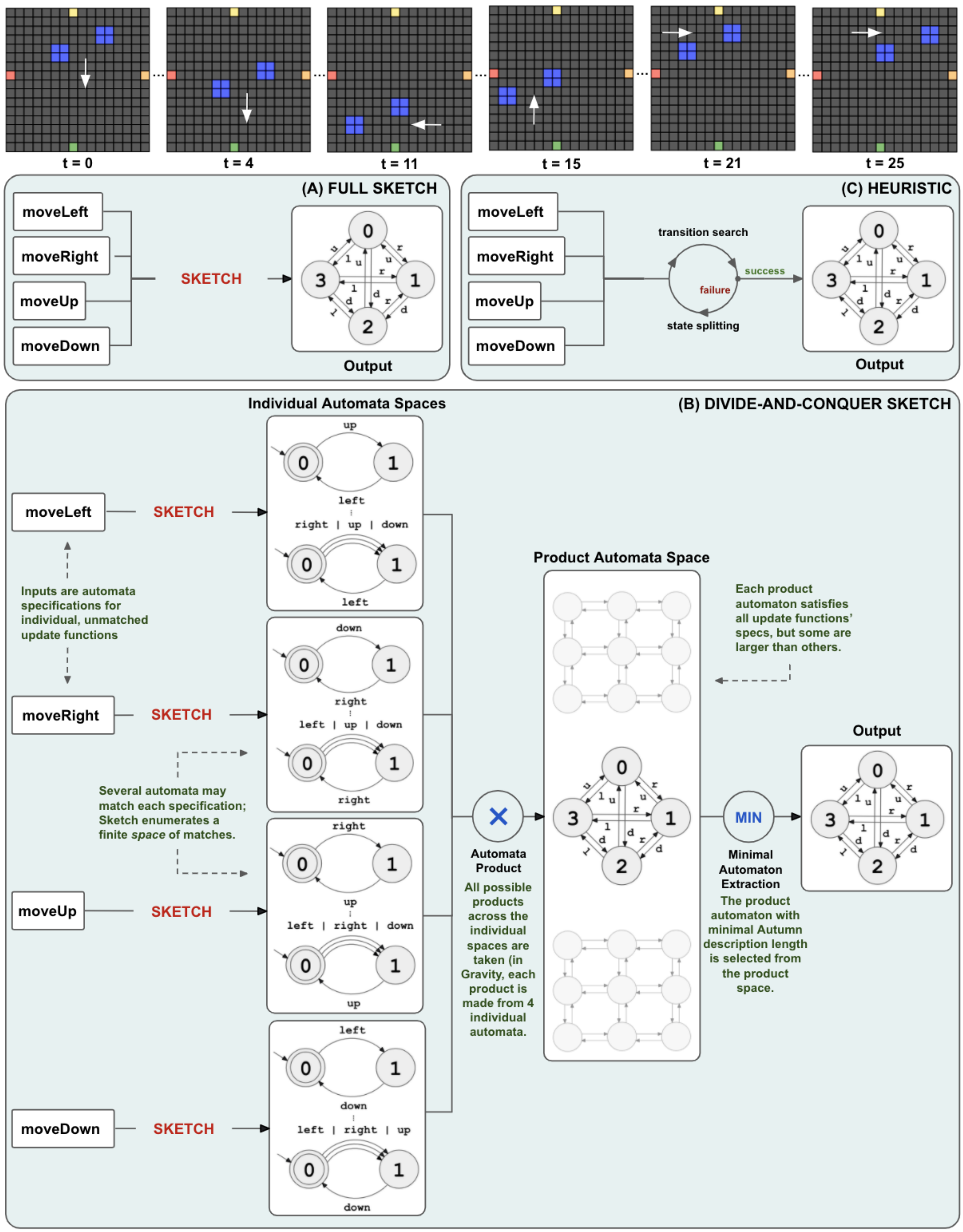


Figure 7. Three variant methods for automata synthesis, shown for Gravity I. The blue blocks move left, right, up, or down depending on the button last clicked. The transition label `left` abbreviates `(clicked leftButton)`, etc. See note in Sec. 4.3.2.

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.

Having discussed this simpler setting in which there is just one unmatched update function in need of latent state, we now return to the full problem setting, in which there may be 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 smallest automata satisfying individual update functions does not necessarily produce the smallest product automaton: It is possible that larger component automata will multiply to form this minimal product instead. Thus, optimizing each individual update function's automaton and multiplying is not a sufficient solution.

We now discuss three distinct algorithms for solving this inductive automata synthesis problem: Full Sketch, Divide-and-Conquer Sketch, and Heuristic. We note that at the current stage of this ongoing work, we synthesize 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). The reason we do not try to find one program-level automaton is because the human-written Autumn programs in our benchmark suite use a different latent variable for each type—a choice that appears to make the programs more human-understandable than having one large product—and these sets of type-level latent automata are also often more concisely expressed in the Autumn language than a single product. We will formalize this approach with respect to the overall synthesis objective of identifying the minimal Autumn program satisfying the observations in the final version of this work.

4.3.1 Algorithm 1: Full Sketch. In the Full Sketch approach, the complete multi-automata synthesis problem (for each object type) is encoded as a Sketch 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`.

4.3.2 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 to 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.

(A note about Figure 6b: For reasons of tractability, we employ a simple heuristic to downsize each individual update function's automata set before taking the product across all automata sets. At a high level, this heuristic identifies subsets of the full automata set that are *observationally equivalent* with respect to the given input observation sequence, and keeps just one automaton from each of these equivalence classes. This step is not shown in the figure. We will give a more detailed explanation of this procedure and definition of observational equivalence in the final version of this paper.)

4.3.3 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 known 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 transition search fails, one of the original states is split into two, and transition search is repeated. This process continues until a satisfying automaton is identified.

5 Preliminary Evaluation

5.1 The Autumn Benchmark Dataset

To evaluate our algorithm, we manually constructed a set of 31 Autumn programs, designed to collectively embody a rich variety of 2D causal mechanisms. These benchmark programs are described in Table 1 (Figure 8). Seven of the models do not contain latent state, and hence test only the functional component of our synthesis procedure, while the remaining 24 models contain latent state, thus also testing the automata synthesis component.

As our evaluation remains ongoing, for our preliminary results, we manually constructed an input user action sequence for each benchmark program, and ran the three synthesis algorithms—Full SAT, Divide-and-Conquer SAT, and Heuristic—on these sequences. We declared a success for a synthesis algorithm if it produced an output program that matches the observation sequence, though it need not be perfectly equivalent to the ground-truth program. Both of these aspects will be updated in our final evaluation, in which we plan to measure the success of our synthesis algorithms on input sequences generated by several human subjects interacting with the models, and define success to be the output program being semantically equivalent to the ground-truth program.

The results of this evaluation are shown in Table 2 (Figure 9) and Figures 10 and 11. While these results are subject to change as we continue to finalize our work, it appears that the Heuristic algorithm is currently most effective: It solves all but four of the benchmarks, and does so in less time than either of the other two algorithms, though the runtime is very similar to Full Sketch's runtime on many models. The Divide-and-Conquer Sketch algorithm is notably slower than both the Heuristic and Full Sketch algorithms on almost all of the models that all three methods solve. Further, while the vast majority of the programs synthesized by the Heuristic and Full Sketch algorithms either exactly or almost exactly match the ground-truth programs, many of the programs synthesized by the Divide-and-Conquer method do not generalize as accurately. This is a result of the fact that we do not enumerate the entire space of automata matching each individual update function before taking the product. We instead just enumerate a small, finite subset, so the computed product is often not optimal.

The most interesting two results in our evaluation are the following: (1) For four of the benchmark programs—Gravity III, Count III, Count IV, and Double II—both Sketch-based algorithms timed out after 24 hours without producing a solution, while the Heuristic algorithm solved all those models in minutes to hours: 2.3, 6.9, 118.3, and 17.3 minutes, respectively. The poor performance of the Full Sketch method on these models is due to the fact that the models' latent state automata are large (e.g. nine states and 24 edges for Gravity III), so the underlying SAT solver does not terminate. Divide-and-Conquer Sketch fails for the same reason, because while individual-update-function-level automata are often smaller than the overall automaton, in these models, each individual automaton is actually the same as the full automaton. Hence, Sketch again does not terminate in the Divide-and-Conquer framing. (2) For one benchmark program, Swap, the Full Sketch approach timed out after 24 hours, but the Divide-and-Conquer Sketch algorithm actually managed to find a solution in 21.4 minutes. (The Heuristic algorithm also solves this model, in 2.3 minutes.) The reason for this unusual result is that the Swap model has a latent state automaton with eight states and 64 edges, too large for the Full Sketch algorithm to handle, but which is the product of eight two- or three-state automata corresponding to the eight distinct update functions in the program. Sketch can more easily identify a two- or three-state automaton satisfying a specification, so the Divide-and-Conquer Sketch algorithm does this eight times and hence terminates successfully.

We also comment on the benchmark programs that none of our algorithms were able to synthesize. For these models, many of the fixes are lower-level modifications to the overall algorithm. For example, for the Grow II and Egg programs, an event predicate needed to express the program is actually just missing from the atomic event space we use for search, so it should be added to the space. Another limitation is that sometimes the optimal co-occurring event computed for a particular latent-state-based update function is incorrect, causing synthesis to fail. However, the second-best co-occurring event—that with the second smallest number of false positives rather than the smallest—may be correct, or the third-best, etc. This general kind of failure can be reduced by implementing a form of "multiplicity handling" with respect to co-occurring events, where instead of trying only the best event and terminating if it causes the rest of synthesis to fail, we try the top-k best events until one hopefully succeeds. These kinds of updates to our current algorithm are ongoing.

Finally, we emphasize that our benchmark results are still preliminary and are subject to change as we continue to modify both the Heuristic and the Sketch-based algorithms, including with the generalizations described above. Some of these modifications will affect all three algorithms' runtimes, like the previously described "multiplicity handling" generalization, while others will affect individual algorithms' runtimes. For example, optimizations to the Sketch implementations could decrease the Sketch-based algorithms' runtimes, while improvements that make the Heuristic algorithm less brittle/more general would increase the Heuristic algorithm's runtimes. More precisely, while the Heuristic algorithm works well on the current benchmark suite, the nature of it being a heuristic means that there are certainly classes of models on which it will fail, which we can patch somewhat with more intricate algorithms. These kinds of changes are likely necessary for the method to generalize both to other Autumn programs we may add to the benchmark suite, as well as externally-sourced programs like those discussed in Section 5.2. In addition, further thinking about our evaluation design, including potentially running the Sketch solver with a few different parameter options to fend against blowup, to ensure the fairest possible comparison between the three algorithms also remains part of future work. These modifications may result in different relative runtimes across the variant algorithms than we currently observe (e.g. potentially lower Sketch runtimes and higher Heuristic runtimes on some benchmarks). In our final evaluation, we will also average the runtimes over more trials; our current results are averaged over 2-4 runs, where the smaller benchmarks were run more times and the larger benchmarks run fewer times.

5.2 Generalization Beyond Autumn Programs

To further assess the generality of our techniques, we plan to run the three synthesis variants on a benchmark dataset that we did not ourselves construct. Using just the Autumn benchmark suite is akin to evaluating on only the "training set" for our algorithm, as AutumnSynth was designed with knowledge of these particular programs in mind. In particular, we will evaluate on the suite of Atari-style games created by Tsividis et. al. (http://pedrotsividis.com/tbrl.html). These games were written in the PyVGDL language for describing grid-world-based video games, and exhibit a number of differences from Autumn programs. These differences include that all the games run on 330 pixels by 900 pixel grids while most Autumn programs run on 16 by 16 grids. As a proof-of-concept that our method can synthesize these externally-sourced benchmark programs, we ran a version of the Heuristic algorithm with minor modifications on an observation sequence from the Tsividis et. al. corpus's Aliens program, shown in Figure 12. The algorithm succeeded, producing an output program with two *object-specific* latent automata describing objects moving at different *speeds*. We are currently generalizing lower-level details of our implementation so as to incorporate the modifications necessary for synthesizing this different flavor of models. Successfully synthesizing a large portion of this external benchmark will concretize the generality of our approach, and we are excited about pursuing this line.

	Name	On-Clauses	Automaton States	Automaton Transitions	Description
No Latent State	Particles	2	0	0	Brownian motion of single-cell objects.
	Ants	3	0	0	Ants foraging for randomly generated food particles.
	Chase	7	0	0	Agent evading randomly generated enemies.
	Magnets	13	0	0	Two magnets displaying attraction/repulsion.
	Space Invaders	12	0	0	A clone of Atari Space Invaders.
	Sokoban	7	0	0	A clone of Sokoban.
	Ice	10	0	0	Water particles behaving like solids vs. liquids.
Latent State	Lights	4	2	2	Clicking turns on/off a set of lights.
	Disease	7	2	2	Sick particles infect healthy particles.
	Grow I	11	2	2	Flowers grow upon water addition and sunlight.
	Grow II	11	2	2	Same as above, but plant stems grow longer.
	Sandcastle I	7	2	2	Water causes sand particles to turn liquid from solid.
	Sandcastle II	7	2	2	Same as above, but buttons match water/sand colors.
	Egg	7	2	2	An egg breaks upon being dropped from high enough.
	Bullets	17	8	12	Agent that can shoot bullets in four directions.
	Gravity I	9	4	12	Blocks move according to four gravity directions.
	Gravity II	14	7	15	Same as above, except colors of added blocks rotate.
	Gravity III	32	9	24	Blocks move according to nine gravity directions.
	Gravity IV	17	8	56	Same as Gravity I, except there are eight gravities.
	Count I	6	3	4	Weighted left/right movement, with two weights.
	Count II	10	5	8	Weighted left/right movement, with four weights.
	Count III	14	7	12	Weighted left/right movement, with six weights.
	Count IV	18	9	16	Weighted left/right movement, with eight weights.
	Double Count I	12	5	8	Weighted left/right/up/down, with four weights.
	Double Count II	20	9	16	Weighted left/right/up/down, with eight weights.
	Wind	9	3	4	Snow falls left, down, or right based on wind state.
	Paint	10	5	5	A simplified clone of MSFT Paint, with five colors.
	Mario	19	5	6	A Mario-style agent collects coins and shoots enemy.
	Mario II	19	7	7	Same as above, but enemy has two lives, not just one.
	Swap	40	8	64	Same as Gravity IV, but clicks also toggle two states.
	Water Plug	8	3	6	Water interacts with a sink and removable sink plug.

Figure 8. Descriptions of the 31 benchmark programs.

	Model Name	Input Length (Frames)	Output Length (Program Lines)	Heuristic Runtime	Sketch Runtime	D&C Sketch Runtime
No Latent State	*Particles*	22	21	12.0	N/A	N/A
	Ants	24	24	221.3	N/A	N/A
	Chase	42	33	51.9	N/A	N/A
	Magnets	53	43	60.8	N/A	N/A
	Space Invaders	42	52	147.2	N/A	N/A
	Sokoban	25	38	56.0	N/A	N/A
	Ice	27	45	3.6	N/A	N/A
Latent State	Lights	24	36	5.0	5.6	8.3
	Disease	22	31	5.3	5.8	6.2
	Grow	40	49	172.3	217.8	276.5
	Grow II	159	49	×	×	×
	Egg	31	43	×	×	×
	Sandcastle I	32	33	12.7	13.7	15.6
	Sandcastle II	32	33	×	×	×
	Bullets	54	54	32.4	45.3	⊥
	Gravity I	19	37	3.9	4.2	4.5
	Gravity II	24	50	10.1	10.5	14.9
	Gravity III	27	83	2.3	⊥	⊥
	Gravity IV	48	54	6.2	7.3	10.9
	Count I	22	31	2.5	2.8	3.6
	Count II	39	39	2.5	10.2	9.9
	Count III	69	47	6.9	⊥	⊥
	Count IV	109	55	118.3	⊥	⊥
	Double Count I	94	43	3.7	8.11	29.3
	Double Count II	156	59	17.3	⊥	⊥
	Wind	21	42	76.6	79.0	79.3
	Paint	27	39	9.5	9.8	21.7
	Mario	81	65	181.9	220.3	235.9
	Mario II	208	65	×	×	×
	Swap	44	99	2.3	⊥	21.4
	Water Plug	42	37	⊥	⊥	⊥

Figure 9. Table of input/output lengths and algorithm runtimes on each of the benchmark programs. A bottom symbol indicates timeout after 24 hours. An X symbol indicates that the benchmark's solution was outside the support of the synthesis algorithms (described in more detail in Section 5.1) and thus we did not time the algorithms on these benchmarks. We will add these evaluations in the final version of the paper, when we have added the generalizations that alleviate these limitations. Finally, the N/A's for the Sketch and D&C Sketch runtimes on the first seven benchmarks are there because those models do not possess latent state, while the three algorithms vary only in their latent automata synthesis procedures. Since we wanted to highlight the runtime differences arising from core automata synthesis differences instead of lower-level algorithmic choices needed to support them (which would be more prominent in models without latent state), we have only evaluated the Heuristic algorithm on these non-latent-state based models for our first evaluation.

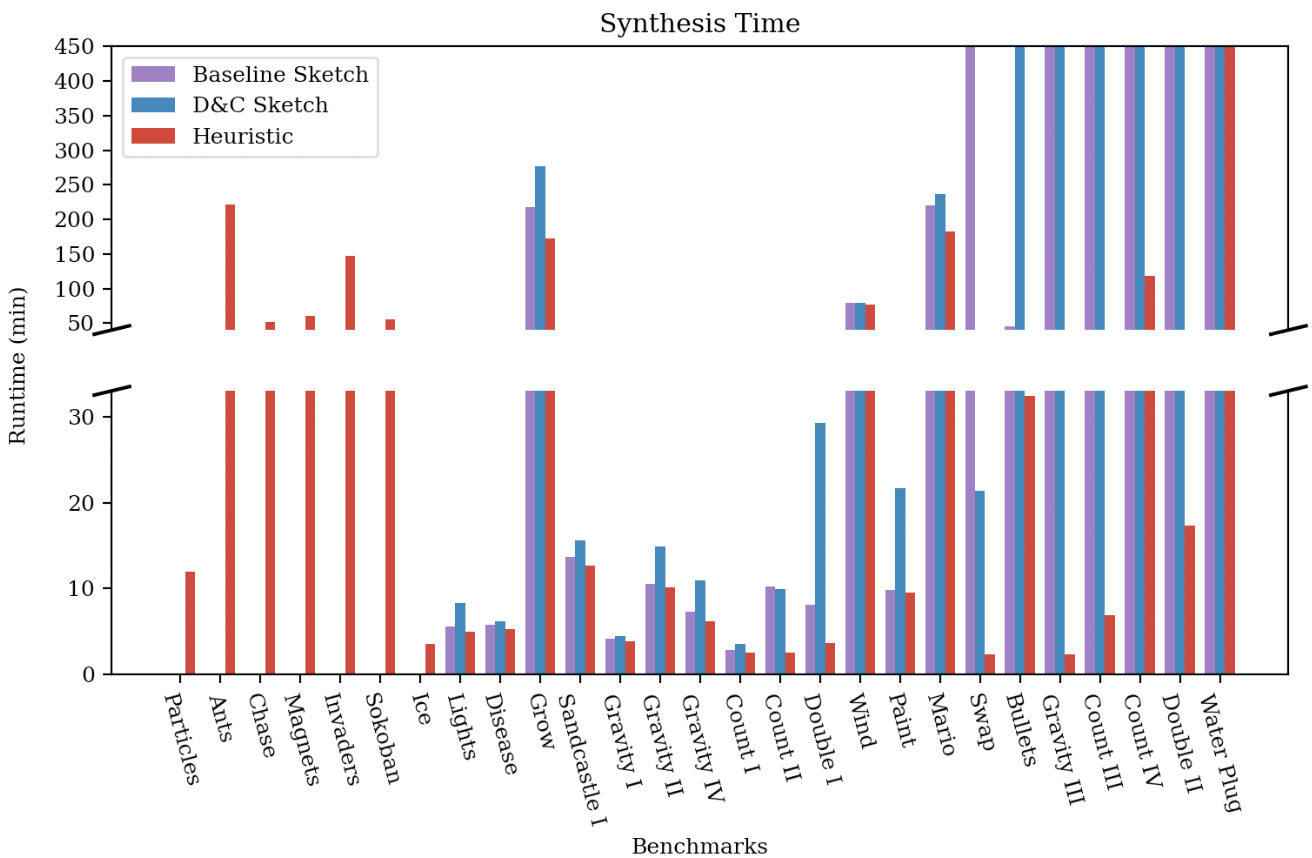


Figure 10. Runtimes for the variant AutumnSynth algorithms on each of the benchmark programs solved by at least one algorithm. Note that the first 7 benchmarks (left of the dashed line; Particles, Ants, Chase, Magnets, Invaders, Sokoban, and Ice) all do not contain latent state, so we currently evaluate only one of the algorithms (Heuristic) on them (see Figure 9 caption for further explanation). We also note that we ran the models with a timeout of 24 hours, so the runtimes that exceed the size of the plot did not finish before then, and that synthesis success is defined as producing a program that matches the observations—not necessarily being semantically equivalent to the ground-truth program. Finally, we note that while these results provide a snapshot of the current state of our project, they are subject to change as we continue to develop our variant algorithms. In particular, yet-to-be-implemented generalizations of the Heuristic method and optimizations to the Sketch-based algorithms could lead to different relative runtimes across the three algorithms (e.g. lower Sketch runtimes and higher Heuristic runtimes) for some benchmarks. See Section 5.1 for a more detailed discussion.

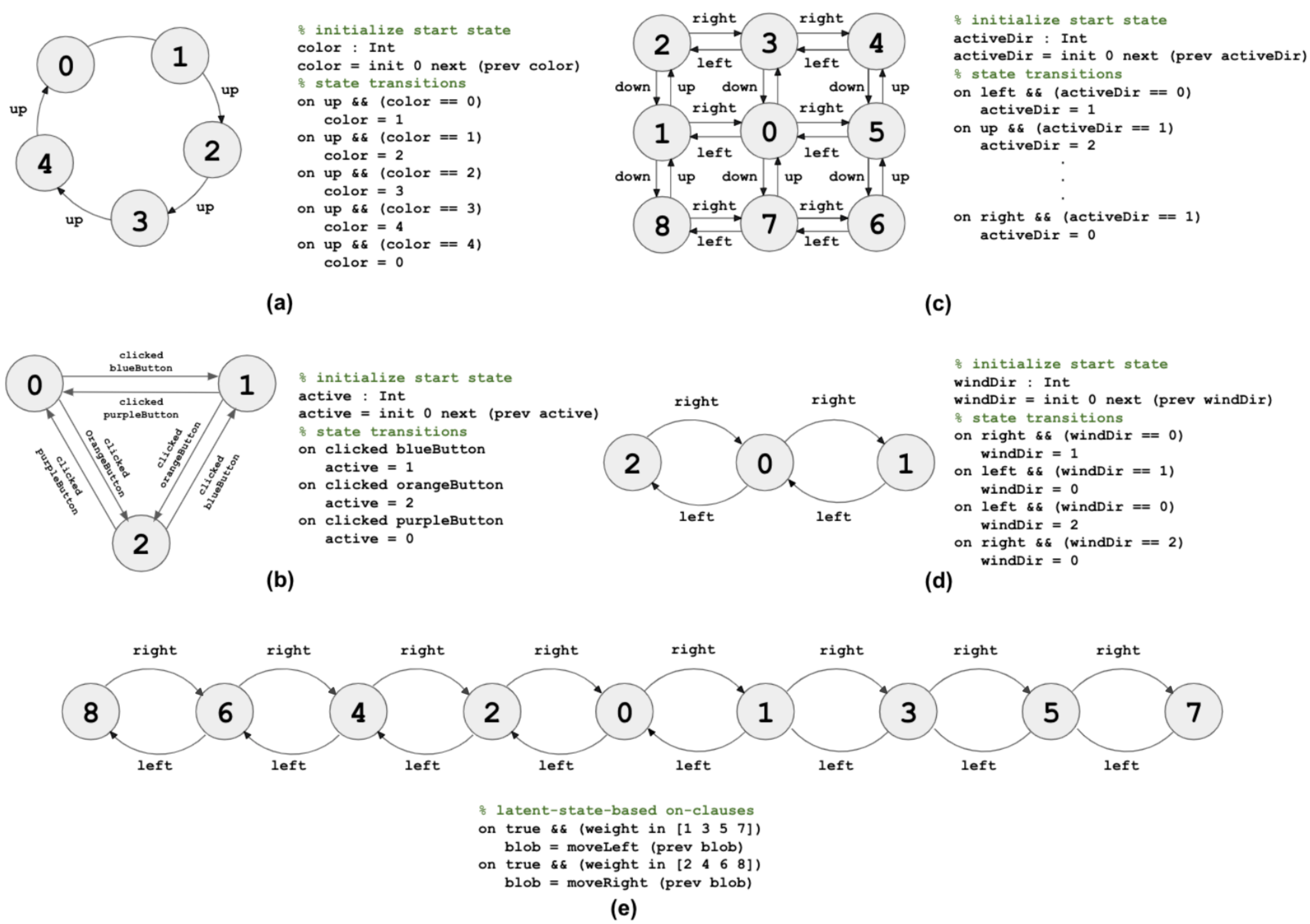


Figure 11. Sample latent state automata synthesized by AutumnSynth. **(a)** Paint model. Each state corresponds to a different color, indicating the color of the block added when a user clicks on an empty grid square. Pressing up cycles through the colors. **(b)** Gravity III model. Each state corresponds to one of the nine directions of motion formed by crossing three possible x-directions (-1, 0, 1) with y-directions (-1, 0, 1). **(c)** 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. **(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.

Figure 12. The Aliens program from the Tsividis et. al. corpus. Pressing arrow keys moves the blue agent left and right, and clicking causes it to shoot a pink bullet upward, as long as there are no other pink bullets already in the frame. Gold enemies are regularly created at the top-left corner, and move right once every three time steps. The enemies randomly shoot red bullets, which move down every two time steps. Pink bullets kill enemies, red bullets kill the agent, and both bullets destroy the gray shield blocks. The latent variables are the enemy and pink bullet speeds: the bullets do not move in sync but rather every two or three time steps from the time of their creation, so object-specific latent fields are used to track when they move.