

Combining Functional and Automata Synthesis to Discover Causal Reactive Programs

Ria A. Das
MIT

Armando Solar-Lezama
MIT

Joshua B. Tenenbaum
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/nnnnnnnn.nnnnnnnn>

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2022 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

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

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

113 In the Mario-style game in this domain that is shown in
 114 Figure 1, an agent (red) moves around with arrow key presses
 115 and can collect coins (yellow). If the agent has collected a
 116 positive number of coins, when the human player clicks,
 117 a bullet (gray) is released upwards from the agent’s position,
 118 and the agent’s coin count is decremented. Otherwise,
 119 clicking does nothing. Notably, the number of coins that the
 120 agent possesses is not displayed anywhere on the grid at
 121 any time, so the only way to write a program that models
 122 this behavior is to define an *unobserved* or *latent variable*,
 123 which tracks the number of coins (bullets) possessed by the
 124 agent. In other words, there is no way to express *why* bullet
 125 addition takes place using just the current visible state of
 126 the program: the objects (with their locations and shapes)
 127 and current user action (click, key press, or none). Instead,
 128 we must define an *invisible* variable that can distinguish be-
 129 tween two grid frames that are visually equivalent, but in
 130 which the agent has collected different numbers of coins
 131 (zero vs. some). Synthesizing this latent variable involves
 132 both identifying the variable’s initial value, as well as learn-
 133 ing *functions* that dictate when (on what stimulus) and how
 134 (increment, decrement, etc.) that value will change. Crucially,
 135 learning this dynamical latent state-based program from ob-
 136 servations alone (a sequence of grid frames and user actions)
 137 is not feasible with standard techniques.

138 To address this gap between current inductive program
 139 synthesis approaches and the reactive setting, we develop a
 140 novel program synthesis algorithm that unites two largely
 141 orthogonal communities within programming languages:
 142 the *functional synthesis* and *automata synthesis* communities.
 143 Specifically, we show that we can inductively synthesize re-
 144 active programs by splitting synthesis into two procedures,
 145 a functional synthesis procedure and an automata synthesis
 146 procedure. The functional synthesis step attempts to synthe-
 147 size the parts of the program that do not depend on latent
 148 state. If functional synthesis fails to synthesize a program
 149 component explaining an observation, the automata synthe-
 150 sis procedure is called. The automata synthesis procedure is
 151 so named because the time-varying latent state in a reactive
 152 system can be viewed as a *finite state automaton*, where the
 153 labels on the automaton transitions are predicates in the
 154 underlying domain-specific language (DSL) used for synthe-
 155 sis (Figure 4). At a high level, based on the specifics of how
 156 the functional synthesis step failed, the automata synthesis
 157 procedure *enriches* the original program state with particular
 158 new latent structure (e.g. a time-varying latent variable like
 159 number of coins) that then allows that functional step to
 160 succeed.

161 By combining functional and automata synthesis tech-
 162 niques, our approach expands the horizon of synthesis prob-
 163 lems that can be solved by either method alone. In particular,
 164 while the functional synthesis community has demonstrated

166 impressive performance at synthesizing complex functional
 167 transformations from input-output data, the applicability
 168 of their techniques is limited by the fact that they cannot
 169 synthesize state-based models, including reactive systems,
 170 which are plentiful in the real world. On the other hand, the
 171 automata synthesis community has seen great success at
 172 synthesizing finite-state automata or *transition systems* from
 173 traces, but their methods do not scale to domains with intri-
 174 cate functional data transformations or very large numbers
 175 of states (which are often more compactly represented using
 176 program abstractions).

177 We suspect that this concept of integrating functional and
 178 automata synthesis is valuable to a wide breadth of syn-
 179 thesis domains. In this paper, we demonstrate its value by
 180 instantiating it in the particular domain of 2D Atari-style
 181 grid-worlds. We develop a DSL called AUTUMN (from *au-*
 182 *tomaton*) that is designed to concisely express a variety of
 183 causal dynamics within these grids. The inductive synthesis
 184 problem addressed by our algorithm is: given a sequence of
 185 observed grid frames and corresponding user actions (clicks
 186 and keypresses), to synthesize the program in the AUTUMN
 187 language that generates the observations. Since AUTUMN pro-
 188 grams encode causal dynamics, this synthesis problem is one
 189 of *causal theory induction*, and is important in both cognitive
 190 science and AI. These fields aspire to the goal of develop-
 191 ing an artificial agent that can learn causal theories as well
 192 as humans can, for which our hybrid functional-automata
 193 synthesis approach offers a potential route.

194 Our synthesis algorithm, named AUTUMNSYNTH, has three
 195 variant implementations, each differing in the algorithm used
 196 to perform automata synthesis from observed data. Two of
 197 these algorithms rely on the Sketch system to discover a
 198 minimal latent state automaton from examples, while the
 199 third algorithm is a heuristic that greedily searches through
 200 the space of automata. We construct a benchmark suite of 31
 201 AUTUMN programs designed to express the diversity of time-
 202 varying causal models that may be manifested in 2D grids,
 203 and evaluate our algorithm implementations on this bench-
 204 mark. Though subject to change as the work progresses,
 205 in our preliminary results, we find that our heuristic algo-
 206 rithm outperforms both Sketch implementations in both
 207 accuracy—it solves the majority of the benchmarks—and run-
 208 time—taking seconds to a few hours—especially on bench-
 209 marks with large automata, signaling the promise of our
 210 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 algo-
 rithms that integrate functional and automata synthe-
 sis, 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.

166
 167
 168
 169
 170
 171
 172
 173
 174
 175
 176
 177
 178
 179
 180
 181
 182
 183
 184
 185
 186
 187
 188
 189
 190
 191
 192
 193
 194
 195
 196
 197
 198
 199
 200
 201
 202
 203
 204
 205
 206
 207
 208
 209
 210
 211
 212
 213
 214
 215
 216
 217
 218
 219
 220

221 2 Overview

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

227 2.1 The AUTUMN Language

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

237 Every AUTUMN program is composed of four parts (Figure
 238 3). The first part defines the grid dimensions and background
 239 color. The second part defines *object types*, which are simply
 240 structs which define an object *shape*, or a list of 2D positions
 241 each associated with a color, as well as a set of *internal fields*,
 242 which store additional information about the object (e.g. a
 243 Boolean *healthy* field may store an indicator of the object's
 244 health). The third part defines *object instances*, which are con-
 245 crete instantiations of the object types defined previously,
 246 as well as *latent variables*, which are values with type *int*,
 247 *string*, or *bool*. Object instances and latent variables are
 248 defined using a primitive AUTUMN language construct called
 249 *initnext*, which defines a *stream* of values over time via the
 250 syntax *var* = *init expr1 next expr2*. The initial value
 251 of the variable (*expr1*) is set with *init*, and the value at
 252 later time steps is defined using *next*. The *next* expression
 253 (*expr2*) is re-evaluated at each subsequent time step to pro-
 254 duce the new value of the variable at that time. Further, the
 255 previous value of a variable may be accessed using the prim-
 256 itive *prev*, e.g. *prev var*. The *next* expression frequently
 257 utilizes the *prev* primitive to express dependence on the past.
 258 For example, the definition of the agent object in the Mario
 259 program from the introduction is *agent* = *init (Agent* (Position 7 15) *next (moveDownNoCollision (prev* agent*))*, indicating that later values of the agent should
 260 move down one unit from the previous value whenever that
 261 is possible without collision.

264
 265
 266 Finally, the fourth segment of an AUTUMN program
 267 defines what we call *on-clauses*, which are expressed via the
 268 high-level form

269 on event
 270 intervention,

272 where *event* is a predicate and *intervention* is a variable
 273 update of the form *var* = *expr*, or multiple such updates. As
 274 suggested by the name *intervention*, an on-clause represents

275 an *override* of the default modification to a variable that is
 276 defined in the next clause. In particular, when the event
 277 predicate evaluates to true, the new value of the variable *var*
 278 at that specific time is computed by evaluating the associ-
 279 ated intervention instead of the standard *next* expression.
 280 Each on-clause may contain multiple update statements for
 281 different variables, and a single program may contain mul-
 282 tiple on-clauses. In the latter scenario, the on-clauses are
 283 evaluated sequentially, with the effect that later on-clauses
 284 may update a variable in a way that composes with updates
 285 from earlier on-clauses, or completely overrides it. In the
 286 rest of the discussion, we use the term *update function* to
 287 mean the same as *intervention*.

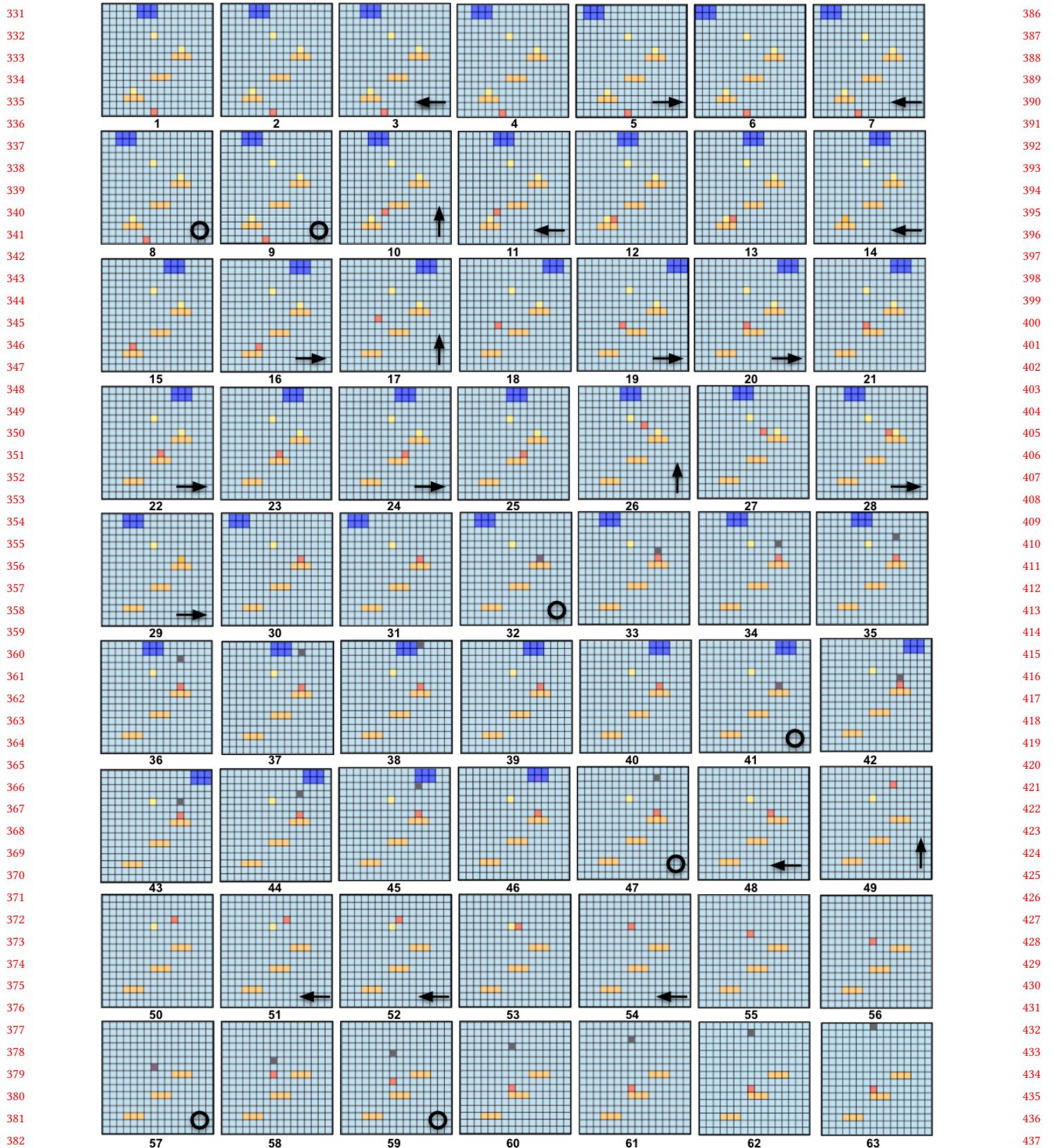
289 2.2 Synthesis Example

290 Synthesizing the correct AUTUMN program from observed
 291 data involves determining the object types, object instance
 292 and latent variable definitions, and on-clauses described pre-
 293 viously. The AUTUMNSYNTH algorithm, as an end-to-end
 294 synthesis algorithm taking images as input, consists of four
 295 distinct steps, each producing a new representation of the
 296 input sequence. These steps are

- 297 1. **perception**, in which object types and instances are parsed
 298 from the observed grid frames;
- 299 2. **object tracking**, which involves assigning each object in
 300 a frame to either (1) an object in the subsequent frame,
 301 deemed to be its transformed image in the next time, or
 302 (2) no object, indicating that the object was removed in
 303 the next time;
- 304 3. **update function synthesis**, in which AUTUMN expres-
 305 sions, called update functions, describing each object-
 306 object mapping from Step 2 are found; and
- 307 4. **cause synthesis**, in which AUTUMN events (predicates)
 308 that *cause* each update function from Step 3 are sought,
 309 and new latent state in the form of automata is constructed
 310 upon event search failure.

311 We give details for these steps in Section 4, with greatest
 312 space given to the step of cause synthesis, since that pro-
 313 cedure represents the most novel aspect of our work. First, we
 314 provide some intuition by briefly describing how these steps
 315 are used to synthesize the Mario program (Figure 4).

316 2.2.1 Perception. The object perception step first extracts
 317 the object types and object instances from the input sequence
 318 of grid frames. The object types are (1) a general single-cell
 319 type with a string color parameter corresponding to the (red)
 320 agent, (yellow) coin, and (gray) bullet objects; (2) a platform
 321 type that is a row of three orange cells; and (3) an enemy type
 322 that is a rectangle of six blue cells. A list of object instances is
 323 extracted from each grid frame in the input sequence, where
 324 an object instance describes the object's type, position, and
 325 any field values. For example, the object instances for the first
 326 grid frame are a red single-celled object (agent) at position
 327 (7, 15); three yellow single-celled objects (coins) at positions
 328 (3, 15), (4, 15), and (5, 15); and one gray single-celled
 329 bullet object at position (10, 15). The Mario program (Figure 4)

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

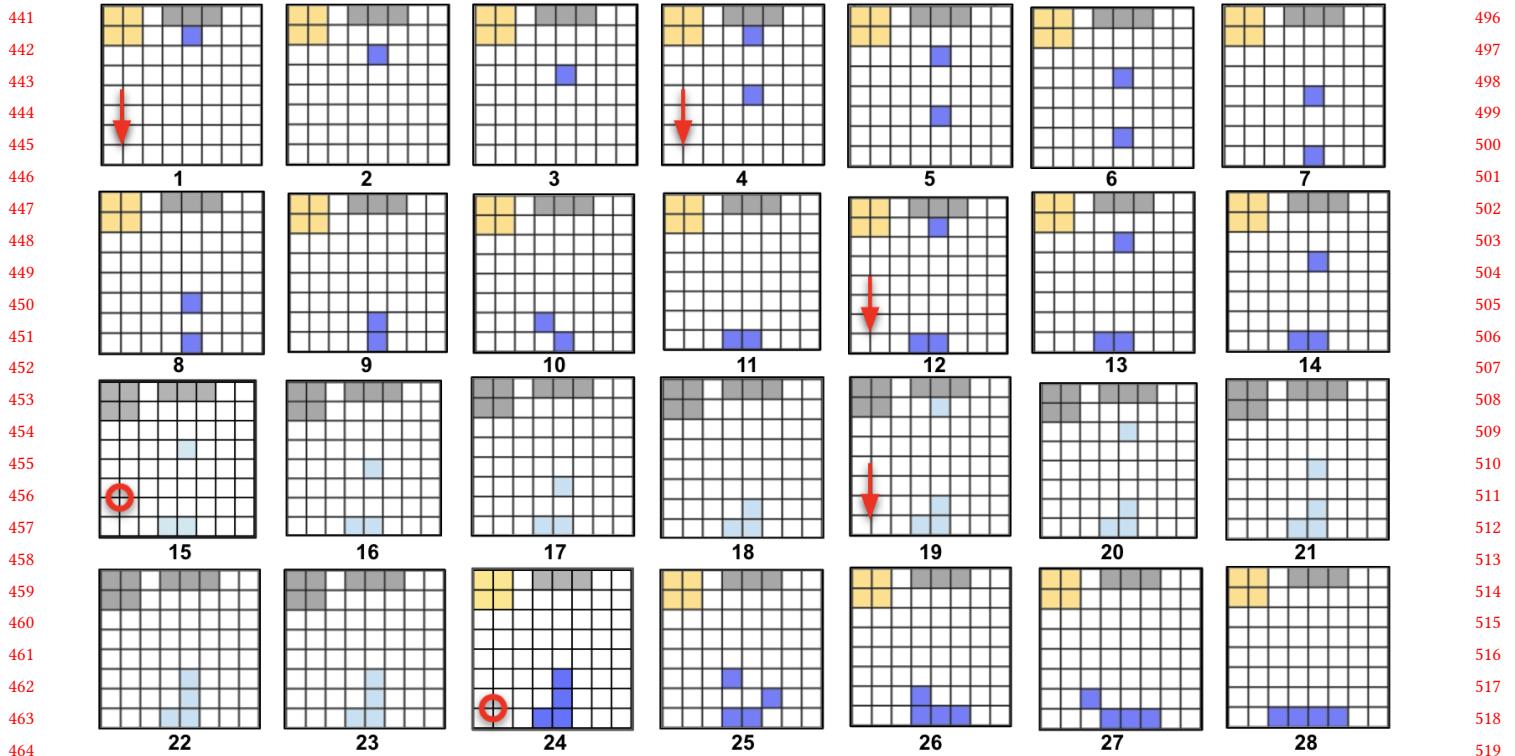


Figure 2. Sequence of grid frames from the Ice program. At times 1 and 4, the user presses down (red arrow), releasing a blue water particle from the gray cloud. The water moves down to the lowest possible height, moving to the side (time 10) if necessary to reach this height. The user presses down again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

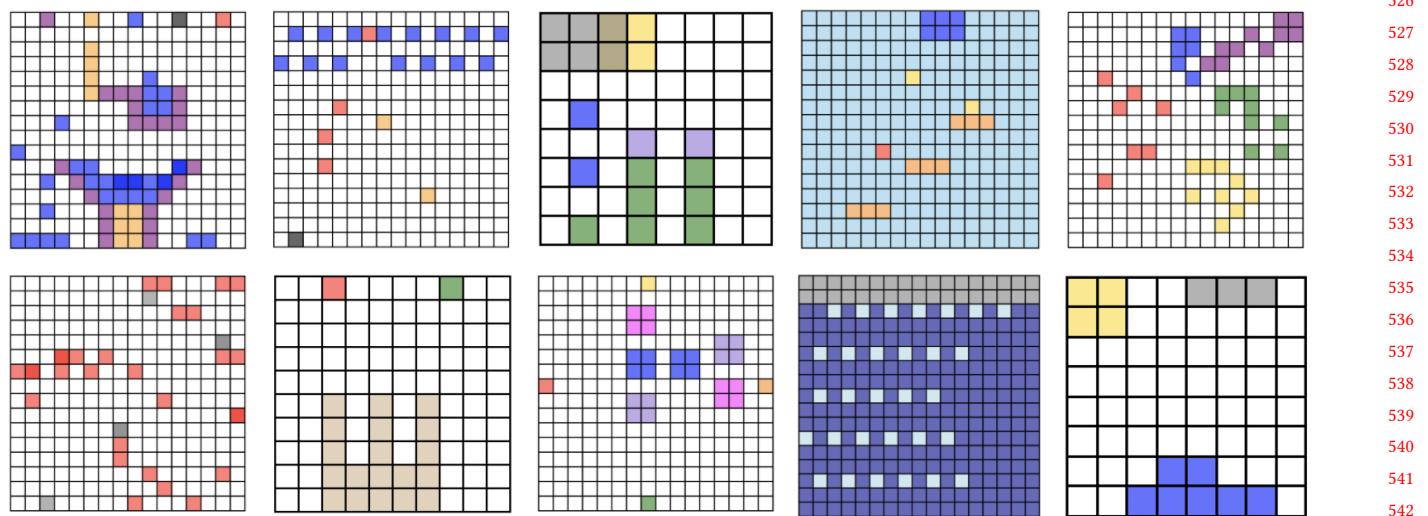


Figure 3. A sample of AUTUMN programs. Clockwise from top-left: water interacting with a sink and sink plug a clone of Space Invaders, plants growing under sunlight and water, a simplified implementation of Mario, a simplified clone of Microsoft Paint, a weather simulation, snow falling left or right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

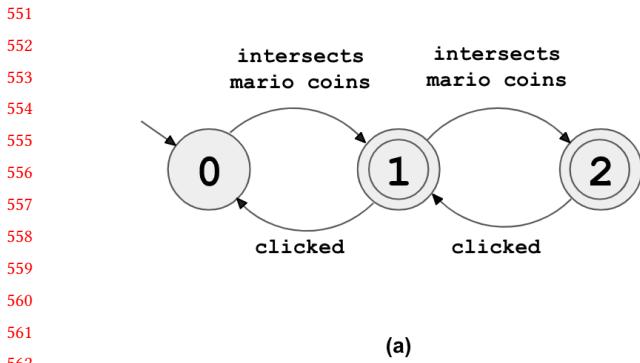


Figure 4. (a) Diagram of automaton representing the numCoins latent variable synthesized for the Mario program. The start value is zero, and the accept values (i.e. the values during which clicked causes a bullet to be added to the scene) are 1 and 2. (b) Description of the numCoins latent variable in the AUTUMN language.

564
565
566
567
568
569
570
571

(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).

572 **Object Tracking.** Next, the object tracking step de-
573 termines how each object in each grid frame *changes to*
574 become a new object in the next grid frame. For example, it
575 identifies that the agent object at position (7, 15) in the sec-
576 ond grid frame corresponds to the agent object at position (6,
577 15) in the third grid frame (i.e. it moved left). Intuitively, this
578 step *tracks* the changes undergone by every object across all
579 grid frames.

580 **Update Function Synthesis.** In the third step of up-
581 date function synthesis, for each mapping between an object
582 in one grid frame and an object in the next that is determined
583 in Step 2, an AUTUMN expression is sought that describes
584 that object-object mapping. For example, this step identifies
585 that the expression `agent = moveLeft (prev agent)` ac-
586 curately describes the change undergone by the agent object
587 between the first and second grid frames. Often, there are
588 multiple such expressions that match any given mapping. For
589 example, the agent's left movement during the first time step
590 might also be described by `agent = moveLeftNoCollision`
591 (`prev agent`) or `agent = moveClosest (prev agent)`.
Platform, where the latter indicates movement one unit
593 towards the nearest object of type Platform. The update
594 function synthesis step collects a set of these possibilities
595 for each object mapping. Ultimately, one update function
596 is selected as the single description for each object-object
597 mapping during the final step of cause synthesis.

598 **Cause Synthesis.** Finally, the cause synthesis step
599 searches for an AUTUMN event or predicate that triggers
600 each update function identified in Step 3. For now, we will
601 assume that we have already selected a single update func-
602 tion that matches each object-object mapping from the set of
603 all possible update functions that do so; we will explain how

604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660

```

% 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)

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.

661 Specifically, we proceed by finding the “closest” event in
 662 the event space that aligns with the update function. This is
 663 the event that *co-occurs* with every update function occur-
 664 rence, but may also occur during *false positive times*: times
 665 when the event is true but the update function does not occur.
 666 For bullet addition, this event is *clicked*, as every bullet is
 667 added when a click takes place, but some clicks do not add a
 668 bullet (specifically, at times 8, 9, 47, and 59). Having identified
 669 this closest event, our goal is then to construct a latent vari-
 670 able that acts as a finite state automaton that *switches* states
 671 between the false positive times and true positive times (i.e.
 672 the times when *clicked* is true and the update function oc-
 673 curs). To be precise, the new variable takes one set of values
 674 during the false positive times, and another set of values
 675 during the true positive times. Calling the values taken by
 676 the latent variable during true positive times *accept values*,
 677 and those taken during the false positive times *non-accept*
 678 *values*, the event

679 *clicked* && (*latentVar* in /* accept values */])

680 perfectly matches the observed update function times. This
 681 is because *clicked* is true during a set of false positive times,
 682 and *latentVar* is in *non-accept* values at exactly those times,
 683 so bullet addition does not take place, as desired. The full
 684 AUTUMN definition of *latentVar*, including the *transition*
 685 *on-clauses* that change its value over time, is shown in Figure
 686 4. The variable name *numCoins* is substituted to note the
 687 equivalence to a *number of collected coins* tracker.

688 The challenge in constructing this latent variable is learning
 689 the transition on-clauses that update the value of the
 690 variable at the appropriate times. Note that these transition
 691 on-clauses represent *edges* in the *automaton* diagrammed in
 692 Figure 5 (hence the use of the term *accept values* or *states*).
 693 We perform the transition learning step as part of a general
 694 automaton search procedure, implemented via a SAT solver
 695 as well as heuristically, to be discussed in Section 4.

696 3 Problem Formulation

697 Having provided a high-level description of the operation
 698 of our synthesis algorithm, we now formalize the full in-
 699 ductive synthesis problem for which our approach produces
 700 approximate solutions.

701 4 The Algorithm

702 We now give detailed descriptions of the steps of our algo-
 703 rithm introduced in Section 2. We focus on the latter two
 704 steps of the algorithm—update function synthesis and cause
 705 synthesis—referring the reader to the Appendix for full de-
 706 tails of the object perception and tracking steps (Step 1 and
 707 2), since they use more standard techniques and are not a
 708 central contribution of our work.

709 4.1 Step 3: Update Function Synthesis

710 Together, the object tracking (Step 2) and update function
 711 synthesis (Step 3) steps in the synthesis procedure answer the
 712 question, “What does each object *do* at each time step?” Ob-
 713 ject tracking first determines which objects in a grid frame
 714 become which objects in the next grid frame, as well as
 715 which objects were just added to or removed from the grid
 716 frame, across the full observation sequence. Then, the update
 717 function synthesis procedure computes an AUTUMN expres-
 718 sion, the update function, that describes every object-object
 719 mapping. This includes update functions describing object
 720 addition and removal, which are represented as mappings
 721 with a null or non-existent object: a null-object mapping
 722 indicates object addition and an object-null mapping indi-
 723 cates object removal. These update functions will eventually
 724 become part of the on-clauses in the final output program.

725 To identify a matching update function, the procedure
 726 simply enumerates through a fixed, finite space of update
 727 function expressions, such as *obj* = *moveLeft* *obj* or *obj*
 728 = *nextLiquid* *obj*. Some of these update function options
 729 are simple translations, like *moveLeft* *obj* and *move* *obj*
 730 -2 0, while others are more *abstract* options that describe
 731 multiple concrete translations under different circumstances.
 732 For example, the *nextLiquid* function causes an object to
 733 move down when there is no object below it (i.e. there is no
 734 chance of collision), and to the left or right if there is an object
 735 below but there exists a path to a lower height in the left or
 736 right direction. There are typically multiple update functions
 737 in the space that describe any given object assignment, so
 738 the procedure collects all of these possibilities.

739 At the end of this process, the synthesized update func-
 740 tions may be visualized in a matrix depiction, which we call
 741 the *update function matrix* (Figure 5). In the update function
 742 matrix, the rows represent *object_id*’s, where objects are
 743 assigned the same *object_id* if one is transformed into the other
 744 over time, and the columns represent times in the observa-
 745 tion sequence (in increasing order). Each cell in the update
 746 function matrix contains the set of possible update function
 747 expressions corresponding to that particular *object_id* at that
 748 particular time, or more precisely, those possibly undergone
 749 by the object *between* the frame at that time and the frame
 750 at the next time.

751 Ultimately, rather than a set of update functions for each
 752 *object_id* at each time, we want a single update function. This
 753 is because we will eventually search for AUTUMN predicates
 754 that evaluate to true at the times that each update function
 755 takes place, to form the on-clauses of the final synthesized
 756 program. Different choices for the single update function in
 757 each cell in the update function matrix changes the sets of
 758 times at which matching predicates must be true. For exam-
 759 ple, say that the sets of possible update functions undergone
 760 by an object in a three-grid-frame observation sequence are
 761 { *moveLeft* }, { *nextLiquid*, *moveLeft* }, and { *nextLiquid*,
 762 763 *moveRight* }. The update function matrix for this object would
 764 have three rows, one for each grid frame, and three columns,
 765 one for each time step. The first row would contain the set
 766 { *moveLeft* } in all three columns. The second row would con-
 767 tain the set { *nextLiquid*, *moveLeft* } in the first column, the set
 768 { *moveRight* } in the second column, and the set { } in the third
 769 column. The third row would contain the set { } in all three
 770 columns.

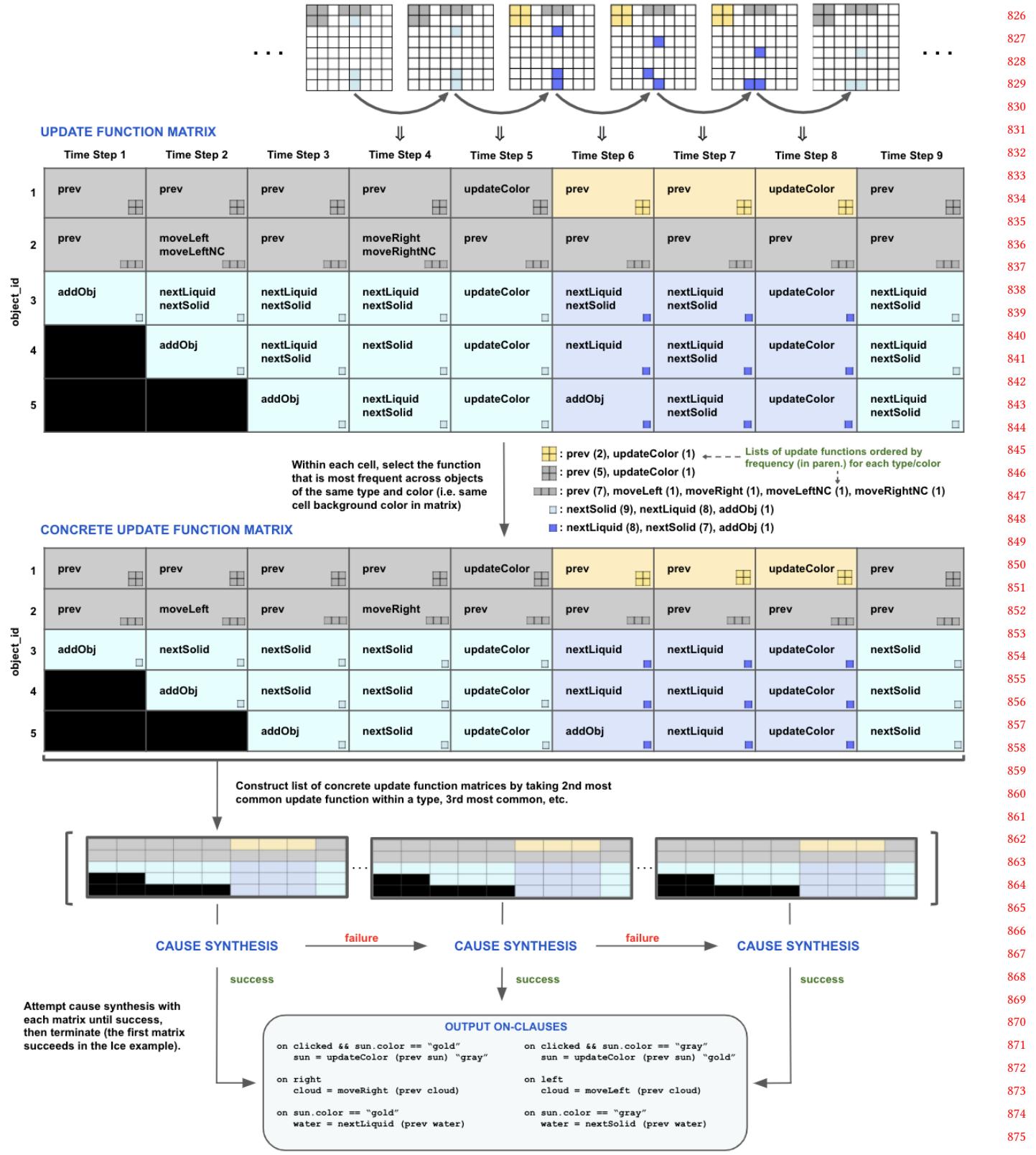


Figure 5. Update Function Synthesis. Each cell of the update function matrix contains a set of update functions that each describes the change undergone by the object with `object_id` equal to the row index during a particular time step (column index). A list of concrete update function matrices, with 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-state-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* (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

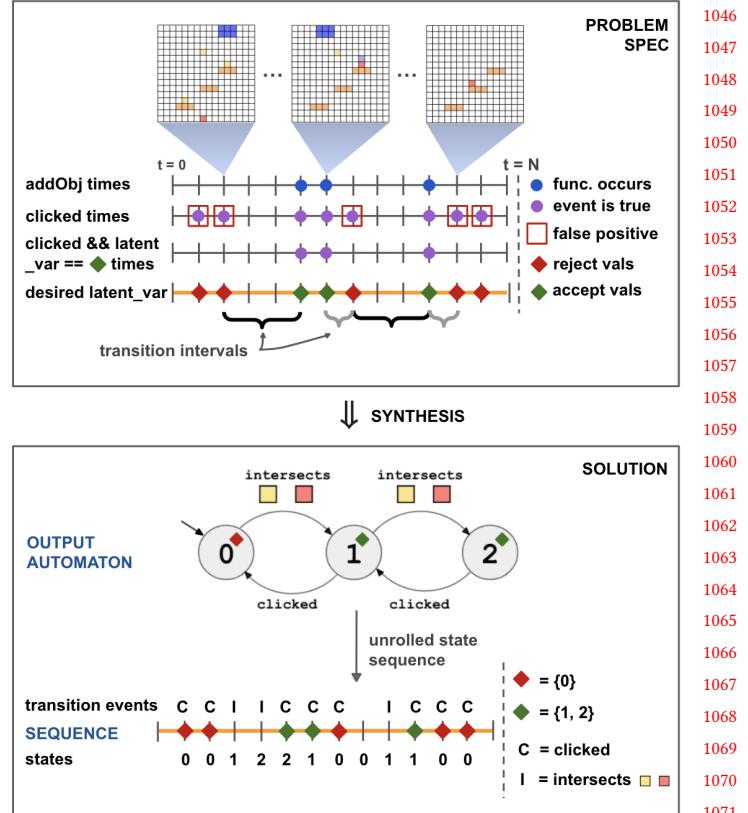


Figure 6. Bird’s-eye view of the automata synthesis problem, using the example of the Mario program. The bullet addition update function, indicated by *addObj*, does not have a matching trigger event. The closest event is *clicked*, which co-occurs with bullet addition but also is true at false positive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

1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100

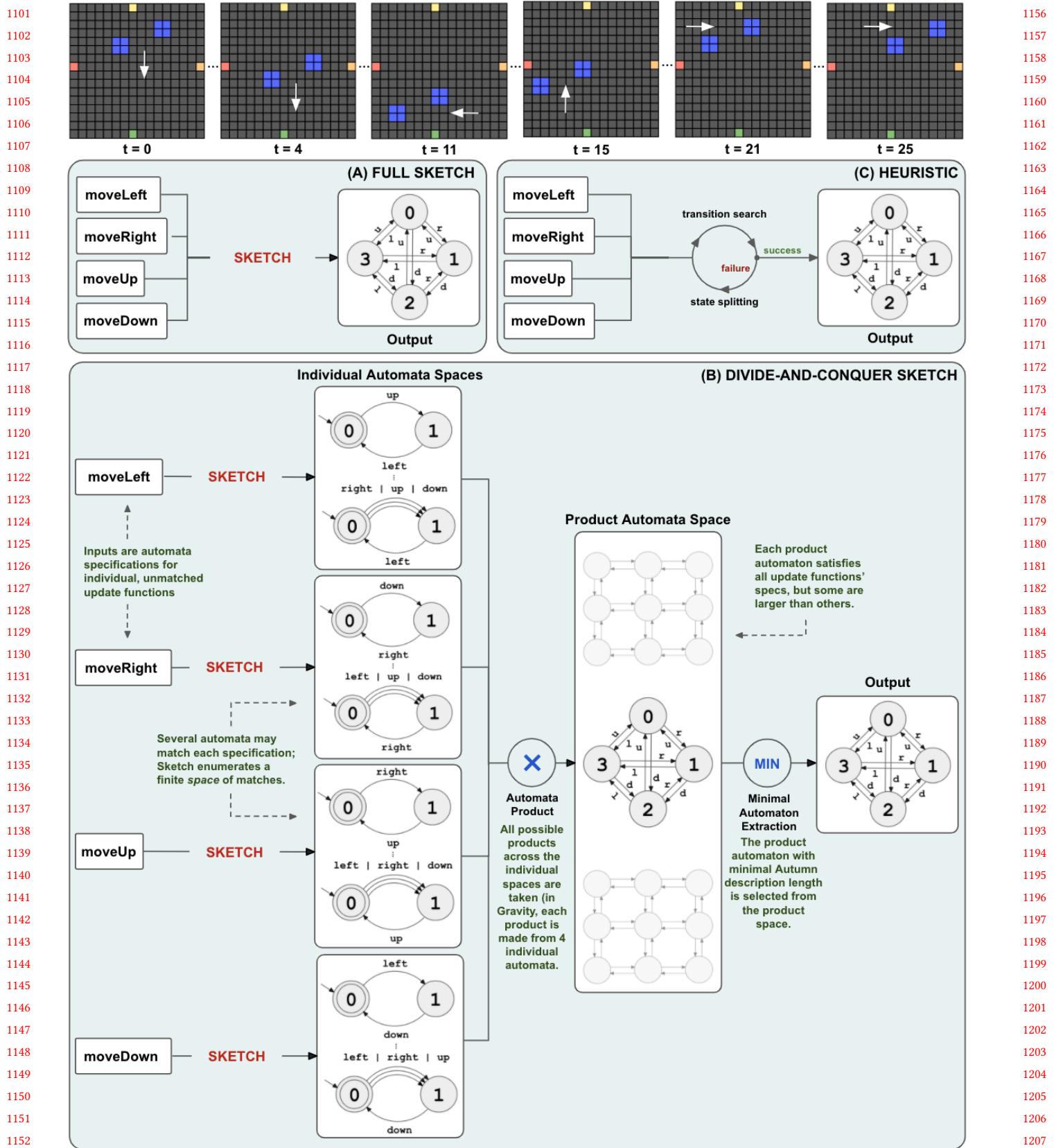


Figure 7. Three variant methods for automata synthesis, shown for Gravity I. The blue blocks move left, right, up, or down depending on the button last clicked. The transition label `left` abbreviates `(clicked leftButton)`, etc. See note in Sec. 4.3.2.

1211 those times (since those times correspond to the rejected
 1212 program state prefixes). Thus, finding such an automaton
 1213 would mean we would have an event that matches the update
 1214 function under consideration.

1215 Having discussed this simpler setting in which there is just
 1216 one unmatched update function in need of latent state, we
 1217 now return to the full problem setting, in which there may
 1218 be multiple unmatched update functions. In this scenario,
 1219 each unmatched update function specifies its own inductive
 1220 automata synthesis problem—a set of positive and negative
 1221 input strings—that if solved will give rise to a matching latent
 1222 state-based predicate. One solution to this “multi-automata”
 1223 synthesis problem is to construct a distinct latent automaton
 1224 (variable) that satisfies each update function. However,
 1225 a smaller number of latent variables is often sufficient to
 1226 explain all the update functions. In fact, the *product* of all the
 1227 individual update function automata is a single automaton
 1228 that satisfies all specifications, up to changing the accept
 1229 states for each update function. However, taking the product
 1230 of the smallest automata satisfying individual update
 1231 functions does not necessarily produce the smallest product
 1232 automaton: It is possible that larger component automata
 1233 will multiply to form this minimal product instead. Thus,
 1234 optimizing each individual update function’s automaton and
 1235 multiplying is not a sufficient solution.

1236 We now discuss three distinct algorithms for solving this
 1237 inductive automata synthesis problem: Full Sketch, Divide-
 1238 and-Conquer Sketch, and Heuristic. We note that at the current
 1239 stage of this ongoing work, we synthesize a single latent
 1240 state automaton that satisfies all unmatched update functions
 1241 within *each object type*, as opposed to a single automaton for
 1242 the entire program (i.e. across all object types). The reason we
 1243 do not try to find one program-level automaton is because the
 1244 human-written AUTUMN programs in our benchmark suite
 1245 use a different latent variable for each type—a choice that
 1246 appears to make the programs more human-understandable
 1247 than having one large product—and these sets of type-level
 1248 latent automata are also often more concisely expressed in
 1249 the AUTUMN language than a single product. We will formalize
 1250 this approach with respect to the overall synthesis
 1251 objective of identifying the minimal AUTUMN program satis-
 1252 fying the observations in the final version of this work.

1253
 1254 **4.3.1 Algorithm 1: Full Sketch.** In the Full Sketch ap-
 1255 proach, the complete multi-automata synthesis problem (for
 1256 each object type) is encoded as a Sketch problem. In other
 1257 words, Sketch is tasked with identifying the minimal au-
 1258 tomaton that accepts each update function’s language, as
 1259 specified by the observed examples, up to changing just the
 1260 accept states. As an example, consider the AUTUMN program
 1261 named Gravity I shown in Figure 6. The blue blocks contin-
 1262 uously move left, right, up, or down depending on which
 1263 of the four colored buttons was last pressed. A matching
 1264 event cannot be found for any of the four update functions

1265 moveLeft, moveRight, moveUp, or moveDown, so their update
 1266 function trajectories are fed to the Sketch solver to produce
 1267 the 4-state automaton shown in Figure 6a. This new latent
 1268 variable then allows a matching predicate to be written for
 1269 each of the four update functions: true && latentVar == 1,
 1270 true && latentVar == 2, true && latentVar == 3, and
 1271 true && latentVar == 4, where the optimal co-occurring
 1272 event is true.

1273
 1274 **4.3.2 Algorithm 2: Divide-And-Conquer Sketch.** Rather
 1275 than attacking the full multi-automata synthesis problem,
 1276 Divide-And-Conquer Sketch tasks Sketch with solving each
 1277 update function’s automata synthesis problem *individually*,
 1278 and then combines those solutions together via product. The
 1279 intuition behind this approach is that synthesizing an au-
 1280 tomaton matching *all* update functions at once may face
 1281 scalability challenges, but finding an automaton matching a
 1282 single update function, which is likely smaller, may be easier.
 1283 As described previously, the smallest automaton satisfying a
 1284 single update function may not give rise to the smallest
 1285 product, so the Divide-and-Conquer algorithm identifies a
 1286 small *set* of automata matching each update function instead.
 1287 It then takes the product over all update functions’ automata
 1288 sets, and computes the minimal automaton from that product
 1289 space. We illustrate this algorithm again with the Gravity I
 1290 example (Figure 6b). The algorithm first identifies a set of
 1291 automata that solve the automata synthesis problems corre-
 1292 sponding to the four unmatched update functions. Note that
 1293 each of these automata have just two states instead of the
 1294 full 4-state solution found in the Full SAT approach. Next,
 1295 it computes all automata *products* over these four automata
 1296 sets, and takes the minimal automaton from this product set,
 1297 which is the 4-state solution seen previously.

1298
 1299 (A note about Figure 6b: For reasons of tractability, we
 1300 employ a simple heuristic to downsize each individual update
 1301 function’s automata set before taking the product across all
 1302 automata sets. At a high level, this heuristic identifies subsets
 1303 of the full automata set that are *observationally equivalent*
 1304 with respect to the given input observation sequence, and
 1305 keeps just one automaton from each of these equivalence
 1306 classes. This step is not shown in the figure. We will give a
 1307 more detailed explanation of this procedure and definition of
 1308 observational equivalence in the final version of this paper.)

1309
 1310 **4.3.3 Algorithm 3: Heuristic.** Despite the simplicity of
 1311 the Sketch-based formulations of automata synthesis, their
 1312 scalability to problem settings with large automata is un-
 1313 clear, due to known limitations of SAT solvers. As such, we
 1314 also implemented a heuristic algorithm that synthesizes an
 1315 automaton satisfying a set of update function trajectories via
 1316 a series of greedy updates to an initial automaton (Figure 6c).
 1317 At a high level, this approach begins with an automaton with
 1318 a small number of states, and repeatedly *splits* states into
 1319 two based on a heuristic related to the search for transition
 1320 events. More precisely, the algorithm begins by searching for

1321 transition events (edges) that result in an automaton that
 1322 produces a particular initial state sequence that has few distinct
 1323 states. If transition search fails, one of the original states is
 1324 split into two, and transition search is repeated. This process
 1325 continues until a satisfying automaton is identified.
 1326

5 Preliminary Evaluation

5.1 The AUTUMN Benchmark Dataset

To evaluate our algorithm, we manually constructed a set
 1330 of 31 AUTUMN programs, designed to collectively embody
 1331 a rich variety of 2D causal mechanisms. These benchmark
 1332 programs are described in Table 1 (Figure 8). Seven of the
 1333 models do not contain latent state, and hence test only the
 1334 functional component of our synthesis procedure, while the
 1335 remaining 24 models contain latent state, thus also testing
 1336 the automata synthesis component.
 1337

As our evaluation remains ongoing, for our preliminary re-
 1338 sults, we manually constructed an input user action sequence
 1339 for each benchmark program, and ran the three synthesis
 1340 algorithms—Full SAT, Divide-and-Conquer SAT, and Heuris-
 1341 tic—on these sequences. We declared a success for a synthesis
 1342 algorithm if it produced an output program that matches the
 1343 observation sequence, though it need not be perfectly equiv-
 1344 alent to the ground-truth program. Both of these aspects
 1345 will be updated in our final evaluation, in which we plan to
 1346 measure the success of our synthesis algorithms on input
 1347 sequences generated by several human subjects interacting
 1348 with the models, and define success to be the output program
 1349 being semantically equivalent to the ground-truth program.
 1350

The results of this evaluation are shown in Table 2 (Figure
 1351 9) and Figures 10 and 11. While these results are subject to
 1352 change as we continue to finalize our work, it appears that
 1353 the Heuristic algorithm is currently most effective: It solves
 1354 all but four of the benchmarks, and does so in less time than
 1355 either of the other two algorithms, though the runtime is
 1356 very similar to Full Sketch’s runtime on many models. The
 1357 Divide-and-Conquer Sketch algorithm is notably slower than
 1358 both the Heuristic and Full Sketch algorithms on almost all
 1359 of the models that all three methods solve. Further, while the
 1360 vast majority of the programs synthesized by the Heuristic
 1361 and Full Sketch algorithms either exactly or almost exactly
 1362 match the ground-truth programs, many of the programs
 1363 synthesized by the Divide-and-Conquer method do not gen-
 1364 eralize as accurately. This is a result of the fact that we do
 1365 not enumerate the entire space of automata matching each
 1366 individual update function before taking the product. We
 1367 instead just enumerate a small, finite subset, so the computed
 1368 product is often not optimal.
 1369

The most interesting two results in our evaluation are the
 1370 following: (1) For four of the benchmark programs—Gravity
 1371 III, Count III, Count IV, and Double II—both Sketch-based
 1372 algorithms timed out after 24 hours without producing a so-
 1373 lution, while the Heuristic algorithm solved all those models
 1374

in minutes to hours: 2.3, 6.9, 118.3, and 17.3 minutes, respec-
 1375 tively. The poor performance of the Full Sketch method on
 1376 these models is due to the fact that the models’ latent state
 1377 automata are large (e.g. nine states and 24 edges for Gravity
 1378 III), so the underlying SAT solver does not terminate. Divide-
 1379 and-Conquer Sketch fails for the same reason, because while
 1380 individual-update-function-level automata are often smaller
 1381 than the overall automaton, in these models, each individ-
 1382 ual automaton is actually the same as the full automaton.
 1383 Hence, Sketch again does not terminate in the Divide-and-
 1384 Conquer framing. (2) For one benchmark program, Swap,
 1385 the Full Sketch approach timed out after 24 hours, but the
 1386 Divide-and-Conquer Sketch algorithm actually managed to
 1387 find a solution in 21.4 minutes. (The Heuristic algorithm also
 1388 solves this model, in 2.3 minutes.) The reason for this unusual
 1389 result is that the Swap model has a latent state automaton
 1390 with eight states and 64 edges, too large for the Full Sketch
 1391 algorithm to handle, but which is the product of eight two-
 1392 or three-state automata corresponding to the eight distinct
 1393 update functions in the program. Sketch can more easily
 1394 identify a two- or three-state automaton satisfying a speci-
 1395 fication, so the Divide-and-Conquer Sketch algorithm does
 1396 this eight times and hence terminates successfully.
 1397

We also comment on the benchmark programs that none
 1398 of our algorithms were able to synthesize. For these models,
 1399 many of the fixes are lower-level modifications to the overall
 1400 algorithm. For example, for the Grow II and Egg programs,
 1401 an event predicate needed to express the program is actu-
 1402 ally just missing from the atomic event space we use for
 1403 search, so it should be added to the space. Another limita-
 1404 tion is that sometimes the optimal co-occurring event com-
 1405 puted for a particular latent-state-based update function is
 1406 incorrect, causing synthesis to fail. However, the second-best
 1407 co-occurring event—that with the second smallest number
 1408 of false positives rather than the smallest—may be correct,
 1409 or the third-best, etc. This general kind of failure can be
 1410 reduced by implementing a form of “multiplicity handling”
 1411 with respect to co-occurring events, where instead of try-
 1412 ing only the best event and terminating if it causes the rest
 1413 of synthesis to fail, we try the top-k best events until one
 1414 hopefully succeeds. These kinds of updates to our current
 1415 algorithm are ongoing.

Finally, we emphasize that our benchmark results are still
 1416 preliminary and are subject to change as we continue to
 1417 modify both the Heuristic and the Sketch-based algorithms,
 1418 including with the generalizations described above. Some
 1419 of these modifications will affect all three algorithms’ run-
 1420 times, like the previously described “multiplicity handling”
 1421 generalization, while others will affect individual algorithms’
 1422 runtimes. For example, optimizations to the Sketch imple-
 1423 ments could decrease the Sketch-based algorithms’ run-
 1424 times, while improvements that make the Heuristic algo-
 1425 rithm less brittle/more general would increase the Heuristic
 1426 algorithm’s runtimes (while the Heuristic algorithm works
 1427 1428 1429 1430

1431 well on the current benchmark suite, the nature of it being a
 1432 heuristic means that there are certainly classes of models on
 1433 which it will fail, which we can patch somewhat with more
 1434 intricate algorithms). Further thinking about our evaluation
 1435 design, including potentially running the Sketch solver with
 1436 a few different parameter options to fend against blowup,
 1437 to ensure the fairest possible comparison between the three
 1438 algorithms remains part of future work. These modifications
 1439 may result in different relative runtimes across the variant
 1440 algorithms than we currently observe (e.g. potentially lower
 1441 Sketch runtimes and higher Heuristic runtimes on some
 1442 benchmarks). In our final evaluation, we will also average
 1443 the runtimes over more trials; our current results are aver-
 1444 aged over 2-4 runs, where the smaller benchmarks were run
 1445 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/tbtl.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.

1446

1447

1448

1449

1450

1451

1452

1453

1454

1455

1456

1457

1458

1459

1460

1461

1462

1463

1464

1465

1466

1467

1468

1469

1470

1471

1472

1473

1474

1475

1476

1477

1478

1479

1480

1481

1482

1483

1484

1485

1486

1487

1488

1489

1490

1491

1492

1493

1494

1495

1496

1497

1498

1499

1500

1501

1502

1503

1504

1505

1506

1507

1508

1509

1510

1511

1512

1513

1514

1515

1516

1517

1518

1519

1520

1521

1522

1523

1524

1525

1526

1527

1528

1529

1530

1531

1532

1533

1534

1535

1536

1537

1538

1539

1540

| | | | | | | | |
|------|-----------------|--|----|---|------|--|------|
| 1541 | | | | | 1596 | | |
| 1542 | | | | | 1597 | | |
| 1543 | | | | | 1598 | | |
| 1544 | | | | | 1599 | | |
| 1545 | | | | | 1600 | | |
| 1546 | | | | | 1601 | | |
| 1547 | | | | | 1602 | | |
| 1548 | | | | | 1603 | | |
| 1549 | | | | | 1604 | | |
| 1550 | | | | | 1605 | | |
| 1551 | | | | | 1606 | | |
| 1552 | | | | | 1607 | | |
| 1553 | | | | | 1608 | | |
| 1554 | | | | | 1609 | | |
| 1555 | | | | | 1610 | | |
| 1556 | No Latent State | Particles | 2 | 0 | 0 | Brownian motion of single-cell objects. | 1611 |
| 1557 | | Ants | 3 | 0 | 0 | Ants foraging for randomly generated food particles. | 1612 |
| 1558 | | Chase | 7 | 0 | 0 | Agent evading randomly generated enemies. | 1613 |
| 1559 | | Magnets | 13 | 0 | 0 | Two magnets displaying attraction/repulsion. | 1614 |
| 1560 | | Space Invaders | 12 | 0 | 0 | A clone of Atari Space Invaders. | 1615 |
| 1561 | | Sokoban | 7 | 0 | 0 | A clone of Sokoban. | 1616 |
| 1562 | | Ice | 10 | 0 | 0 | Water particles behaving like solids vs. liquids. | 1617 |
| 1563 | Latent State | Lights | 4 | 2 | 2 | Clicking turns on/off a set of lights. | 1618 |
| 1564 | | Disease | 7 | 2 | 2 | Sick particles infect healthy particles. | 1619 |
| 1565 | | Grow I | 11 | 2 | 2 | Flowers grow upon water addition and sunlight. | 1620 |
| 1566 | | Grow II | 11 | 2 | 2 | Same as above, but plant stems grow longer. | 1621 |
| 1567 | | Sandcastle I | 7 | 2 | 2 | Water causes sand particles to turn liquid from solid. | 1622 |
| 1568 | | Sandcastle II | 7 | 2 | 2 | Same as above, but buttons match water/sand colors. | 1623 |
| 1569 | | Egg | 7 | 2 | 2 | An egg breaks upon being dropped from high enough. | 1624 |
| 1570 | | Bullets | 17 | 8 | 12 | Agent that can shoot bullets in four directions. | 1625 |
| 1571 | | Gravity I | 9 | 4 | 12 | Blocks move according to four gravity directions. | 1626 |
| 1572 | | Gravity II | 14 | 7 | 15 | Same as above, except colors of added blocks rotate. | 1627 |
| 1573 | | Gravity III | 32 | 9 | 24 | Blocks move according to nine gravity directions. | 1628 |
| 1574 | | Gravity IV | 17 | 8 | 56 | Same as Gravity I, except there are eight gravities. | 1629 |
| 1575 | | Count I | 6 | 3 | 4 | Weighted left/right movement, with two weights. | 1630 |
| 1576 | | Count II | 10 | 5 | 8 | Weighted left/right movement, with four weights. | 1631 |
| 1577 | | Count III | 14 | 7 | 12 | Weighted left/right movement, with six weights. | 1632 |
| 1578 | | Count IV | 18 | 9 | 16 | Weighted left/right movement, with eight weights. | 1633 |
| 1579 | | Double Count I | 12 | 5 | 8 | Weighted left/right/up/down, with four weights. | 1634 |
| 1580 | | Double Count II | 20 | 9 | 16 | Weighted left/right/up/down, with eight weights. | 1635 |
| 1581 | | Wind | 9 | 3 | 4 | Snow falls left, down, or right based on wind state. | 1636 |
| 1582 | | Paint | 10 | 5 | 5 | A simplified clone of MSFT Paint, with five colors. | 1637 |
| 1583 | | Mario | 19 | 5 | 6 | A Mario-style agent collects coins and shoots enemy. | 1638 |
| 1584 | | Mario II | 19 | 7 | 7 | Same as above, but enemy has two lives, not just one. | 1639 |
| 1585 | | Swap | 40 | 8 | 64 | Same as Gravity IV, but clicks also toggle two states. | 1640 |
| 1586 | | Water Plug | 8 | 3 | 6 | Water interacts with a sink and removable sink plug. | 1641 |
| 1587 | | | | | | | 1642 |
| 1588 | | | | | | | 1643 |
| 1589 | | | | | | | 1644 |
| 1590 | | | | | | | 1645 |
| 1591 | | Figure 8. Descriptions of the 31 benchmark programs. | | | | | 1646 |
| 1592 | | | | | | | 1647 |
| 1593 | | | | | | | 1648 |
| 1594 | | | | | | | 1649 |
| 1595 | | | | | | | 1650 |

| | | | | | | | | |
|------|---|-----------------|-----------------------|-------------------------------|-------------------|----------------|--------------------|------|
| 1651 | | | | | | 1706 | | |
| 1652 | | | | | | 1707 | | |
| 1653 | | | | | | 1708 | | |
| 1654 | | | | | | 1709 | | |
| 1655 | | | | | | 1710 | | |
| 1656 | | | | | | 1711 | | |
| 1657 | | | | | | 1712 | | |
| 1658 | | | | | | 1713 | | |
| 1659 | No Latent State | Model Name | Input Length (Frames) | Output Length (Program Lines) | Heuristic Runtime | Sketch Runtime | D&C Sketch Runtime | 1714 |
| 1660 | | Particles | 22 | 21 | 12.0 | N/A | N/A | 1715 |
| 1661 | | Ants | 24 | 24 | 221.3 | N/A | N/A | 1716 |
| 1662 | | Chase | 42 | 33 | 51.9 | N/A | N/A | 1717 |
| 1663 | | Magnets | 53 | 43 | 60.8 | N/A | N/A | 1718 |
| 1664 | | Space Invaders | 42 | 52 | 147.2 | N/A | N/A | 1719 |
| 1665 | | Sokoban | 25 | 38 | 56.0 | N/A | N/A | 1720 |
| 1666 | | Ice | 27 | 45 | 3.6 | N/A | N/A | 1721 |
| 1667 | Latent State | Lights | 24 | 36 | 5.0 | 5.6 | 8.3 | 1722 |
| 1668 | | Disease | 22 | 31 | 5.3 | 5.8 | 6.2 | 1723 |
| 1669 | | Grow | 40 | 49 | 172.3 | 217.8 | 276.5 | 1724 |
| 1670 | | Grow II | 159 | 49 | ✗ | ✗ | ✗ | 1725 |
| 1671 | | Egg | 31 | 43 | ✗ | ✗ | ✗ | 1726 |
| 1672 | | Sandcastle I | 32 | 33 | 12.7 | 13.7 | 15.6 | 1727 |
| 1673 | | Sandcastle II | 32 | 33 | ✗ | ✗ | ✗ | 1728 |
| 1674 | | Bullets | 54 | 54 | 32.4 | 45.3 | ⊥ | 1729 |
| 1675 | | Gravity I | 19 | 37 | 3.9 | 4.2 | 4.5 | 1730 |
| 1676 | | Gravity II | 24 | 50 | 10.1 | 10.5 | 14.9 | 1731 |
| 1677 | | Gravity III | 27 | 83 | 2.3 | ⊥ | ⊥ | 1732 |
| 1678 | | Gravity IV | 48 | 54 | 6.2 | 7.3 | 10.9 | 1733 |
| 1679 | | Count I | 22 | 31 | 2.5 | 2.8 | 3.6 | 1734 |
| 1680 | | Count II | 39 | 39 | 2.5 | 10.2 | 9.9 | 1735 |
| 1681 | | Count III | 69 | 47 | 6.9 | ⊥ | ⊥ | 1736 |
| 1682 | | Count IV | 109 | 55 | 118.3 | ⊥ | ⊥ | 1737 |
| 1683 | | Double Count I | 94 | 43 | 3.7 | 8.11 | 29.3 | 1738 |
| 1684 | | Double Count II | 156 | 59 | 17.3 | ⊥ | ⊥ | 1739 |
| 1685 | | Wind | 21 | 42 | 76.6 | 79.0 | 79.3 | 1740 |
| 1686 | | Paint | 27 | 39 | 9.5 | 9.8 | 21.7 | 1741 |
| 1687 | | Mario | 81 | 65 | 181.9 | 220.3 | 235.9 | 1742 |
| 1688 | | Mario II | 208 | 65 | ✗ | ✗ | ✗ | 1743 |
| 1689 | | Swap | 44 | 99 | 2.3 | ⊥ | 21.4 | 1744 |
| 1690 | | Water Plug | 42 | 37 | ⊥ | ⊥ | ⊥ | 1745 |
| 1691 | | | | | | | | 1746 |
| 1692 | | | | | | | | 1747 |
| 1693 | | | | | | | | 1748 |
| 1694 | 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. | | | | | | | 1749 |
| 1695 | | | | | | | | 1750 |
| 1696 | | | | | | | | 1751 |
| 1697 | | | | | | | | 1752 |
| 1698 | | | | | | | | 1753 |
| 1699 | | | | | | | | 1754 |
| 1700 | | | | | | | | 1755 |
| 1701 | | | | | | | | 1756 |
| 1702 | | | | | | | | 1757 |
| 1703 | | | | | | | | 1758 |
| 1704 | | | | | | | | 1759 |
| 1705 | | | | | | | | 1760 |

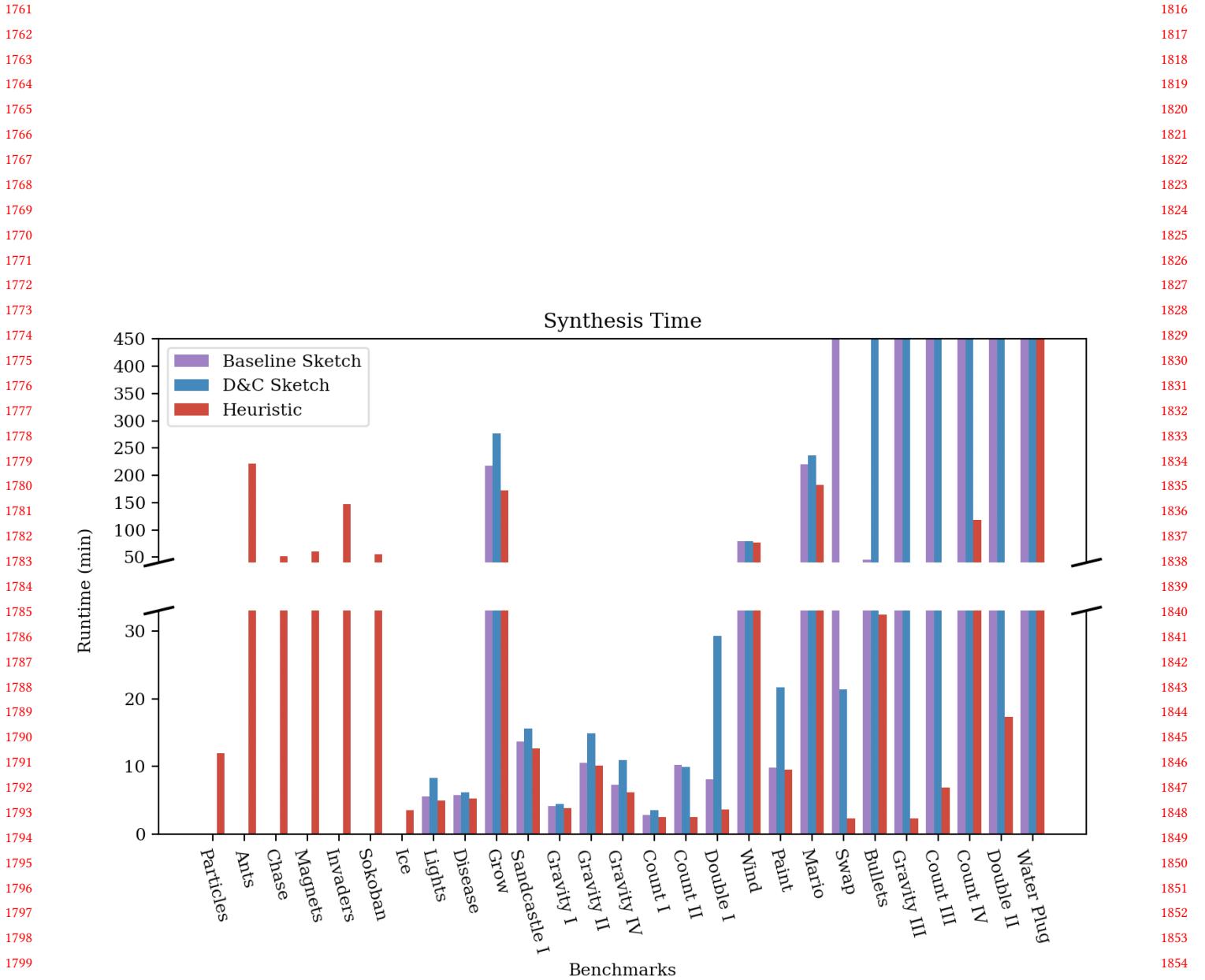


Figure 10. Runtimes for the variant AUTUMNSYNTH algorithms on each of the benchmark programs solved by at least one algorithm. Note that the first 7 benchmarks (left of the dashed line; Particles, Ants, Chase, Magnets, Invaders, Sokoban, and Ice) all do not contain latent state, so we currently evaluate 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.

1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870

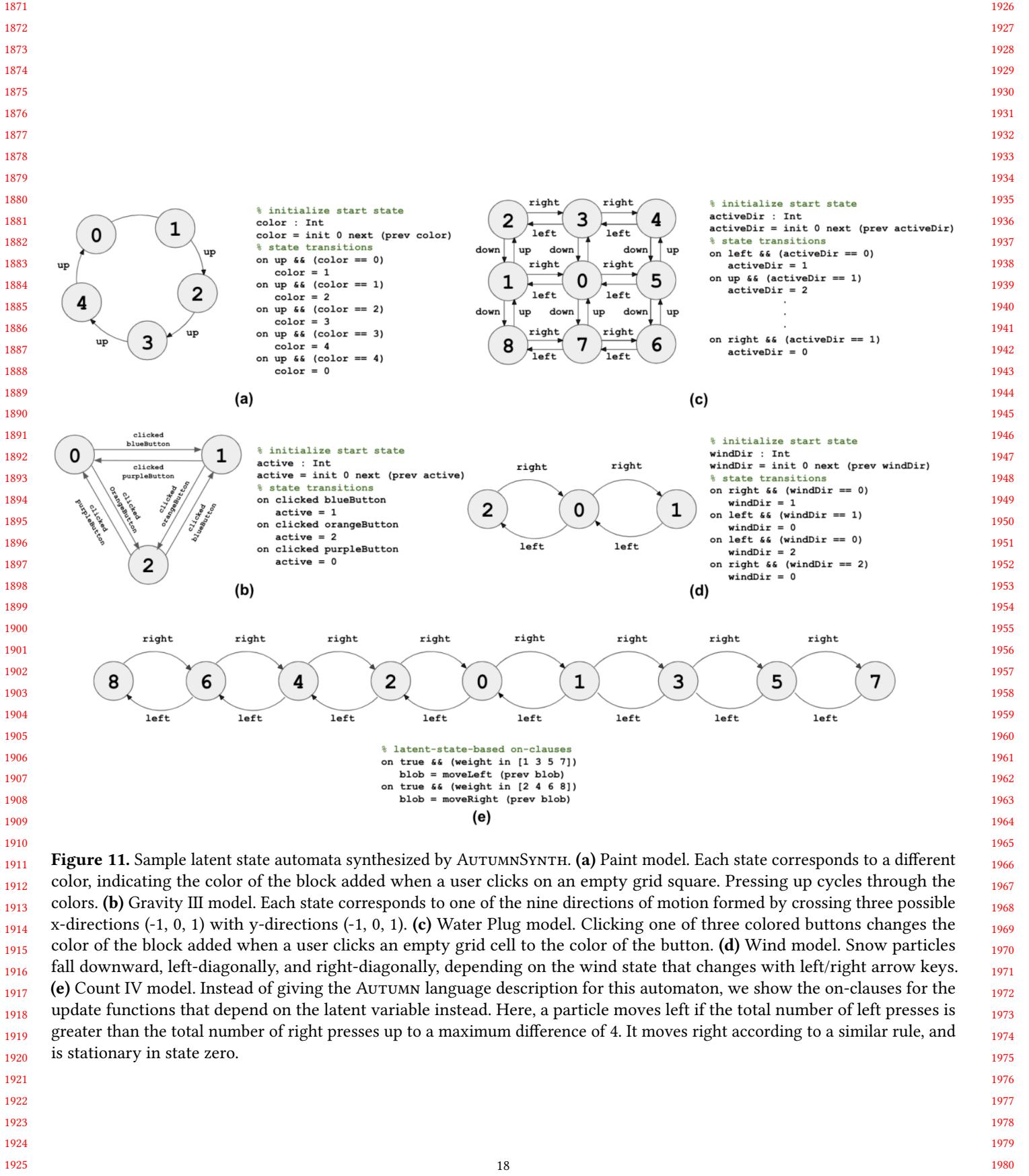


Figure 11. Sample latent state automata synthesized by AUTUMNSYNTH. **(a)** Paint model. Each state corresponds to a different color, indicating the color of the block added when a user clicks on an empty grid square. Pressing up cycles through the colors. **(b)** Gravity III model. Each state corresponds 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.

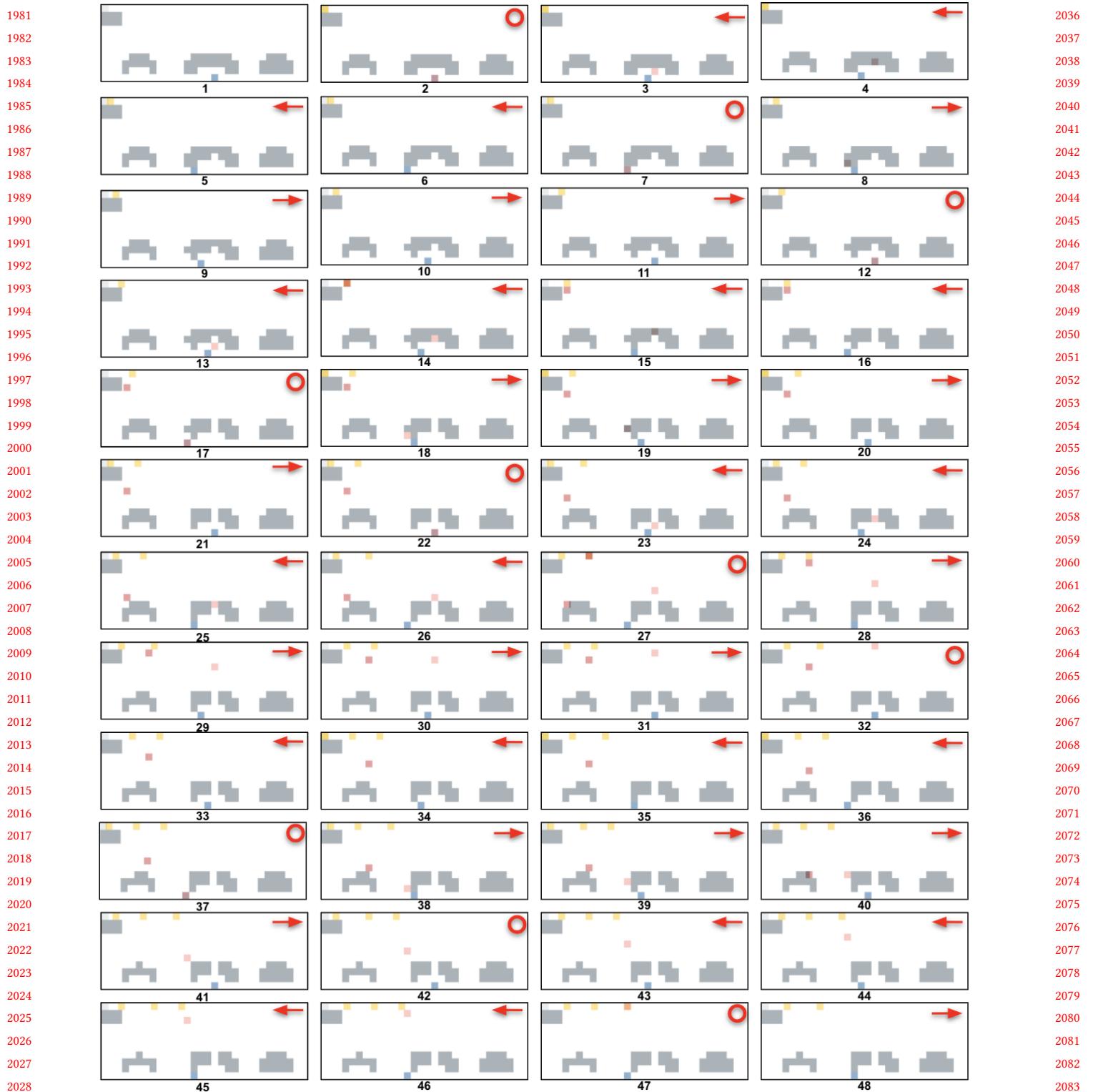


Figure 12. The Aliens program from the Tsividis et. al. corpus. Pressing arrow keys moves the blue agent left and right, and clicking causes it to shoot a pink bullet upward, as long as there are no other pink bullets already in the frame. Gold enemies are regularly created at the top-left corner, and 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.

2035