Combining Functional and Automata Synthesis to Discover Causal Reactive Programs

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

Abstract

While program synthesis has recently garnered interest as an alternative to deep-learning-based approaches in AI, it still faces several limitations. One is that existing methods cannot learn models with *time-varying latent state*, a common feature of real-world systems. We develop a new synthesis approach that overcomes this challenge by uniting two disparate communities within synthesis: *functional synthesis* and *automata synthesis*. We instantiate our algorithm in the domain of causal learning in 2D, Atari-style grid worlds, and our preliminary evaluation shows promising results.

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 language phonology and synthesizing computer-aided design (CAD) programs from 3D mesh models (Ellis et al., 2019. In prep.; Nandi et al., 2020).

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. Despite the promise of formulating theory induction as program synthesis, however, 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 (Ellis et al., 2019). 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 (Ellis et al., 2018). 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 in unsupervised learning, but also *how* that latent state *evolves* over time must be identified, in the form of programmatic rules.

For concreteness, we introduce the simple yet rich domain of Atari-style, time-varying 2D grid worlds (Figure 1) that we consider in this paper, which demonstrates these shortcomings of inductive program synthesis. This domain is of considerable 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 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

[*]Equal contribution [1]MIT [2]Basis and Columbia University. Correspondence to: Ria A. Das <riadas@mit.edu>.

Proceedings of the 39th International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022. Copyright 2022 by the author(s).

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 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 tries 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. 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 problems that can be solved by either method alone. 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 success at synthesizing finite-state automata or *transition systems* from traces, but their methods do not scale to domains with very large numbers of states (which are often more compactly represented using program abstractions) (Vaandrager, 2017).

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*, 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 (Solar-Lezama, 2008) 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 30 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. We further test on an externally-sourced benchmark of 27 grid world video games. Though subject to change as the work progresses, in our preliminary results, we find that our heuristic algorithm solves most of the benchmarks in both our own suite and the externally-sourced one, outperforming the Sketch methods and also synthesizing some large latent state automata along the way, a signal of the promise of our formulation.

2. Overview

In this section, we give a high-level overview of the AUTUMN language and AUTUMNSYNTH algorithm.

2.1. The AUTUMN Language

The AUTUMN language was designed to concisely express a rich variety of causal mechanisms in interactive 2D grid worlds. The language is *functional reactive*, indicating that it augments the standard functional language definition with primitive support for temporal events. The key elements of an AUTUMN program are (1) object type definitions, (2) object instance and latent variable definitions, and (3) *on-clauses*. The most interesting component to synthesize are these on-clauses, which describe the causal dynamics of the model via a sequence of statements with the syntax

```
on event
   update_function
```

where `event` is a predicate and `update_function` is a modification to an object or latent variable with the form `var = expr`, which takes place when `event` is true.

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. ***event and automata 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 provide some intuition by briefly describing how these steps are used to synthesize the Mario program.

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 include (1) a general single-cell type with a string color parameter corresponding to the (red) agent, (yellow) coin, and (gray) bullet objects and (2) a platform type that is a row of three orange cells. A list of object instances is extracted from each grid frame in the input sequence. For example, one object instance in the first grid frame is a red single-celled object (agent) at position (7, 15).

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

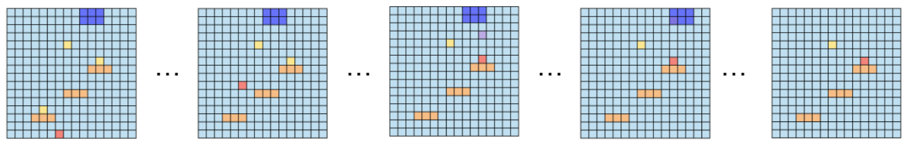
In the third step of update function synthesis, for each mapping between an object in one grid frame and an object in the next that is determined in Step 2, an AUTUMN expression is sought that describes that object-object mapping. For example, this step identifies that the expression `agent = moveLeft (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 = moveClosest (prev agent) Platform`, which 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.

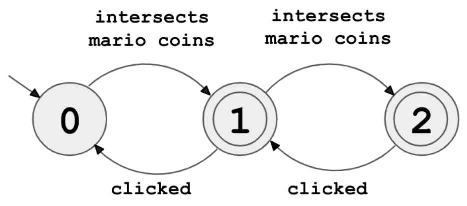
2.2.4. EVENT AND AUTOMATA SYNTHESIS

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. Initially,





```
% 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
```

Figure 1. Top row: Stills from the Mario model. The red agent initially cannot shoot bullets (purple), but after jumping up and collecting coins (yellow), shoots on a user click. The agent can no longer shoot when all its bullets are used up. A bullet kills the blue enemy. Bottom-Left: 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. Bottom-Right: AUTUMN description of the `numCoins` latent variable, as synthesized by AUTUMNSYNTH (variable renamed for simplicity).

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 update function is not always necessary.

The interesting case in the cause synthesis step is when a matching AUTUMN event cannot be found for an 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, and occurs during the fewest number of *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 && (latentV in [ accept vals ])
```

perfectly matches the observed update function times. This is because `clicked` is true during a set of false positive times, and `latentV` is in *non-accept* values at exactly those times, so bullet addition does not take place, as desired. The full AUTUMN definition of `latentV`, including the *transition on-clauses* that change its value over time, is shown in Figure 1. 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 in Figure 1 (hence the term *accept values*). We perform the transition learning step as part of a general automaton search procedure, implemented via a SAT solver as well as heuristically.

3. Experiments

For each model in our own benchmark suite and the external suite, we manually constructed an input sequence of user actions. In our preliminary results, AUTUMNSYNTH synthesized 27 out of 30 programs in our suite and 21 out of 27 programs in the external suite (Tsividis et al., 2021); see Appendix for details.

References

Ellis, K., Ritchie, D., Solar-Lezama, A., and Tenenbaum, J. Learning to infer graphics programs from hand-drawn images. In *NeurIPS*, 2018.

Ellis, K., Nye, M. I., Pu, Y., Sosa, F., Tenenbaum, J., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a REPL. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 9165–9174, 2019.

Ellis, K., Albright, A., Solar-Lezama, A., Tenenbaum, J. B., and O'Donnell, T. J. Synthesizing theories of human language with bayesian program induction. 2019. In prep.

Nandi, C., Willsey, M., Anderson, A., Wilcox, J. R., Darulova, E., Grossman, D., and Tatlock, Z. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pp. 31–44, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386012. URL https://doi.org/10.1145/3385412.3386012.

Solar-Lezama, A. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008. URL http://people.csail.mit.edu/asolar/papers/thesis.pdf.

Tsividis, P. A., Loula, J., Burga, J., Foss, N., Campero, A., Pouncy, T., Gershman, S. J., and Tenenbaum, J. B. Human-level reinforcement learning through theory-based modeling, exploration, and planning, 2021. URL https://arxiv.org/abs/2107.12544.

Vaandrager, F. Model learning. *Commun. ACM*, 60 (2):86–95, January 2017. ISSN 0001-0782. doi: 10.1145/2967606. URL https://doi.org/10.1145/2967606.

A. Further AUTUMN Language Details

Every AUTUMN program is composed of four parts (Figure 2). The first part defines the grid dimensions and background color. The second part defines *object types*, which are simply structs which define an object *shape*, or a list of 2D positions each associated with a color, as well as a set of *internal fields*, which store additional information about the object (e.g. a Boolean `healthy` field may store an indicator of the object's health). The third part defines *object instances*, which are concrete instantiations of the object types defined previously, as well as *latent variables*, which are values with type `int`, `string`, or `bool`. Object instances and latent variables are defined using a primitive AUTUMN language construct called `initnext`, which defines a *stream* of values over time via the syntax `var = init expr1 next expr2`. The initial value of the variable (`expr1`) is set with `init`, and the value at later time steps is defined using `next`. The `next` expression (`expr2`) is re-evaluated at each subsequent time step to produce the new value of the variable at the present time. Further, the previous value of a variable may be accessed using the primitive `prev`, e.g. `prev var`. Indeed, the `next` expression frequently utilizes the `prev` primitive to express dependence on the past. For example, the definition of the Mario object in the example program from the introduction is `mario = init (Mario (Position 7 15)) next (moveDownNoCollision (prev mario))`, indicating that later values of Mario should move down one unit from the previous value whenever that is possible without collision.

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

```
on event
   update_function,
```

where `event` is a predicate (Boolean expression) and `update_function` is a variable update of the form `var = expr`, or multiple such updates. 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 `update_function` 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.

B. Some Evaluation Details

Due to the page limit, we emphasize that the exposition of our method is a high-level one, omitting lower-level details in the interest of providing an intuitive sense of the technique. For example, one detail skipped for conciseness is that, when selecting a co-occurring event as described in Section 2.2.4., the event with the fewest false positives is chosen from a *subset* of the event space rather than the full event space itself, where the subset consists of events more likely to be actual co-occurring events based on our knowledge of the domain. This reduces ambiguity due to there being multiple co-occurring events that match a given update function. These details will be fully explained in the final version of our paper.

In addition, the ongoing nature of our submission manifests in the fact that several aspects of our evaluation design will be updated before the final version of the paper, especially with respect to the second, externally-sourced benchmark suite. In particular, we manually curated input sequences for our benchmarks in a way we knew would be compatible with heuristics embedded in our algorithm. In future work, we also plan to evaluate on input sequences generated by other, non-expert users in a user study. We further note that we currently declare synthesis success if the synthesized program generates the input observation sequence, though it need not be semantically equivalent to the ground-truth program. We plan to measure proximity to the ground-truth program in the future by evaluating both the synthesized and ground-truth programs on an independent set of test input sequences, and determining how often both programs produce the same output.

Regarding the external benchmark suite, we note that many of those grid-world models actually exhibit some *random behavior*. While a very small amount of non-determinism is also present in a few of our own benchmark models and is handled by AUTUMNSYNTH, we had to manually check that those synthesized programs were accurate by inspection, since it is difficult to automatically check if a particular observation sequence falls into the set that may be produced by a nondeterministic AUTUMN program. (To be more precise about the randomness handling in the algorithm, if a deterministic program is not found, the method searches for (deterministic) events that match update functions that may use a `uniformChoice` library function, e.g. `bullets = addObj bullets (Bullet (uniformChoice (prev sources)).origin))`. Since the AUTUMN programs synthesized to model the external benchmarks are quite a bit

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

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

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

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

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

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

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

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

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

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

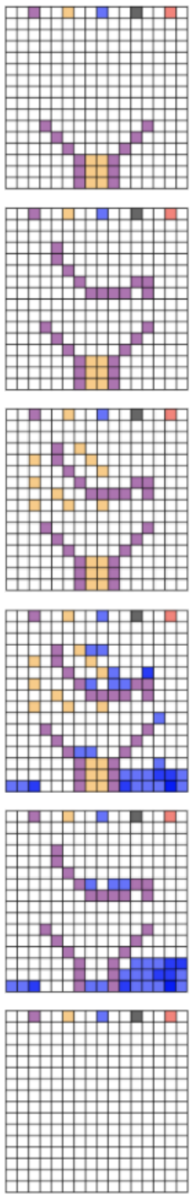
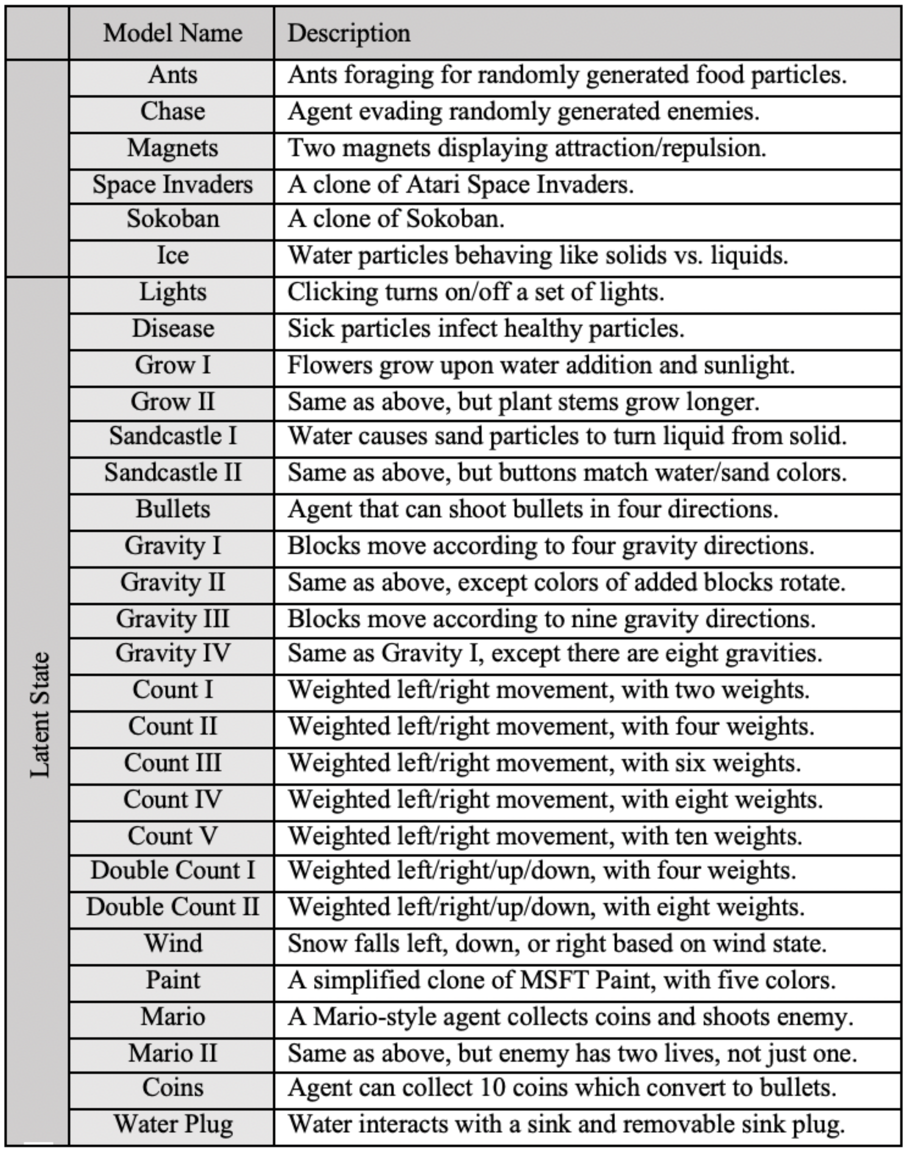


Figure 2. AUTUMN program describing the Water Plug model. In the first frame, the purple structure at the bottom is a vessel, and the orange structure is a plug that does not let water pass into the vessel. Excluding the top row of buttons, purple squares are vessel particles, orange squares are plug particles, and blue squares are water particles. Clicking an uncolored (free) position adds a particle to that position, where the type of particle depends on which of the top-left three buttons was clicked last. The right-side frames are in order (from top to bottom) but with time jumps: the user events during these jumps are the following: 1-2: clicking several free positions (new purple); 2-3: clicking top orange button then several free positions (new orange); 3-4: clicking top blue button then several free positions (new blue, though water moves down rather than being stationary); 4-5: clicking black button (orange removed); 5-6: clicking red button (all removed). We also note that the grid size (16 by 16) and background color (white) definitions are omitted for space reasons.

longer than those for our own benchmark suite, we have not yet performed this manual check on all of those synthesized programs, so it is possible that some are not correct. However, we automatically checked that the deterministic AUTUMN programs synthesized were correct (i.e. produced the given observation sequence), and spot-checked one of the more complex non-deterministic programs (the Aliens benchmark), and found it was correct by manual inspection. This suggests that our synthesizer is operating correctly on this second benchmark suite.

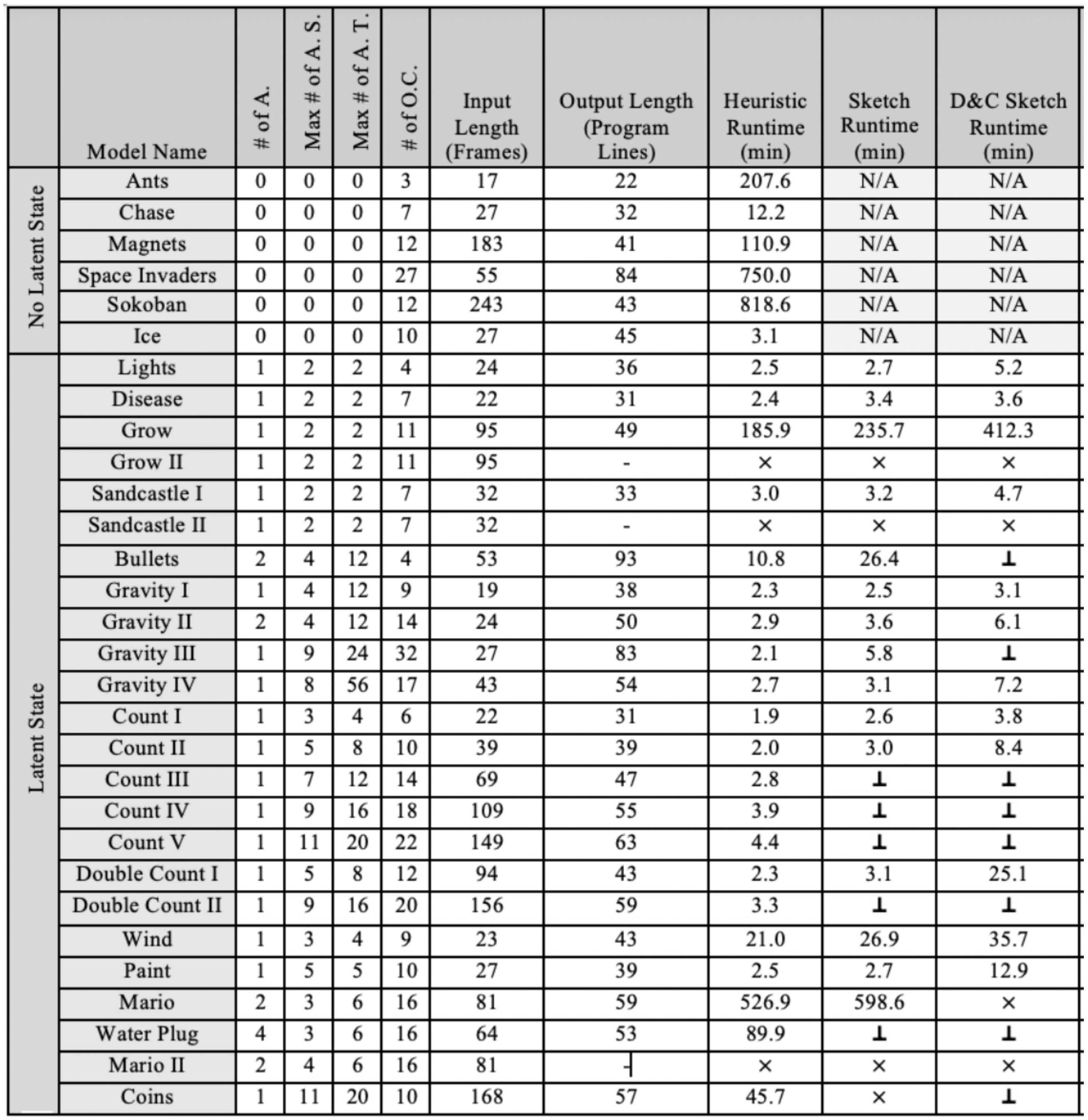
Finally, we take care to emphasize again that this is preliminary work, subject to change as additional progress is made, and should not be used as the basis for overzealous generalization. Still, the fact that the complex dynamics, and especially latent state automata, that underlie these models is being captured in some form by our procedure is exciting.

C. Results



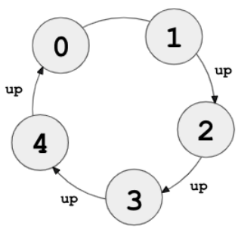
	Model Name	Description
	Ants	Ants foraging for randomly generated food particles.
	Chase	Agent evading randomly generated enemies.
	Magnets	Two magnets displaying attraction/repulsion.
	Space Invaders	A clone of Atari Space Invaders.
	Sokoban	A clone of Sokoban.
	Ice	Water particles behaving like solids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

Figure 3. Descriptions of the 30 benchmark programs in our AUTUMN benchmark suite, called the Causal Inductive Synthesis Corpus (CISC).



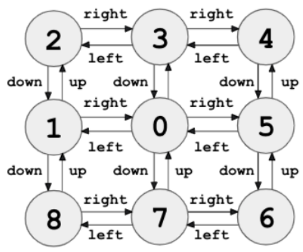
	Model Name	# of A.	Max # of A. S.	Max # of A. T.	# of O.C.	Input Length (Frames)	Output Length (Program Lines)	Heuristic Runtime (min)	Sketch Runtime (min)	D&C Sketch Runtime (min)
No Latent State	Ants	0	0	0	3	17	22	207.6	N/A	N/A
	Chase	0	0	0	7	27	32	12.2	N/A	N/A
	Magnets	0	0	0	12	183	41	110.9	N/A	N/A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

Figure 4. Table of input/output lengths and algorithm runtimes on each of the benchmark programs. The column header abbreviations signify the following: # of A. → # of Automata, Max # of A. S. → Max. # of Automaton States, Max # of A. T. → Max. # of Automaton Transitions, # of O.C. → # of On-Clauses. 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 and thus we did not time the algorithms on these benchmarks. In addition, the N/A's for the Sketch and Divide & Conquer (D&C) Sketch runtimes on the first six 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.



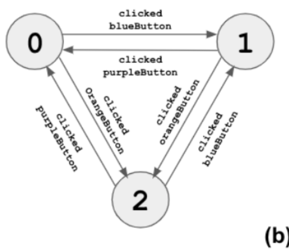
```
% initialize start state
color : Int
color = init 0 next (prev color)
% state transitions
on up && (color == 0)
   color = 1
on up && (color == 1)
   color = 2
on up && (color == 2)
   color = 3
on up && (color == 3)
   color = 4
on up && (color == 4)
   color = 0
```

(a)



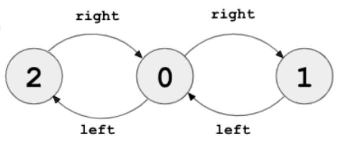
```
% initialize start state
activeDir : Int
activeDir = init 0 next (prev activeDir)
% state transitions
on left && (activeDir == 0)
   activeDir = 1
on up && (activeDir == 1)
   activeDir = 2
            .
            .
            .
on right && (activeDir == 1)
   activeDir = 0
```

(c)



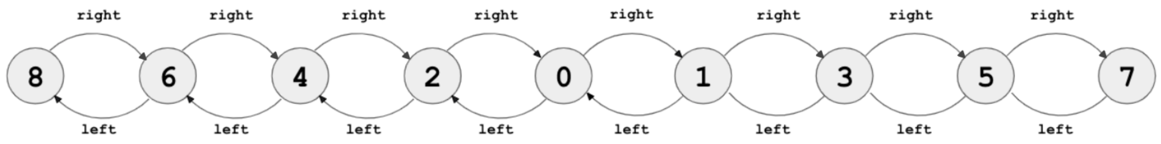
```
% initialize start state
active : Int
active = init 0 next (prev active)
% state transitions
on clicked blueButton
   active = 1
on clicked orangeButton
   active = 2
on clicked purpleButton
   active = 0
```

(b)



```
% initialize start state
windDir : Int
windDir = init 0 next (prev windDir)
% state transitions
on right && (windDir == 0)
   windDir = 1
on left && (windDir == 1)
   windDir = 0
on left && (windDir == 0)
   windDir = 2
on right && (windDir == 2)
   windDir = 0
```

(d)



```
% latent-state-based on-clauses
on true && (weight in [1 3 5 7])
   blob = moveLeft (prev blob)
on true && (weight in [2 4 6 8])
   blob = moveRight (prev blob)
```

(e)

Figure 5. 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.

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

Table 1. Preliminary results from running AUTUMNSYNTH on the external benchmark suite. The runtimes indicate that the synthesis algorithm terminated with a synthesized program, not that the synthesized program necessarily exactly matches the input frame, since that is challenging to automatically check due to the randomness exhibited by most models (exact match checks are performed for the eight deterministic models in the suite, however). As such, it is possible that some of these synthesis successes are not perfect matches to the input sequence, since our checks by manual inspection may not be complete. This will be updated for the final version of our paper; see Appendix B for more details.

D. Additional Figures

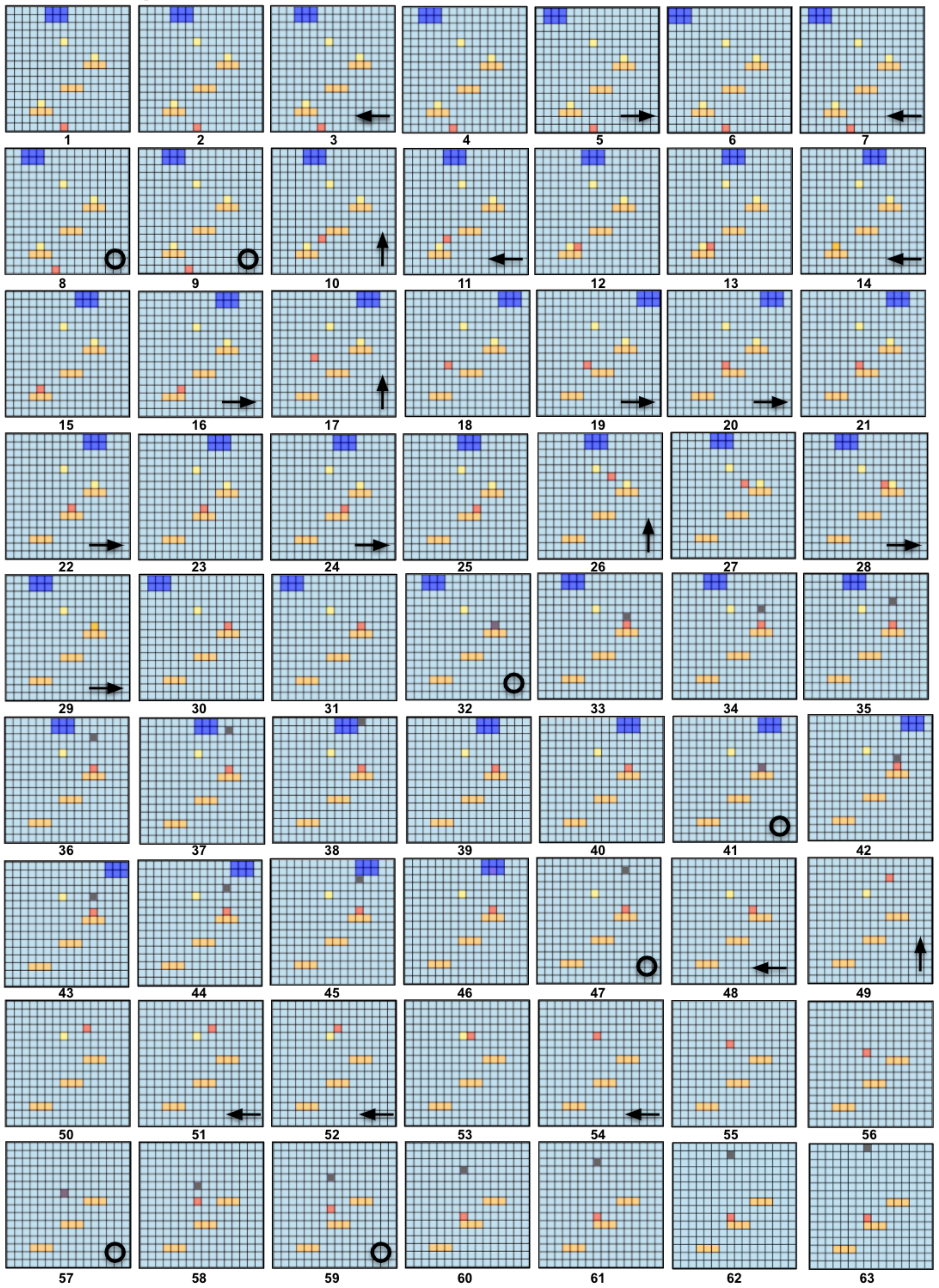


Figure 6. A full observation trace from the Mario program. Black arrows indicate user keypresses and circles indicate user clicks.

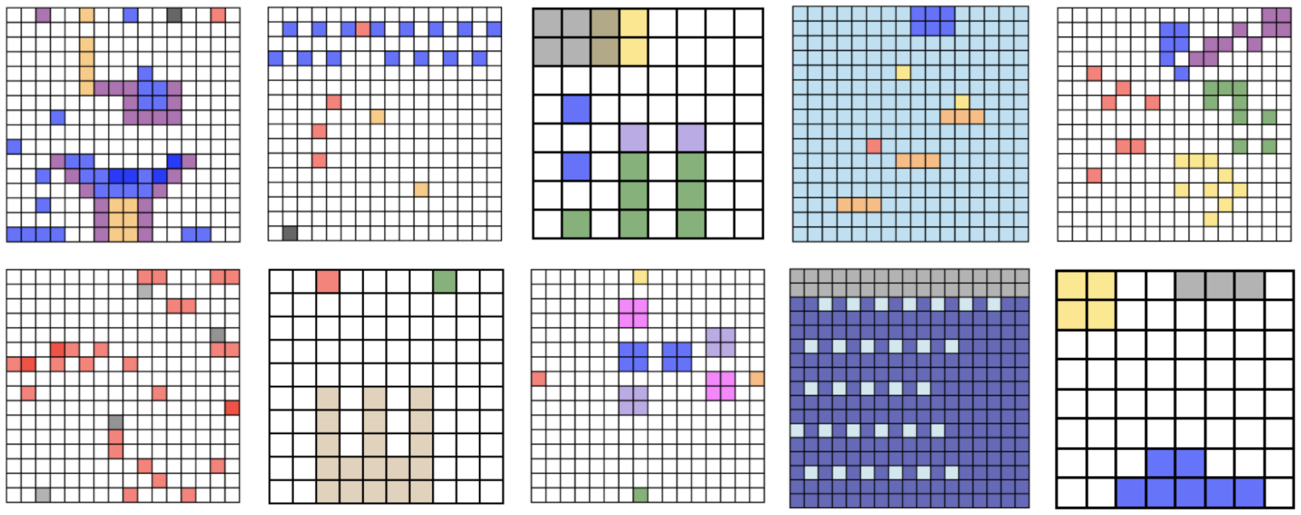


Figure 7. A sample of AUTUMN programs. Clockwise from top-left: water interacting with a sink and sink plug a clone of Space Invaders, plants growing under sunlight and water, a simplified implementation of Mario, a simplified clone of Microsoft Paint, a weather simulation, snow falling left or right with varying wind, an alternative gravity simulation, a sand castle susceptible to destruction by water, and ants foraging for food.

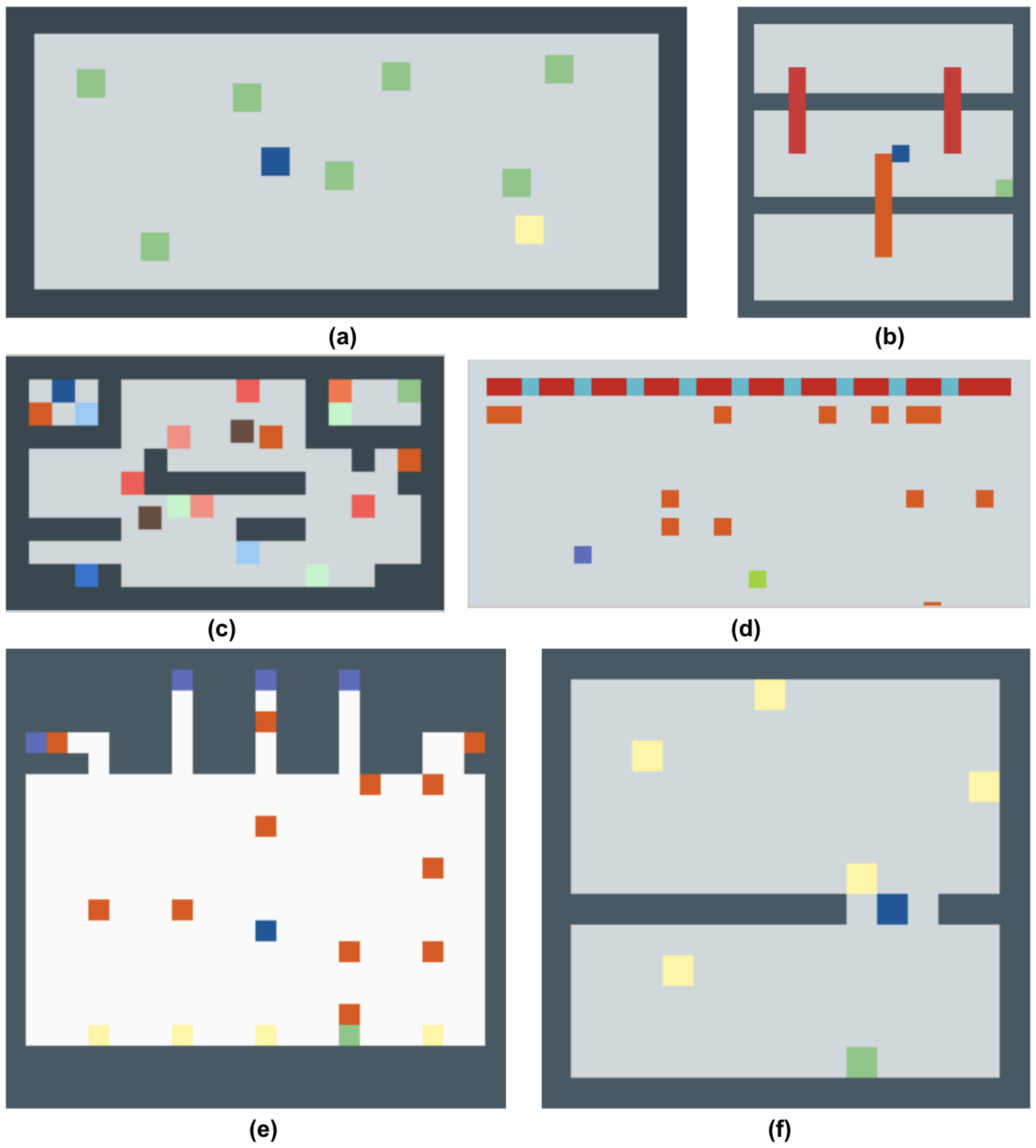


Figure 8. Stills from a sample of programs in the EMPA suite, resized to fit neatly into the figure. (a) Avoid George, where the dark blue agent must avoid the yellow enemy, which chases it and the randomly moving green objects. (b) Closing Gates, in which the dark blue agent must get to the green goal before the gates close. (c) Portals, in which some blocks teleport the agent to other blocks. (d) My Aliens, in which the agent collects orange and is killed by purple objects. (e) Plaque Attack, in which the agent can shoot at orange enemies before they reach the yellow goals. (f) Bees and Birds, where the randomly moving yellow objects can kill the enemy before it reaches the green goal.