Combining Functional and Automata Synthesis to Learn Causal Reactive Programs

by

Ria A. Das
S.B., Computer Science and Engineering and Mathematics,
Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 2022

Certified by .....
Armando Solar-Lezama
Professor
Thesis Supervisor

Certified by .....
Joshua B. Tenenbaum
Professor
Thesis Supervisor

Certified by .....
Zenna Tavares
Associate Research Scientist, Columbia University
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Combining Functional and Automata Synthesis to Learn Causal Reactive Programs

by

Ria A. Das

Submitted to the Department of Electrical Engineering and Computer Science
on May 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Despite impressive advances that have made it the mainstream route towards building human-like AI, deep learning suffers from key limitations that make it unlikely to replicate human intelligence on its own. Specifically, it is very data-hungry, often generalizes poorly to new scenarios, and is not very interpretable, lacking features like compositionality that characterize human knowledge. Given these shortcomings, we explore a different approach to engineering human-like AI called *program synthesis*, in which learned knowledge is represented in the form of a symbolic program. Programs can be learned from limited data and can interpretably capture a wide variety of structured knowledge. However, existing synthesis methods do not scale to long programs that model very complex datasets. In this thesis, we expand the horizon of programs that can be realistically synthesized by bridging methods from two orthogonal communities within programming languages: the *functional synthesis* and *automata synthesis* communities. We focus on the particular domain of causal mechanism discovery in Atari-style grid worlds, and develop a synthesis algorithm that infers a program describing the causal rules of the world from a sequence of observations. We evaluate our algorithm on two benchmark datasets, including one that we constructed using a new programming language called Autumn. Our ongoing results signal the promise of our method, both for modeling efficient, human-like causal discovery and in synthesis and learning contexts more broadly.

Thesis Supervisor: Armando Solar-Lezama
Title: Professor

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor

Thesis Supervisor: Zenna Tavares
Title: Associate Research Scientist, Columbia University

Acknowledgments

I am incredibly grateful to my advisors, Armando Solar-Lezama, Joshua Tenenbaum, and Zenna Tavares, for all of their guidance in bringing this thesis to reality, and in helping me become the researcher I am today. When I joined Armando's lab as one of Zenna's undergraduate research assistants in March 2020, I knew almost nothing about programming languages, even less about cognitive science, even less about what it meant to be a good researcher, and even less about what on earth I wanted to do with my life. Now, two years later, while I still have an enormous amount to learn, I can say that I have achieved much clarity thanks to the interactions and discussions I've had with them. I am especially grateful for all of the challenging but ultimately very insightful conversations about how to best speak to the programming language community versus the AI community versus the cognitive science community about our work, which helped me figure out which communities I'd most like to be a part of after my MEng. I am also thankful for the degree of independence they permitted me to have during the MEng, even though there were many, many months in the beginning when it really wasn't clear if what I was working on had any value at all (I was sure that Armando was going to fire me for wasting his money!). Beyond the research, I also deeply appreciate all of the general life advice they gave me, especially in the last few months as I tried to figure out what to do after graduating.

I owe special acknowledgement to Zenna in particular. Not only was the original idea to study an AUTUMN-like domain his own, but his generosity in sharing his visions both about the project and about the science of intelligence in general greatly influenced my own interests and approaches. He is also incredibly generous in offering his time whether I want to talk about low-level technical problems I'm having, higher-level direction-setting questions, or just life generally. I am still in disbelief that he has responded to so many of my late night/weekend cries for help across time zones and continents, and hope that he will be able to take more relaxing vacations now that I am graduating. Despite my frequent venting about life, I am extremely glad that the other UROP adviser whose job posting I was considering two years ago took

too long to respond to my email.

Finally, I'd like to thank my friends and family, without whose encouragement this thesis and MEng journey would have never happened. There were many moments in the last few years when I almost give up. I'd have done so if it weren't for the people who picked up my phone calls and responded to my messages regardless of the hour, and for them I am extremely grateful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 A Bird's Eye View

In the history of artificial intelligence (AI), the last decade will be remembered as the breakout years of deep learning. From conversational agents like Siri, to machines that can beat human experts at games like Go and chess, to advances in the natural sciences like AlphaFold, neural network-based architectures have promised to touch nearly all aspects of our lives [18, 11]. The versatility of this modeling paradigm have led some to ask whether deep learning alone will be sufficient to achieve the original dream of artificial intelligence, as espoused in the mid-twentieth century: a machine that learns as flexibly as humans do.

Despite their many triumphs, however, deep learning approaches still lag behind human abilities in several key ways, suggesting they are not the end-all solution to the problem of replicating human intelligence. In particular, they are (1) very data-hungry, requiring magnitudes of training data to learn concepts that humans can learn much more efficiently [15, 10, 21]. For example, children can learn how a new toy or video game works in just seconds, while DeepMind's Atari-playing agent MuZero requires significantly more gameplay before learning how to win [18]. Moreover, (2) the learned knowledge in these models, in the form of neural network weights and other latent values, often does not generalize to out-of-distribution scenarios [16]. This flaw manifests in the fact that an agent may make pathological—even danger-

ous, in the case of robots or self-driving cars, for example—mistakes on new inputs despite behaving correctly in old scenarios, due to spurious intricacies deep within its network layers [5]. Finally, the un-human-like generalization mistakes made by these models are particularly troublesome because (3) models learned via deep learning are effectively *black boxes* that are challenging to interpret [13]. This makes it difficult to ascertain what exactly a model has learned and thereby catch errors before they happen, a feature less present in human learning.

Given these shortcomings, in this thesis, we explore an alternative approach to engineering human-like artificial intelligence that answers some of these challenges. Specifically, we posit that using *symbolic programs* as the representation of knowledge learned by an artificial system, as opposed to deep networks, may lead to stronger generalization properties while also being more data-efficient and interpretable. Learning a symbolic program as a model of data is a form of *program synthesis*, a field that sits at the intersection of AI and programming languages. Traditionally, program synthesis was viewed as a technique for automating programming tasks, such as automatically generating ("synthesizing") a tricky function one may encounter in day-to-day programming from a set of input-output examples demonstrating that function's behavior. In the last few years, however, three advantages of programs over deep learning models have fueled the nascent hypothesis that programs may also be a useful *modeling mechanism* in artificial intelligence contexts. These advantages are (1) programs can very compactly and interpretably express a wide variety of structured knowledge; (2) they can often be synthesized from very small input datasets, thanks to the inductive bias embedded in the underlying domain-specific language (DSL) in which the program is expressed; and (3) they often generalize well on new inputs, thanks to their concise forms that protect against overfitting. Recent work has demonstrated the potential of using programs as a model representation in a number of domains, such as synthesizing computer-aided design (CAD) programs from 3D drawings [17].

Unfortunately, for all of its advantages, program synthesis does face several obstacles before it can develop into a mainstream route towards more human-like AI.

One major hurdle is the scalability of program synthesis algorithms. Most traditional methods for program synthesis do not scale to synthesizing long programs that model very complex, intricate datasets. This is because the search space of possible programs that may be expressed in a language is combinatorially large. Further, standard synthesis approaches are general-purpose algorithms, which trade off performance to be applicable to a wide breadth of problems. As a result, most successes of program synthesis in the past have occurred in fairly constrained domains, where the synthesized programs are not much longer than a few lines. In order for program synthesis to become more widely useful, advances must be made to expand the class of programs that can be realistically synthesized beyond just these fairly small programs.

In this work, we make progress on this elusive front by developing a new synthesis algorithm that expands the horizon of programs that could be synthesized with existing techniques. In particular, we develop an algorithm for synthesizing programs known as *functional reactive programs* (FRPs) from observation data in a particular domain. Functional reactive programs are programs that describe the evolution of a system over time, where the system can change in *reaction* to external signals in addition to its normal, unperturbed evolution. The *functional* descriptor indicates that the state of the system is a pure function of its state at the previous time, without any side effects. Since real-world phenomena are time-varying, reactive programs may be used to model a wide variety of useful real-world settings, from industrial equipment controllers to the changing environment of a robot or self-driving car over time. For our purposes, we instantiate our synthesis algorithm in the particular domain of time-varying, Atari-style grid worlds, and synthesize a program that encodes the causal dynamics or "rules" underlying the game from an observed sequence of grid frames and player actions. Beyond providing a simple environment in which to explore the fundamental nature of our algorithm, learning models of video game dynamics is also a relevant application to cognitive science, where scientists have studied how children learn "theories" of how games work. In the following section, we provide more concrete details about this connection to cognitive science as well as about our specific algorithmic contribution, which will set the stage for the rest of this thesis.

1.2 Bridging Functional and Automata Synthesis

Much of the work at the intersection of program synthesis and AI can be framed as addressing the challenge of *theory induction*: Given an observation, what is the underlying *theory* or *model* that generates or explains that observation? We use *theory* to mean not just formal scientific theories, but also everyday cognitive explanations that humans derive on the fly to explain new observations. For example, a child who has figured out how a new toy works after a few minutes of play has come up with a *theory* of the toy's mechanism. While there are many possibilities for the choice of theory representation in AI systems, programs offer the benefits that they can often be synthesized from small data (sample-efficiency) and that their concise, modular form often gives them strong generalization properties. These features have made program synthesis popular in cognitive AI as a potential 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, or synthesizing programs from input-output examples, cannot synthesize non-trivial reactive models. This is because synthesizing *time-varying unobserved state* or *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. For instance, a classic teaching example for inductive program synthesis is a synthesizer that takes the set of input-output pairs $\{(2, 5), (3, 7), (4, 9)\}$ and outputs the function $f(x)$ $\{2x+1\}$ expressed in some programming language. In other words, there are no concerns about identifying

latent state, as both 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 introduce the simple yet rich domain of Atari-style, time-varying 2D grid worlds that we study in this thesis (Figures 1-1, 1-2, and 1-3), which demonstrates these shortcomings of inductive program synthesis. In the Mario-style game in this domain that is shown in Figure 1-1, an agent (red) moves around with arrow key presses and can collect coins (yellow). If the agent has collected a positive number of coins, when the human player clicks, a bullet (gray) is released upwards from the agent's position, and the agent's coin count is decremented. Otherwise, clicking does nothing. Notably, the number of coins that the agent possesses is not displayed anywhere on the grid at any time (i.e. it is not observable), 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.

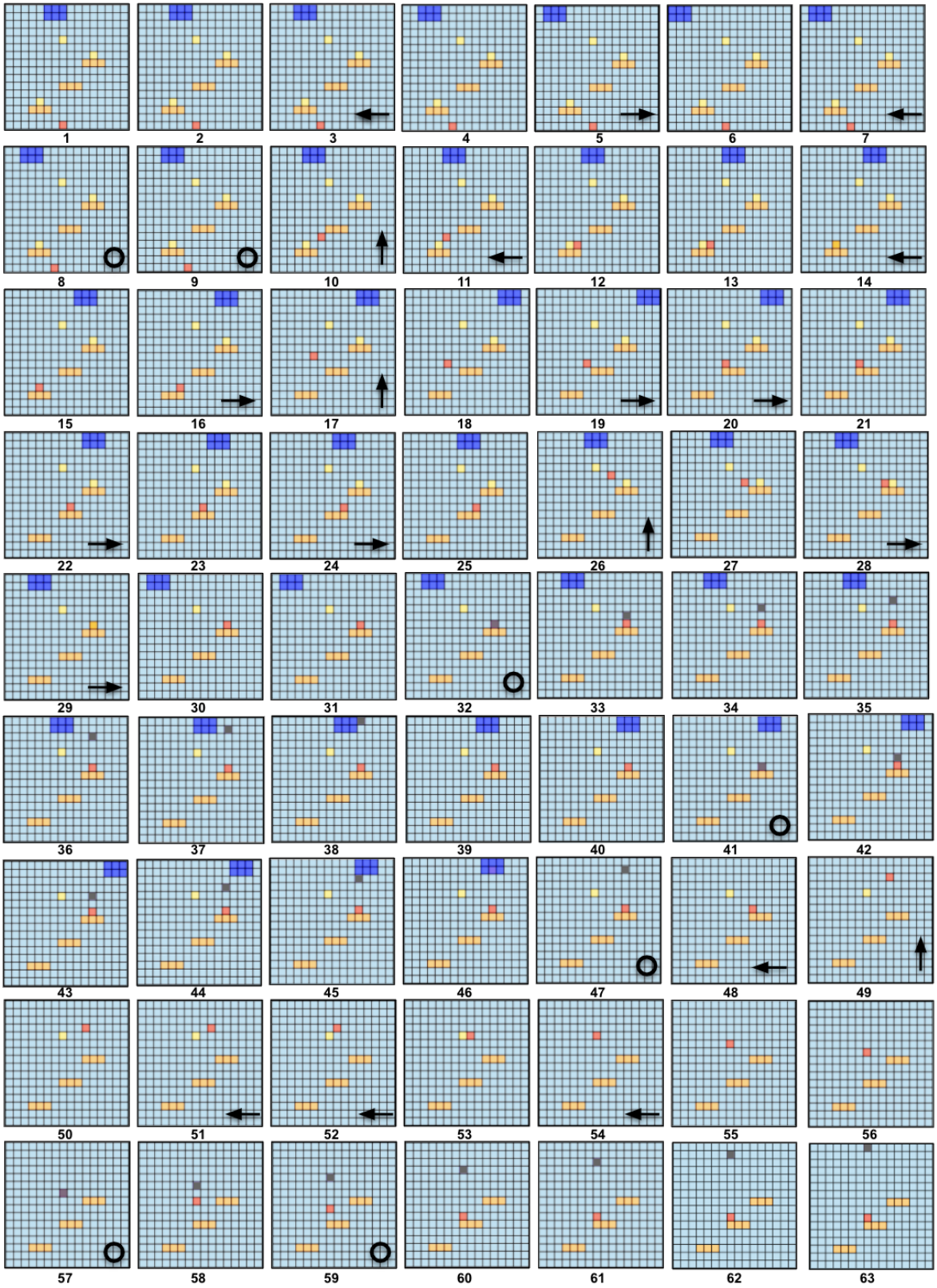


Figure 1-1: An observation trace from the Mario program. Black arrows indicate user keypresses and circles indicate user clicks.

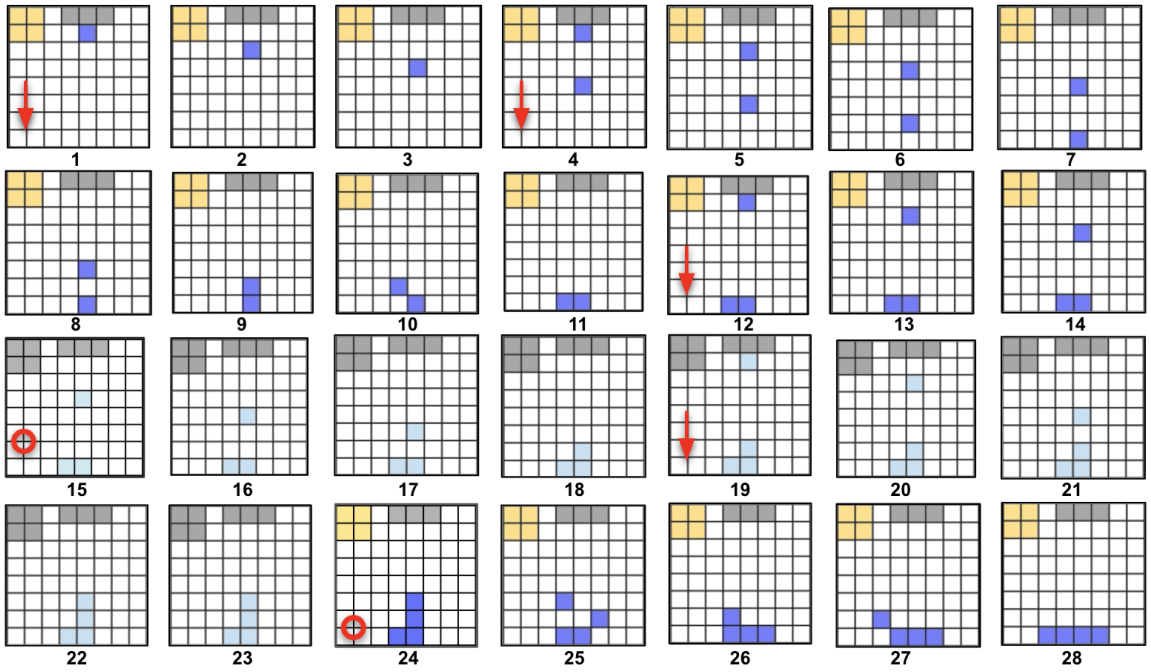


Figure 1-2: Sequence of grid frames from the Ice program. At times 1 and 4, the user presses down (red arrow), releasing a blue water particle from the gray cloud. The water moves down to the lowest possible height, moving to the side (time 10) if necessary to reach this height. The user presses down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

To address this gap between current inductive program synthesis approaches and the reactive setting, we develop a novel program synthesis algorithm that unites two largely orthogonal communities within programming languages: the *functional synthesis* and *automata synthesis* communities. Specifically, we show that we can inductively synthesize reactive programs by splitting synthesis into two procedures, a functional synthesis procedure and an automata synthesis procedure. The functional synthesis step attempts to synthesize the parts of the program that do not depend on latent state. If functional synthesis fails to synthesize a program component explaining an observation, the automata synthesis procedure is called. The automata

synthesis procedure is so named because the time-varying latent state in a reactive system can be viewed as a *finite state automaton*, where the labels on the automaton transitions are predicates in the underlying domain-specific language (DSL) used for synthesis (Figure 1-4). At a high level, based on the specifics of how the functional synthesis step failed, the automata synthesis procedure *enriches* the original program state with particular new latent structure (e.g. a time-varying latent variable like number of coins) that then allows that functional step to succeed.

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

We suspect that this concept of integrating functional and automata synthesis is valuable to a wide breadth of synthesis domains, though in this paper, we concentrate on the 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 grid worlds. The inductive synthesis problem addressed by our algorithm is, given a sequence of observed grid frames and corresponding user actions (clicks and keypresses), to synthesize the program in the AUTUMN language that generates the observations. Since AUTUMN programs encode causal dynamics, this synthesis problem is one of *causal theory induction*, and is important in both cognitive science and AI. This interest stems from the fact that humans are able to learn *causal theories*—or full explanations of which stimuli *cause* which changes in the environment—of grid worlds incredibly quickly, a feat yet to be replicated by AI systems. Since these fields

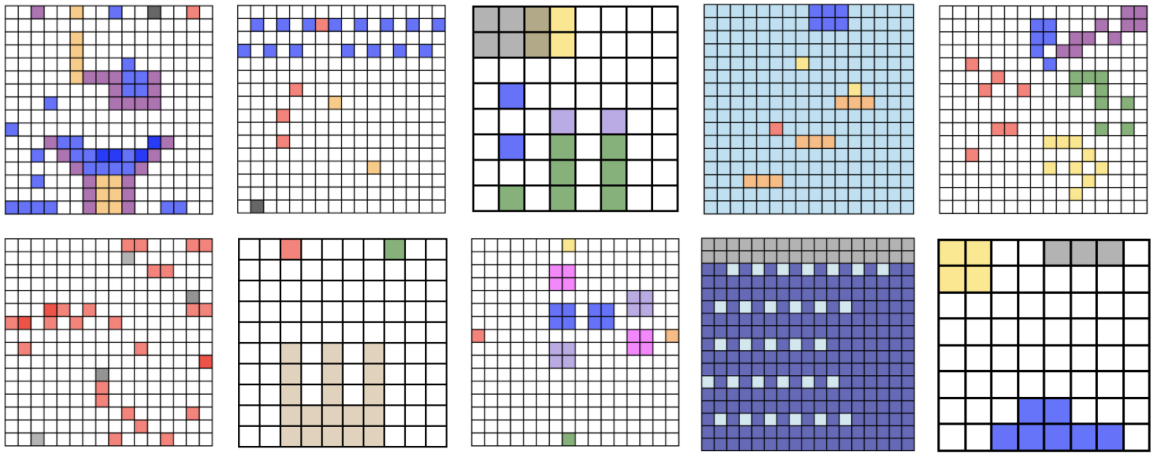
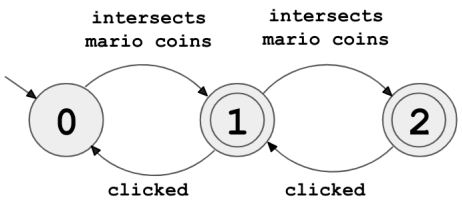


Figure 1-3: A sample of AUTUMN programs. Clockwise from top-left: water interacting with a sink and sink plug a clone of Space Invaders, plants growing under sunlight and water, a simplified implementation of Mario, a simplified clone of Microsoft Paint, a weather simulation, snow falling left or right with varying wind, an alternative gravity simulation, a sand castle susceptible to destruction by water, and ants foraging for food.

aspire to the goal of developing such an artificial agent that can learn causal theories as well as humans can, our hybrid functional-automata synthesis approach offers a potential route towards a solution where program synthesis is the computational engine replicating human theory induction.

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 symbolic synthesis system to discover a minimal latent state automaton from examples, while the third algorithm is a cognitively-inspired, heuristic approach that greedily searches through the space of automata. We construct a benchmark suite of 32 AUTUMN programs designed to express the diversity of time-varying causal models that may be manifested in 2D grids, as well as find an externally-sourced benchmark suite of 27 grid-world environments, and evaluate our algorithm implementations on these benchmarks. While



(a)

```
% initialize start state
numCoins : Int
numCoins = init 0 next (prev numCoins)
% state transitions
on intersects mario coins && (numCoins == 0)
   numCoins = 1
on intersects mario coins && (numCoins == 1)
   numCoins = 2
on clicked && (numCoins == 2)
   numCoins = 1
on clicked && (numCoins == 1)
   numCoins = 0
```

(b)

Figure 1-4: (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.

we are continuing to improve each of the three algorithms, we currently find that our heuristic algorithm and one of the Sketch implementations solve the majority of the benchmarks and do so with similar runtimes, except for a few benchmarks that possess larger automata, on which the domain-specific tricks embedded in the heuristic algorithm currently give it better performance. Our evaluation on the external benchmark remains ongoing, but our preliminary results also show that our algorithm is capturing the underlying causal structures, an exciting confirmation. These empirical results signals the promise of our formulation, and we are eager to continue developing and evaluating our method in the future.

To summarize, we make the following contributions:

(1) a functional reactive, domain-specific programming language called AUTUMN for expressing interesting causal dynamics in 2D, time-varying grid-worlds;

(2) a benchmark dataset of AUTUMN programs to evaluate and spur the development of new learning algorithms;

(3) a novel inductive program synthesis algorithm that learns causal reactive programs from observation data (AUTUMNSYNTH); and

(4) a guiding example of how to design synthesis algorithms that integrate functional and automata synthesis, enabling synthesis of programs beyond the scope

of either alone.

Chapter 2

Background and Related Work

2.1 Program Synthesis

The idea of automated programming is an old one, tracing its roots back to the early days of computing and AI. As early as 1945, Alan Turing claimed the following: "Instruction tables will have to be made up by mathematicians with computing experience and perhaps a certain puzzle-solving ability ... This process of constructing instruction tables should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself" [3]. Indeed, the ideas behind modern theories of computation and the idea to automate the very automation of computation itself did not arise too distantly from each other in the history of the field.

Over the decades, interest in program synthesis has waned and waxed, as new techniques fought with the fundamental challenge of computational tractability that plagues search in a combinatorially large program space. In the last fifteen years, however, the field has seen quite rapid growth thanks to a few fundamental strides forward—like that of the Sketch synthesis engine [20] in the late 2000s—as well as the growing accessibility of large amounts of computational power. An example of program synthesis in the real world today include the FlashFill system shipped with Microsoft Excel, which generalizes a small number of spreadsheet rows written by a user to many rows by synthesizing a small program that maps entries between a row's

cells [9].

With respect to our particular problem setting—learning functional reactive programs that describe the causal dynamics of grid world environments—the most relevant prior work are approaches that synthesize reactive models as finite state machines (FSMs). For example, techniques exist to synthesize communication protocols or embedded controllers like those used in autonomous systems from logical descriptions of their desired behavior over time, including constraints such as what the system should do *until* some event or *always* under a condition [1]. These methods, however, cannot handle model state as complex as our framing, in which objects may be created and destroyed and may each have their own hidden state in addition to coordinates [23]. More precisely, learning a finite state machine as a model of grid world dynamics is at best very computationally intensive and at worst impossible, because even just the visible state of the grid has so many configurations, resulting in an extremely large state space, and the number of objects in the world over time may also be unbounded (i.e. no longer finite). In addition, the specifications in the form of logical formulas that these FSM synthesis approaches begin with provide richer information about the solution space than just raw observations, as we use instead [14].

The other line of related work is that of functional synthesis approaches that can synthesize transformations on very complex data structures, but have no concerns about discovering latent state. An example of this class of approach is work by Ellis et. al. on inferring graphics programs from hand-drawn images [4]. There, a program that specifies how a particular image should be generated by a computational drawing engine is reverse-engineered from the image itself. There is no latent state involved in this procedure, as all the information needed to perform this reverse-engineering from image to program is visible in the image. In contrast, each grid frame in our problem setting may have invisible state in addition to visible state that affects how the state evolves through subsequent grid frames, and we must learn the value of that invisible state in addition to functions that describe how it changes across time.

2.2 Scientist as Child: Theory Learning in Children

Outside of the traditional programming languages community, there has also been interest in program synthesis from cognitive scientists. In particular, young children have been shown to build structured causal theories of their environments [19, 8, 7] in ways very similar to how scientists themselves develop theories. This connection is concisely captured in the phrase "scientist as child," the title of a 1996 study on the topic by cognitive scientist Alison Gopnik [6]. While there has been progress on several related fronts, no artificial system has yet been created that can learn causal theories of new mechanisms as efficiently and flexibly as children can. Current approaches fall short of the goal of human-like causal model discovery from observations in two ways. First, the choice of model representation, e.g. the popular class of *causal graphical models*, is often not expressive enough to concisely capture the complexity of real-world causal mechanisms. Second, even when the model representation is sufficiently rich, e.g. in deep learning, the model learning algorithm is often unrealistically data-hungry, requiring large numbers of observations while humans can generalize from much less.

On the other hand, using programs as models of knowledge learned by an artificial system overcomes both of these concerns, and also gives rise to additional similarities with human cognition. Programs can succinctly express a richer space of theories than graphical representations, and further can often be learned with very little data, unlike deep learning models. Moreover, the compositional structure of programs lends itself well to modeling the flexible ways in which humans *use* knowledge, such as easily abstracting known concepts into higher-level categories and composing disparate ideas to imagine new ones. Foundational work in computational cognitive science has begun to concretize this connection between programs and cognition, including work by Lake et. al. that models the human ability to learn concepts of handwritten characters as a form of *probabilistic program induction* [12]. We take inspiration from these earlier milestones in our thesis, as we try to apply the same technique of employing inductive program synthesis to model human learning in the new, more complex domain of

reactive game-style environments.

2.3 Neurosymbolic Learning

Lastly, a recent exciting development in the broader machine learning community known as *neurosymbolic learning* has some notable connections with our work in this thesis. Neurosymbolic algorithms are a class of techniques that blend deep learning and symbolic methods to take advantage of the strengths of both [2]. More specifically, these algorithms are designed to exploit the ability of deep learning to flexibly recognize complex patterns in data, in concert with the ability of symbolic representations to model higher-level structures and concepts. Examples of neurosymbolic algorithms include methods that synthesize programs where some of the primitives in the programs were neurally learned functions rather than pre-specified in the language, as well as program synthesis engines where enumerative search through a program space is sped up by a neural "guide" suggesting which search paths to check first.

Since the deep learning component of these methods mitigates the impact of the shortcomings of program synthesis, like the need to hand-curate DSLs to apply in different domains as well as the poor scaling of enumerative search, neurosymbolic techniques are a very promising path forward for making program synthesis a more practical route towards human-like AI. As part of future work, we plan to integrate elements of deep learning into the program synthesis algorithm we develop in this thesis to improve its performance and generalizability, as we discuss in Chapter 6.

Chapter 3

The AUTUMN Domain

In this chapter, we describe the AUTUMN domain, which we developed to explore interesting questions at the intersection of program synthesis, causal inference, machine learning, and human cognition. Beyond just the specific directions explored in this thesis, our founding vision for the domain comprised of the following broad objectives:

(1) to highlight the shortcomings of current learning algorithms—from structure learning of causal graphical models (DAGs) to deep learning to program synthesis—when it comes to capturing the complexities inherent in time-varying, real-world causal dynamics given just a few observations;

(2) to provide a substrate to spur the development of new learning algorithms that are able to learn these kinds of generalizable models from minimal training data;

(3) to serve as a platform for designing and running cognitive science experiments with human subjects to gain further insight into how humans learn these intricate causal models so efficiently, where those insights can then be used to further inspire computational approaches.

Notably, while we elected to pursue the development of a new program synthesis algorithm to overcome the limitations highlighted in the AUTUMN domain (due to the advantages of program synthesis that we described previously), attempting to

develop neural or other approaches is also an interesting problem, and at the very least would be important baselines to compare our method against. We leave these directions as future work.

In the following sections, we describe the design of the AUTUMN language, in which a wide variety of interesting causal models in 2D grid worlds may be expressed (Section 3.1); a benchmark suite of AUTUMN programs, which we call the Causal Inductive Synthesis Corpus (CISC), that showcases the expressibility of the language (Section 3.2); and a web interface using which AUTUMN programs can be written and visualized, which we plan to use as the base for a more robust platform for performing cognitive science experiments in upcoming future work (Section 3.3).

3.1 The AUTUMN Language

AUTUMN was designed to concisely express a rich variety of causal mechanisms in interactive 2D grid worlds (Figure 1-3). These mechanisms range from distillations of real-world, everyday causal phenomena, such as water interacting with a sink or plants growing upon exposure to sunlight, to video game-inspired domains such as Atari's Space Invaders. The language is functional reactive, meaning that it augments the standard functional language definition with primitive support for temporal events. Importantly, there is no notion of "reward" in the AUTUMN language, unlike in standard reinforcement learning contexts, which at a glance may look somewhat similar to the AUTUMN domain. The absence of rewards from our design was a conscious choice made because our focus is on learning models of the causal dynamics of these kinds of environments, also known as "world models," rather than how to win. In fact, "winning" is not a meaningful concept in most of the AUTUMN programs we have written, since they need not be games at all.

We next describe the structure of an AUTUMN program (Figures 3-1 and 3-2). Every AUTUMN program is composed of four parts. 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

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

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

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

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

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

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

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

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

on clicked waterButton
  active = "water"
```

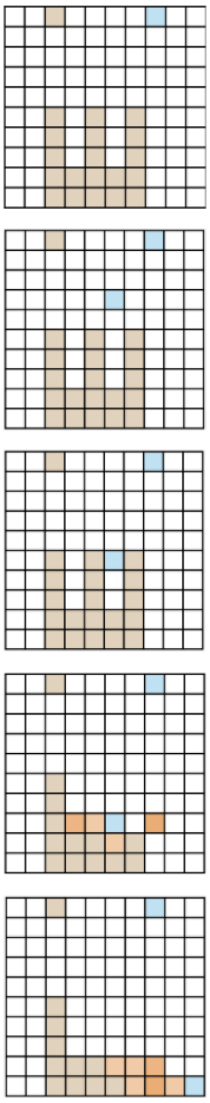


Figure 3-1: AUTUMN program for the Sandcastle model, with frames (ordered, but with time jumps in between) taken from a sample evaluation. The last of the top two buttons clicked dictates whether a sand particle or water particle is added upon clicking a free position. If water is adjacent to sand, it changes the `dry` field of the sand, so that it changes color and behaves as a liquid (i.e. moves to the lowest reachable height) as opposed to stacking.

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

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

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

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

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

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

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

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

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

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

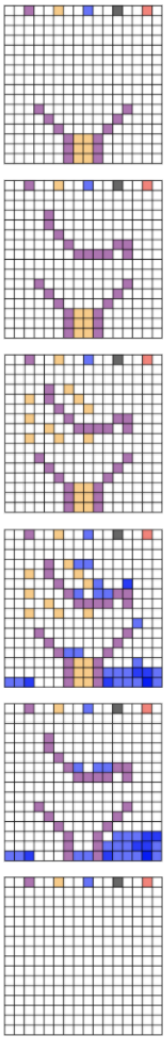


Figure 3-2: AUTUMN program describing the Water Plug model. In the first frame, the purple structure at the bottom is a vessel, and the orange structure is a plug that does not let water pass into the vessel. Excluding the top row of buttons, purple squares are vessel particles, orange squares are plug 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).

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

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

```
on event
   intervention,
```

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

with the effect that later on-clauses may update a variable in a way that composes with updates from earlier on-clauses, or completely overrides it. In the rest of the discussion, we use the term *update function* to mean the same as *intervention*.

3.2 The Causal Inductive Synthesis Corpus (CISC)

Using the AUTUMN language, we constructed a benchmark suite of 32 AUTUMN programs that we call the *Causal Inductive Synthesize Corpus*, where the name indicates that the purpose of the dataset is to serve as a challenge suite for algorithms that inductively synthesize causal models. Brief descriptions of each of the models in the benchmark are given in Table 3.1. Some of the models describe simplified, grid-world versions of real world phenomena such as falling rain and snow or growing plants, while other models are inspired by classic video games like Atari Space Invaders and Super Mario. Still others are more abstract, such as simulations of blocks moving under changing gravities. Six of the AUTUMN programs in the suite do not contain any latent variables, indicating that the state of the program at any time is fully encoded in the visual state, while the remaining 26 models all contain at least one latent variable, which must be inferred from the observations in order to synthesize the correct program. We discuss these models further and in the context of our synthesis algorithm in our evaluation section (Chapter 5).

3.3 Web Interface

To make it easier to write, run, and interact with AUTUMN programs (human-written or automatically synthesized), we developed a web interface that provides these functionalities. A still from the interface is displayed in Figure 3-3. On the left half of the interface, a text editor with syntax highlighting enables users to write AUTUMN programs, currently expressed in the form of Lisp-style S-expressions. On the right side, these programs may be visualized and interacted with, where the control buttons for the simulation (not including the standard arrow key presses and grid clicks that are

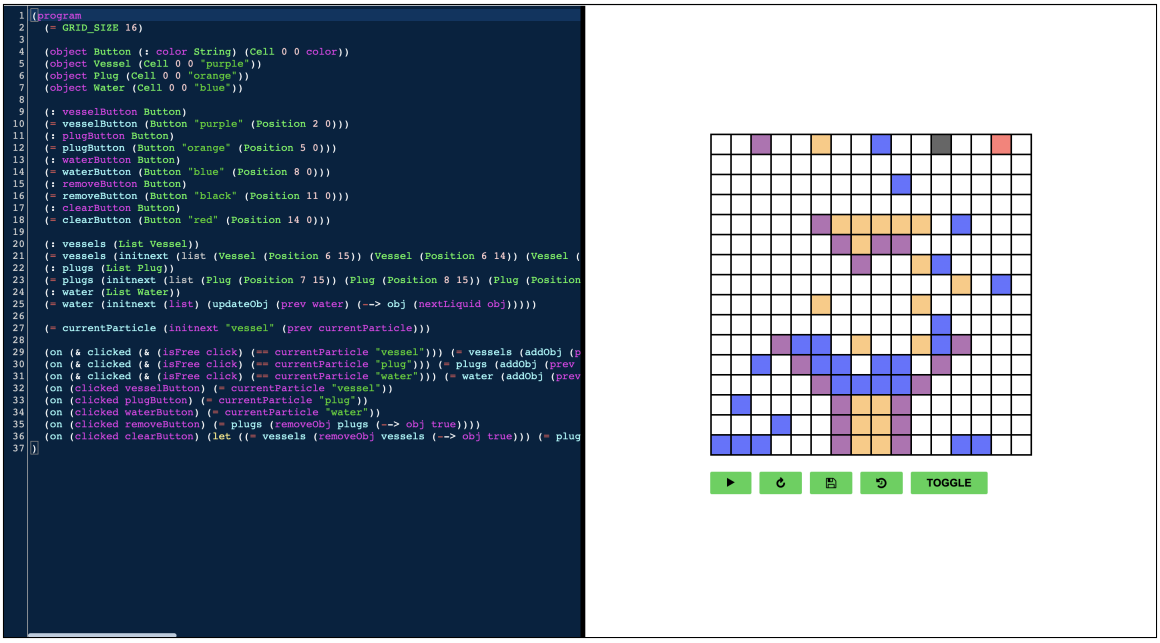


Figure 3-3: Still from the web interface for writing and interacting with AUTUMN programs, showing the Water Plug program from the benchmark suite.

read by the AUTUMN program itself) are a run/pause button, a restart button that resets the program state to the initial state, a replay button that lets a user view the sequence of observations just created, and a save button that writes the sequence of observed grid frames and associated sequence of user events to the local file system. While this web interface has been used for our own language and algorithm development purposes so far, we plan to extend it into a broader platform for others to write and view their own and others' AUTUMN programs, as well as for use in cognitive science experiments with human subjects, as part of near-term future work.

	Name	On-Clauses	Automaton States	Automaton Transitions	Description
	Ants	3	0	0	Ants foraging for randomly generated food particles.
	Chase	7	0	0	Agent evading randomly generated enemies.
	Magnets	13	0	0	Two magnets displaying attraction/repulsion.
	Space Invaders	12	0	0	A clone of Atari Space Invaders.
	Sokoban	7	0	0	A clone of Sokoban.
	Ice	10	0	0	Water particles behaving like solids vs. liquids.
Latent State	Lights	4	2	2	Clicking turns on/off a set of lights.
	Disease	7	2	2	Sick particles infect healthy particles.
	Grow I	11	2	2	Flowers grow upon water addition and sunlight.
	Grow II	11	2	2	Same as above, but plant stems grow longer.
	Sandcastle I	7	2	2	Water causes sand particles to turn liquid from solid.
	Sandcastle II	7	2	2	Same as above, but buttons match water/sand colors.
	Bullets	7	8	12	Agent that can shoot bullets in four directions.
	Gravity I	17	4	12	Blocks move according to four gravity directions.
	Gravity II	9	7	15	Same as above, except colors of added blocks rotate.
	Gravity III	14	9	24	Blocks move according to nine gravity directions.
	Gravity IV	32	8	56	Same as Gravity I, except there are eight gravities.
	Count I	17	3	4	Weighted left/right movement, with two weights.
	Count II	6	5	8	Weighted left/right movement, with four weights.
	Count III	10	7	12	Weighted left/right movement, with six weights.
	Count IV	14	9	16	Weighted left/right movement, with eight weights.
	Double Count I	18	5	8	Weighted left/right/up/down, with four weights.
	Double Count II	12	9	16	Weighted left/right/up/down, with eight weights.
	Wind	20	3	4	Snow falls left, down, or right based on wind state.
	Paint	19	5	5	A simplified clone of MSFT Paint, with five colors.
	Mario	19	5	6	A Mario-style agent collects coins and shoots enemy.
	Mario II	40	7	7	Same as above, but enemy has two lives, not just one.
	Coins I	18	6	10	Agent can collect 15 coins which convert to bullets.
	Coins II	22	8	14	Same as above, with 28 coins.
	Coins III	26	10	18	Same as above, with 45 coins.
	Water Plug	8	3	6	Water interacts with a sink and removable sink plug.

Table 3.1: Descriptions of the 32 benchmark programs in the Causal Inductive Synthesis Corpus.

Chapter 4

The AutumnSynth Algorithm

We next describe the AutumnSynth algorithm, which synthesizes Autumn programs from a sequence of observed grid frames and associated user actions. We begin with a brief overview that explains how the algorithm synthesizes the Mario program from the introduction, and follow with a more detailed treatment of each of the steps of the procedure.

4.1 Overview

Synthesizing the correct Autumn program from observed data involves determining the object types, object instance and latent variable definitions, and on-clauses described in the previous chapter. 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. **object 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 the object was removed in the next time;

3. **update function synthesis**, in which AUTUMN expressions, called update functions, describing each object-object mapping from Step 2 are found; and

4. **cause synthesis**, in which AUTUMN events (predicates) that *cause* each update function from Step 3 are sought, and new latent state in the form of automata is constructed upon event search failure.

We give details for these steps in Section 4.2, with greatest space given to the step of cause synthesis, since that procedure represents the most novel aspect of our work. First, we provide some intuition by briefly describing how these steps are used to synthesize the Mario program (Figure 4-1).

4.1.1 Sample Execution

Phase I: Object Perception

The object perception step first extracts the object types and object instances from the input sequence of grid frames. The object types are (1) a general single-cell type with a string color parameter corresponding to the (red) agent, (yellow) coin, and (gray) bullet objects; (2) a platform type that is a row of three orange cells; and (3) an enemy type that is a rectangle of six blue cells. A list of object instances is extracted from each grid frame in the input sequence, where an object instance describes the object's type, position, and any field values. For example, the object instances for the first grid frame are a red single-celled object (agent) at position (7, 15); three yellow single-celled objects (coins) at positions (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).

Phase II: 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

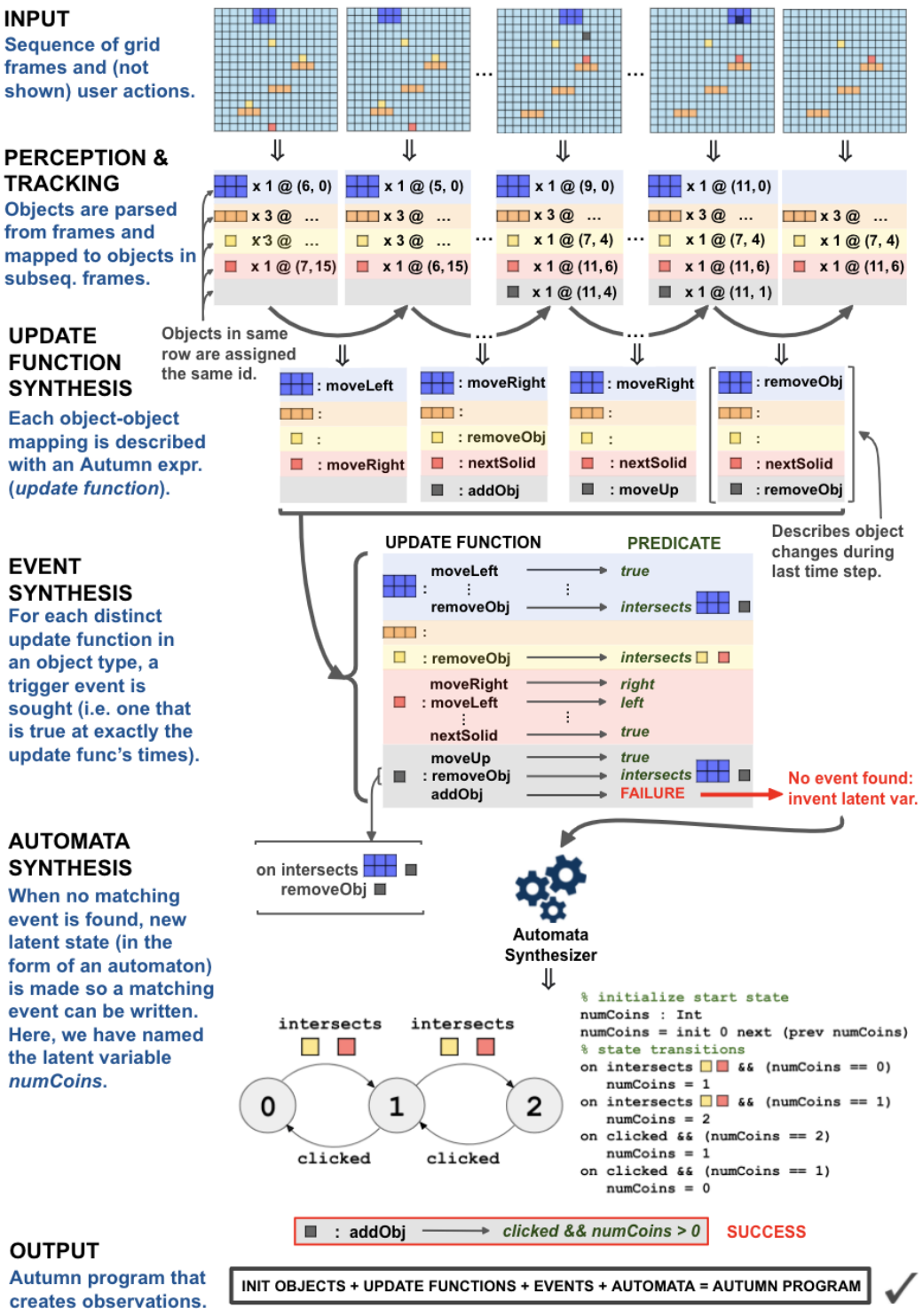


Figure 4-1: Sample execution of the AUTUMNSYNTH algorithm on an input sequence taken from the Mario example. The event and automata synthesis steps together compose the cause synthesis step (Phase IV) as described in Section 4.1.1.

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.

Phase III: Update Function Synthesis

In the third step of update function synthesis, for each mapping between an object in one grid frame and an object in the next that is determined in Step 2, an AUTUMN expression is sought that describes that object-object mapping. For example, this step identifies that the expression `agent = moveLeft (prev agent)` accurately describes the change undergone by the agent object between the first and second grid frames. Often, there are multiple such expressions that match any given mapping. For example, the agent's left movement during the first time step might also be described by `agent = moveLeftNoCollision (prev agent)` or `agent = moveClosest (prev agent) Platform`, where the latter indicates movement one unit towards the nearest object of type `Platform`. The update function synthesis step collects a set of these possibilities for each object mapping. Ultimately, one update function is selected as the single description for each object-object mapping during the final step of cause synthesis.

Phase IV: Cause Synthesis

Finally, the cause synthesis step searches for an AUTUMN event or predicate that triggers each update function identified in Step 3. For now, we will assume that we have already selected a single update function that matches each object-object mapping from the set of all possible update functions that do so; we will explain how we perform this selection process in Section 3. To find an AUTUMN event that triggers a particular update function, we collect the set of times that the update function takes place, and enumerate through a space of AUTUMN events until we find one that evaluates to true at each of those times. For example, say that the agent object in Mario undergoes the update function `agent = moveLeft (prev agent)` at times 1, 4, and 5. If the AUTUMN event `left`, which indicates that a left keypress

has occurred, evaluates to true at those three times, then the on-clause

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

accurately describes that particular update function's occurrence. The search space of AUTUMN predicates is defined over the *program state*, which consists of the current object instances, latent variables, and user events. At the start of this step in the algorithm, there are not yet any latent variables in the program state, so the possible events use only the objects and user events (e.g. `clicked`, `clicked agent`, or `intersects bullet enemy`). Lastly, this event-finding process is complicated slightly by the fact that on-clauses may *override* each other, so perfect alignment between trigger event and observed update function is not always necessary. This nuance will be explained in Section 4.2.

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

Specifically, we proceed by finding the "closest" event that aligns with the update function in a subset of the event space called the co-occurring event space. This "closest" event *co-occurs* with every update function occurrence, but may also occur during *false positive times*: times when the event is true but the update function does not occur. For bullet addition, this event is `clicked`, as every bullet is added when a click takes place, but some clicks do not add a bullet (specifically, at times 8, 9, 47, and 59). Having identified this closest event, our goal is then to construct a latent variable that acts as a finite state automaton that *switches* states between the false positive

times and true positive times (i.e. the times when `clicked` is true and the update function occurs). To be precise, the new variable takes one set of values during the false positive times, and another set of values during the true positive times. Calling the values taken by the latent variable during true positive times *accept values*, and those taken during the false positive times *non-accept values*, the event

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

perfectly matches the observed update function times. This is because `clicked` is true during a set of false positive times, and `latentVar` is in *non-accept* values at exactly those times, so bullet addition does not take place, as desired. The full AUTUMN definition of `latentVar`, including the *transition on-clauses* that change its value over time, is shown in Figure 1-4. The variable name `numCoins` is substituted to note the equivalence to a *number of collected coins* tracker.

The challenge in constructing this latent variable is learning the transition on-clauses that update the value of the variable at the appropriate times. Note that these transition on-clauses represent *edges* in the *automaton* diagrammed in Figure 1-4 (hence the use of the term *accept values* or *states*). We perform the transition learning step as part of a general automaton search procedure, implemented via a SAT solver as well as heuristically, to be discussed in Section 4.2.4.

4.2 AUTUMNSYNTH Details

We now describe each of these four steps of the synthesis algorithm in more detail.

4.2.1 Phase I: Object Perception

In the perception step, each frame in the observation sequence is parsed into a set of object variables. Each object variable is characterized by a *shape*, which is a list of 2D grid positions relative to $(0, 0)$ that are each associated with a color, as well as an

origin, which indicates the location of the object in the grid frame (the positions of the shape are translated by the origin to obtain the final rendering of each object). We use two different object parsing algorithms, each of which produces a different representation. We perform the rest of the synthesis procedure atop both of these parsing results one at a time, and take the output program from the first parsing using which the procedure succeeds.

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

4.2.2 Phase II: Object Tracking

Together, the second and third steps in the synthesis procedure answer the question, "What does each object do at each time step?" Concretely, this means identifying the *update function* undergone by each object in each frame to produce the object's rendering in the subsequent frame. The first element of answering this question is *object tracking* (Step 2), which involves assigning each object in a frame either to (1) an object in the subsequent frame, which is considered to be the transformed image of the object after the time step, or (2) no object, which means that the object has been removed after the time step. Multiple objects may not map to the same object in the subsequent frame, and further, objects in the subsequent frame without a pre-image in the previous frame are deemed to have been just *added* to

the program in the current time. The algorithm that performs this mapping is based upon a heuristic that embodies the following prior assumption about object motion in AUTUMN: Objects are unlikely to move very far in a single time step. As such, the tracking algorithm performs assignments based on a proximity metric that tries to maximally assign objects in one frame to their closest objects (with the same object type) in the next frame.

4.2.3 Phase III: Update Function Synthesis

The update function synthesis procedure computes an AUTUMN expression, the update function, that describes every object-object mapping. This includes update functions describing object addition and removal, which are represented as mappings with a `null` or non-existent object: a `null`-object mapping indicates object addition and an object-`null` mapping indicates object removal. These update functions will eventually become part of the on-clauses in the final output program.

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

At the end of this process, the synthesized update functions may be visualized in a matrix depiction, which we call the *update function matrix* (Figure 4-2). In the update function matrix, the rows represent object_id's, where objects are assigned the same object_id if one is transformed into the other over time, and the columns represent times in the observation sequence (in increasing order). Each cell in the update function matrix contains the set of possible update function expressions corresponding

to that particular object_id at that particular time, or more precisely, those possibly undergone by the object *between* the frame at that time and the frame at the next time.

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

To handle this uncertainty with regard to which single update function in 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

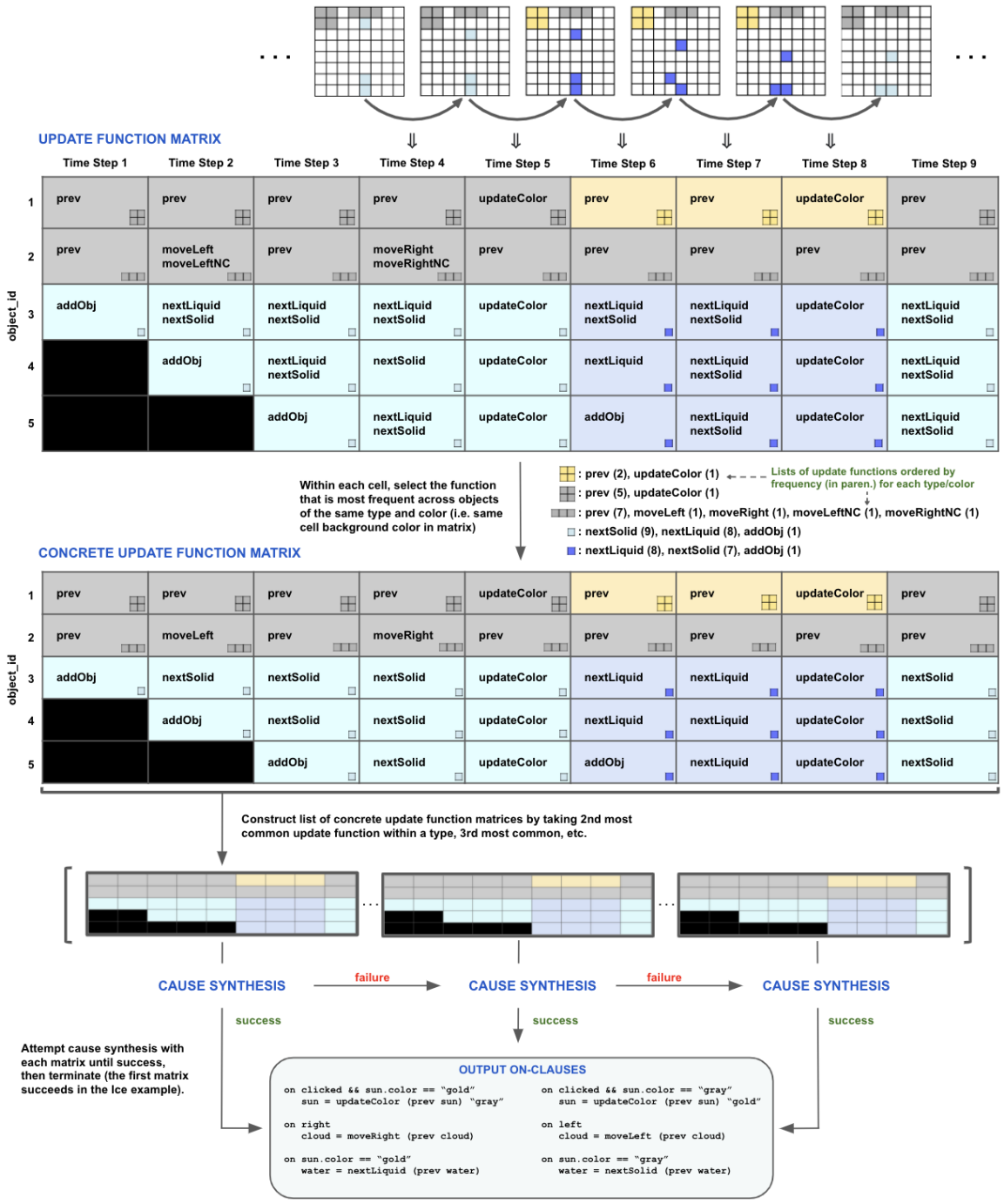


Figure 4-2: Update Function Synthesis. Each cell of the update function matrix contains a set of update functions that each describes the change undergone by the object with object_id equal to the row index during a particular time step (column index). A list of concrete update function matrices, with one update function per cell, is extracted via frequency-based heuristic.

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. Specifically, the heuristic operates by determining the top-k most frequent update functions across each object type in the program, and constructs a concrete matrix by choosing the ith most frequent update function whenever it exists in a cell, for i in 1 through k, independently for each object type (Figure 4-2). In other words, the output is a list of possible concrete matrices, where the first matrix is constructed by choosing the first most common update function for every object type, the second matrix was constructed by choosing the *second* most common update function for the first object type but the first most common update function for the rest of the types, the third was creating by choosing the second most common update function for the second type and the first most common for all the others, and so on. Hence, there are a maximum of k^n concrete update function matrices produced if there are n distinct object types and each combination produces a unique filtering. In the three-frame and single-object-type example observation sequence described previously, the update function sequence `moveLeft`, `moveLeft`, `moveLeft` would be selected for the first concrete matrix, because `moveLeft` occurs three times in update function matrix while `nextLiquid` occurs only twice.

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 4-2.

Nondeterminism in AUTUMN Programs and CISC

Before discussing the subsequent cause synthesis phase of the algorithm, we briefly comment on support for writing nondeterministic programs in the AUTUMN language. AUTUMN provides a built-in function called `uniformChoice`, which selects one ele-

ment uniformly at random from a non-empty list. Using this operator, a vast range of interesting causal probabilistic AUTUMN programs can be written. However, since inferring the correct probability distribution described by a probabilistic AUTUMN program adds a completely new level of complexity, our current synthesis algorithm is focused only on synthesizing *deterministic* AUTUMN programs, with one very small-scoped exception. This exception is non-nested use of the `uniformChoice` operator at the *update function* level of an on-clause, but not at the *event* level.

Specifically, if no deterministic AUTUMN program is found by the synthesizer, i.e. cause synthesis fails on every concrete update function matrix identified through the update function synthesis step, the algorithm will try to construct new concrete matrices using `uniformChoice`-based update functions. Currently, the algorithm only allows these random update functions to have the form `addObj (uniformChoice [ \* list of object positions *\ ])`. For example, it is possible that the set of possible update functions in the unfiltered matrix for a certain object at a certain time is `{ addObj (Bullet (Position 5 5)), addObj (Bullet (uniformChoice (map (-> obj obj.origin) sourceObjects))) }`, if (5, 5) is the location of an object in the list. The Space Invaders model in CISC displays this structure, as the bullets in that program are shot out of a randomly selected enemy object at regular time intervals. Hence, it is possible that a matching event may be found for a `uniformChoice`-based update function, even if none are found for deterministic update functions, so this limited form of nondeterminism in AUTUMN programs is supported by the synthesizer when a deterministic solution cannot be found.

4.2.4 Phase IV: 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 used 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 a set of 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.

Automata Synthesis

The input to the automata synthesis step is a set of update function trajectories, one for each unmatched update function from the previous step. Each update function trajectory is a single vector $v \in \{-1, 0, 1\}^T$. The goal of the automata synthesis procedure is to construct the simplest latent state automaton that enables us to write latent-state-based event predicates matching each v. For ease of exposition, we will begin by describing the automata synthesis procedure for the scenario in which there is exactly one unmatched update function for which a latent-stated-based predicate must be constructed. We will then describe the extension to the more general scenario of multiple unmatched update functions.

To start, we frame our overall problem with respect to the classic formulation of automata synthesis given input-output examples. Classically, the problem of inductive automata synthesis is to determine the minimum-state automaton that accepts a given set of accepted input strings (positive examples) and rejects a given set of rejected input strings (negative examples). In our scenario, these positive and negative input "strings" may be determined from the sequence of program states (one per time) corresponding to the observation sequence. In particular, we consider the set of *prefixes* (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 (Figure 4-3). 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. Since there may be multiple such co-occurring events with the same minimal number of false positives, we further restrict search for this event to a *co-occurring event space* that is smaller than the full event space, and contains events that are more likely to be correct co-occurring events, based on our knowledge of the domain. In the Mario example, this co-occurring event is `clicked`. We then partition the set of program state sequence prefixes into those that end with a program state in which the update function took place and those in which it did not take place. The former

set is the set of positive examples and the latter is the set of negative examples in our automata synthesis problem.

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

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

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

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

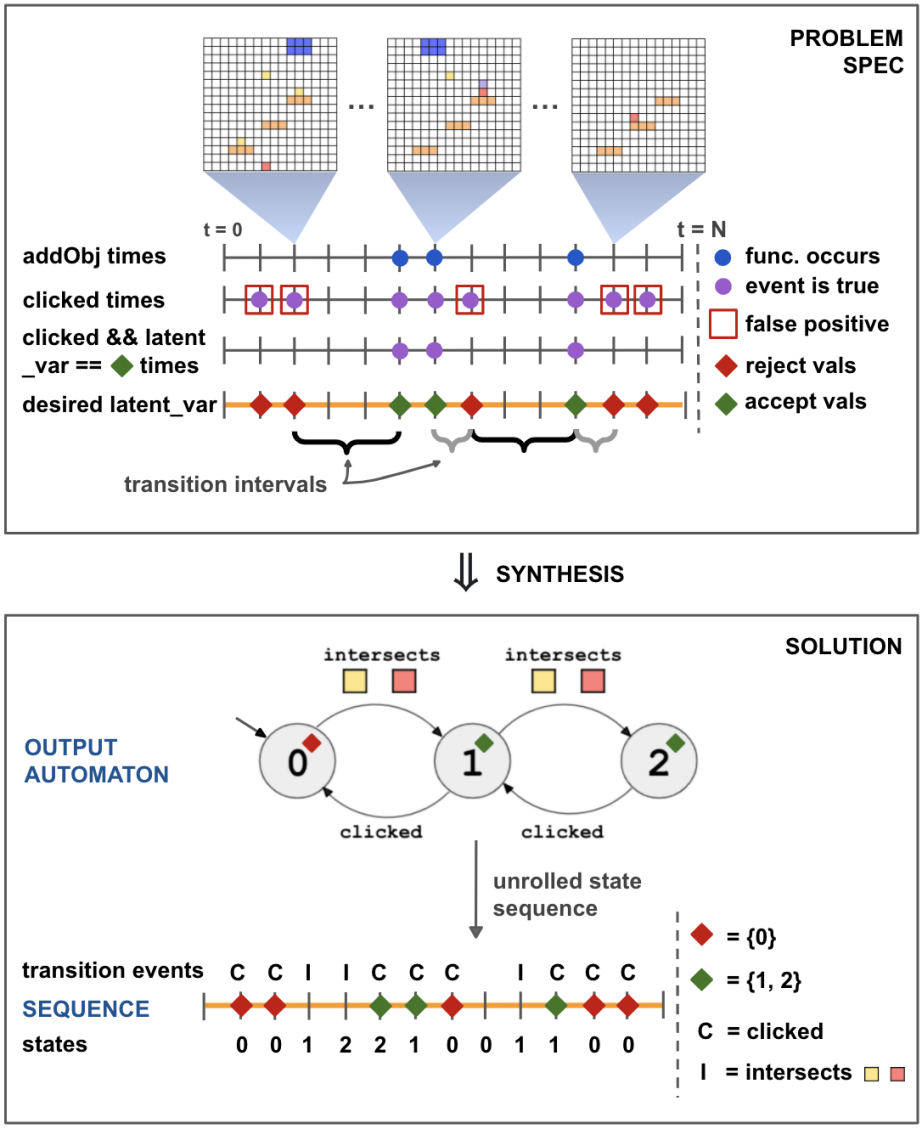


Figure 4-3: Bird's-eye view of the automata synthesis problem, using the example of the Mario program. The bullet addition update function, indicated by `addObj`, does not have a matching trigger event. The closest event is `clicked`, which co-occurs with bullet addition but also is true at false positive 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.

We now discuss three distinct algorithms for solving this inductive automata synthesis problem: Full Sketch, Divide-and-Conquer Sketch, and Heuristic. We note that at the current stage of this ongoing work, we synthesize latent automata that satisfy all unmatched update functions within *each object type*, as opposed to sharing automata for use across different object types. The reason for this is because the human-written AUTUMN programs in our benchmark suite use different latent variables for each type—a choice that appears to make the programs more human-understandable than having one large product—and these sets of type-level latent automata are also often more concisely expressed in the AUTUMN language than a single product.

Algorithm 1: Full Sketch

In the Full Sketch approach, Sketch is tasked with identifying the minimal automaton that, for a *set* of unmatched update functions, accepts each update function's language as specified by the observed examples up to changing just the accept states corresponding to each update function. Specifically, the set of all unmatched update functions for each object type is divided into groups that have the *same co-occurring event*, and Sketch tries to find a satisfying automaton (i.e. latent variable) for each of these groups. As an example, consider the AUTUMN program named Gravity I shown in Figure 4-4. The blue blocks continuously move left, right, up, or down depending on which of the four colored buttons was last pressed. A matching event cannot be found for any of the four update functions `moveLeft`, `moveRight`, `moveUp`, or `moveDown`, and they all have the same co-occurring event of `true`, so their update function trajectories are fed to the Sketch solver to produce the 4-state automaton shown in Figure 4-4a. This new latent variable then allows a matching predicate to be written for each of the four update functions: `true && latentVar == 1`, `true && latentVar == 2`, `true && latentVar == 3`, and `true && latentVar == 4`.

Algorithm 2: Divide-And-Conquer Sketch

Rather than attacking multi-automata synthesis problems head on, Divide-And-Conquer Sketch tasks Sketch with solving each update function's automata synthesis problem *individually*, and then combines those solutions together via product. The intuition behind this approach is that synthesizing an automaton matching multiple update functions at once could face scalability challenges, but finding an automaton matching a single update function, which is likely smaller, may be easier. As described previously, the smallest automaton satisfying a single update function may not give to rise to the smallest product, so the Divide-and-Conquer algorithm identifies a small *set* of automata matching each update function instead. It then takes the product over all update functions' automata sets, and computes the minimal automaton from that product space. We illustrate this algorithm again with the Gravity I example (Figure 4-4b). The algorithm first identifies a set of automata that solve the automata synthesis problems corresponding to the four unmatched update functions. Note that each of these automata have just two states instead of the full 4-state solution found in the Full SAT approach. Next, it computes all automata *products* over these four automata sets, and takes the minimal automaton from this product set, which is the 4-state solution seen previously.

(A note about Figure 4-4b: For reasons of tractability, we employ a simple heuristic to downsize each individual update function's automata set before taking the product across all automata sets. At a high level, this heuristic identifies subsets of the full automata set that are *observationally equivalent* with respect to the given input observation sequence, and keeps just one automaton from each of these equivalence classes. This step is not shown in the figure. We will give a more detailed explanation of this procedure and definition of observational equivalence in the final version of this paper.)

Algorithm 3: Heuristic

Despite the simplicity of the Sketch-based formulations of automata synthesis, their scalability to problem settings with large automata is unclear, due to known limitations of SAT solvers. As such, we also implemented a heuristic algorithm that synthesizes an automaton satisfying a set of update function trajectories via a series of greedy updates to an initial automaton (Figure 4-4c). At a high level, this approach begins with an automaton with a small number of states, and repeatedly *splits* states into two based on a heuristic related to the search for transition events. More precisely, the algorithm begins by searching for transition events (edges) that result in an automaton that produces a particular initial state sequence that has few distinct states. If transition search fails, one of the original states is split into two, and transition search is repeated. Since state splitting changes the specifications for the desired transition events, in other words by changing when the transition event must be true versus false to produce the desired state sequence, it enables transition search after a first failure. This cycle of transition search and state splitting continues until a satisfying automaton is identified.

Lastly, we note that the descriptions in this chapter provide a high-level view of our synthesis algorithm, omitting some lower-level heuristic variations and details. We will formalize and include these elements for the final version of this paper.

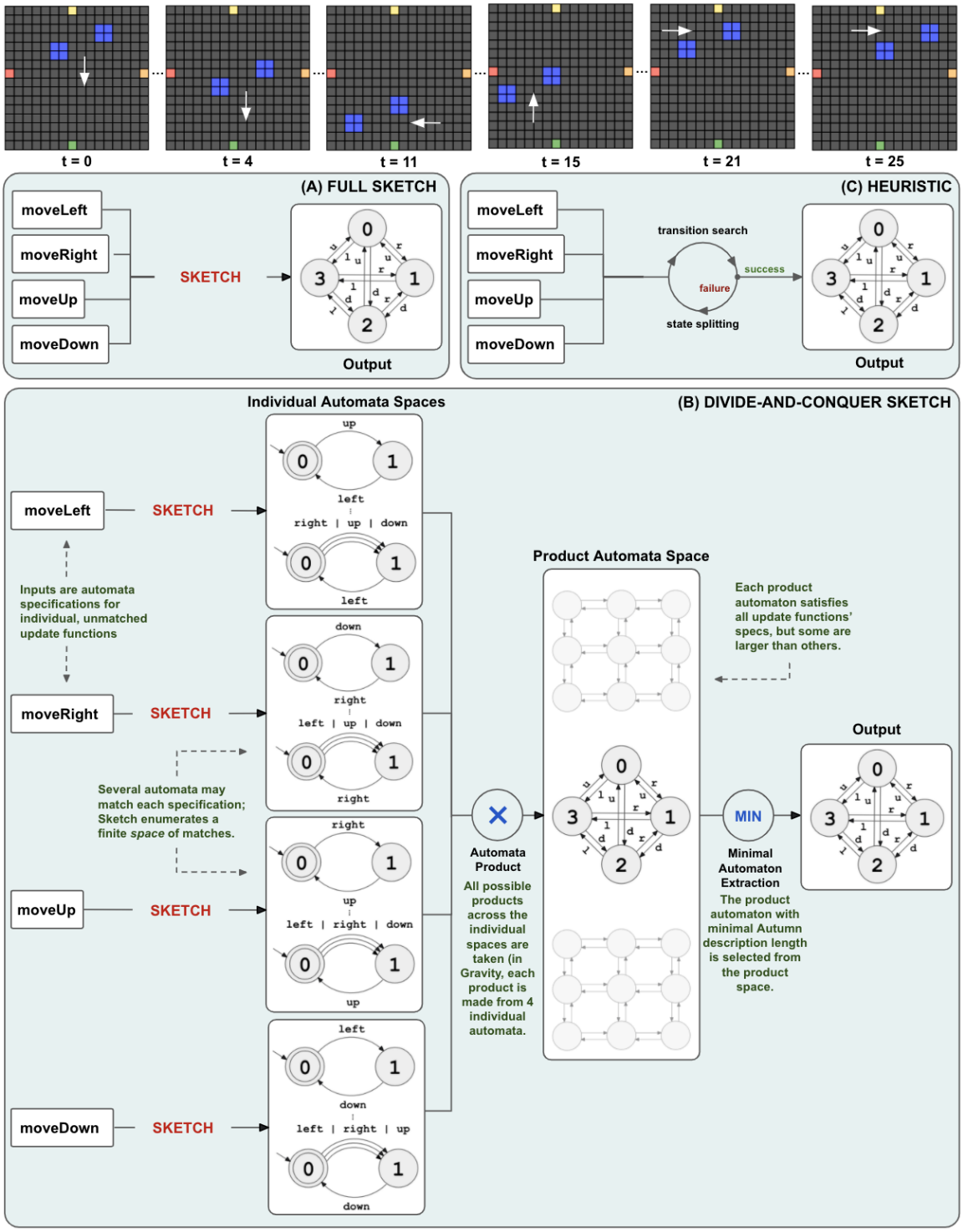


Figure 4-4: Three variant methods for automata synthesis, shown for Gravity I. The blue blocks move left, right, up, or down depending on the button last clicked. The transition label `left` abbreviates `(clicked leftButton)`, etc. See note in Sec. 4.3.2.

Chapter 5

Evaluation

5.1 CISC Benchmark

As our evaluation remains ongoing, for our preliminary results, we manually constructed an input user action sequence for each benchmark program, and ran the three synthesis algorithms—Full Sketch, Divide-and-Conquer Sketch, and Heuristic—on these sequences. We declared a success for a synthesis algorithm if it produced an output program that matched the observation sequence, though it need not be perfectly equivalent to the ground-truth program (for the benchmarks with random behavior, this equivalence was checked by manual inspection, since it is otherwise difficult to determine if a particular observed sequence could be produced by a nondeterministic AUTUMN program). Both of these aspects will be updated in our final evaluation, in which we plan to measure the success of our synthesis algorithms on input sequences generated by several human subjects interacting with the models, and define success to be how frequently the synthesized program is equivalent to the ground-truth program on an independent *test set* of input sequences.

The results of our evaluation are shown in Table 5.1 and Figure 5-1. While these results are subject to change as we continue to finalize our work, the Heuristic algorithm is currently most effective on this suite: It solves all but three of the benchmarks, and mostly does so in less time or very close to the time of the best of the other two algorithms (the runtime is very similar to Full Sketch's runtime on

many models). The Divide-and-Conquer Sketch algorithm is notably slower than both the Heuristic and Full Sketch algorithms on all but one of the models that all three methods solve. Further, while the vast majority of the programs synthesized by the Heuristic and Full Sketch algorithms either exactly or almost exactly match the ground-truth programs, some of the programs synthesized by the Divide-and-Conquer method do not generalize as accurately. This is a result of the fact that we do not enumerate the entire space of automata matching each individual update function before taking the product. We instead just enumerate a small, finite subset, so the computed product is often not optimal.

The most interesting result in our evaluation is the following: For seven of the benchmark programs—Counts III, Count IV, Count V, Double Count II, Coins I, Coins II, and Coins III—both Sketch-based algorithms timed out after 12 hours without producing a solution, while the Heuristic algorithm solved all those models in minutes to hours: 1.5, 2.0, 2.5, 13.7, 47.8, 163.3, and 560.3 minutes, respectively. The poor performance of the Full Sketch method on these models is due to the fact that the models' underlying latent automata have large numbers of *hidden states*, which are extra accept states for any of the state-based update functions. More precisely, every update function triggered by a latent-state-based event must have at least one accept state in the automaton. For this reason, the number of states in the automaton must be at least the number of update functions, so the Sketch solver begins its search for a satisfying automaton by searching among automata with this minimum state count. If a solution is not found, Sketch will keep incrementing this state count until it finds a correct automaton. Hence, when the actual number of states in the desired automaton is much larger than the number of update functions (e.g. of the ten states in the Coins III automaton, 8 states are hidden), the underlying SAT solver does not terminate quickly, resulting in the observed timeouts. Divide-and-Conquer Sketch fails for the same reason, because while individual-update-function-level automata may sometimes be smaller than the overall automaton, in these models, each individual automaton is actually the same as the full automaton. Hence, Sketch again does not terminate in the Divide-and-Conquer framing.

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

Finally, we emphasize that our benchmark results are still preliminary and are subject to change as we continue to modify both the Heuristic and the Sketch-based algorithms, including with the generalizations described above. Some of these modifications will affect all three algorithms' runtimes, like the previously described "multiplicity handling" generalization, while others will affect individual algorithms' runtimes. For example, optimizations to the Sketch implementations could decrease the Sketch-based algorithms' runtimes, while improvements that make the Heuristic algorithm less brittle/more general would increase the Heuristic algorithm's runtimes. More precisely, while the Heuristic algorithm works well on the current benchmark suite, the nature of it being a heuristic means that there are certainly classes of models on which it will fail, which we can patch somewhat with more intricate algorithms. These kinds of changes are likely necessary for the method to generalize both to other AUTUMN programs we may add to the benchmark suite, as well as alternative user action sequences on the current programs beyond just those inputs that we hand-curated. With respect to different user input sequences in particular, we currently have selected inputs that we knew were compatible with the heuristics embedded in our algorithm (including the state synthesis heuristics), so modifica-

tions will be necessary to ensure success on other inputs, including those generated by users without knowledge of the synthesizer. Lastly, further thinking about our evaluation design, including potentially running the Sketch solver with a few different parameter options to fend against blowup, to ensure the fairest possible comparison between the three algorithms also remains part of future work. These modifications may result in different relative runtimes across the variant algorithms than we currently observe (e.g. potentially lower Sketch runtimes and higher Heuristic runtimes on some benchmarks).

5.2 External Benchmark Suite: Preliminary Results

In addition to our manually-constructed dataset, we also wanted to evaluate our method on an externally-sourced benchmark. We elected to use the suite of grid world games developed by Tsividis et. al. as part of their work titled "Human-Level Reinforcement Learning through Theory-Based Modeling, Exploration, and Planning" [22]. In that paper, the authors designed a set of 27 distinct grid world games that each have two to five levels, making for a total of 90 games when all levels are included (Figures 5-3 and 5-4). They used these games to train a video game playing agent called EMPA (for the Exploring, Modeling, and Planning Agent), so we refer to their benchmark suite as the EMPA suite for the rest of this section. The games were written in the PyVGDL language for describing grid-world-based video games, and exhibit a number of differences from AUTUMN programs. These include that the games are all run on very large grids, such as 330 pixels by 900 pixel grids, while most Autumn programs run on 16 by 16 grids (the unit size of a cell in the visual AUTUMN grid is larger than a single pixel). In addition, they are all *games*, with a single movable player agent and specific win states and lose states, whereas AUTUMN programs

	Model Name	Input Length (Frames)	Output Length (Program Lines)	Heuristic Runtime	Sketch Runtime	D&C Sketch Runtime
	Ants	24	24	95.8	N/A	N/A
	Chase	42	33	18.0	N/A	N/A
	Magnets	53	43	69.7	N/A	N/A
	Space Invaders	42	52	70.4	N/A	N/A
	Sokoban	25	38	20.7	N/A	N/A
	Ice	27	45	2.5	N/A	N/A
Latent State	Lights	24	36	3.2	2.9	6.6
	Disease	22	31	4.6	5.3	6.8
	Grow	40	49	123.2	179.2	246.6
	Grow II	40	49	×	×	×
	Sandcastle I	32	33	11.1	9.4	14.8
	Sandcastle II	32	33	×	×	×
	Bullets	54	54	16.2	25.6	⊥
	Gravity I	19	37	3.3	2.8	5.2
	Gravity II	24	50	7.4	7.1	10.5
	Gravity III	27	83	2.1	5.4	⊥
	Gravity IV	48	54	3.6	4.0	10.7
	Count I	22	31	2.7	2.8	5.7
	Count II	39	39	2.6	3.3	12.8
	Count III	69	47	1.5	⊥	⊥
	Count IV	109	55	2.0	⊥	⊥
	Count V	149	64	2.5	⊥	⊥
	Double Count I	94	43	3.7	4.5	44.3
	Double Count II	156	59	13.7	⊥	⊥
	Wind	21	42	12.7	14.2	16.1
	Paint	27	39	6.3	6.2	18.4
	Mario	81	65	550.4	626.7	655.8
	Water Plug	42	37	429.8	432.3	250.6
	Mario II	81	65	×	×	×
	Coins I*	287	62	47.8	⊥	⊥
	Coins II*	408	69	163.3	⊥	⊥
	Coins III*	607	77	560.3	⊥	⊥

Table 5.1: Table of input/output lengths and algorithm runtimes on each of the benchmark programs. A bottom symbol indicates timeout after 12 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. In addition, 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. Finally, Coins I, II, and III are marked with an asterisk to indicate we only ran those models with the "single-cell" object parsing algorithm rather than both algorithms in turn, due to lack of time. This will also be updated in the future.

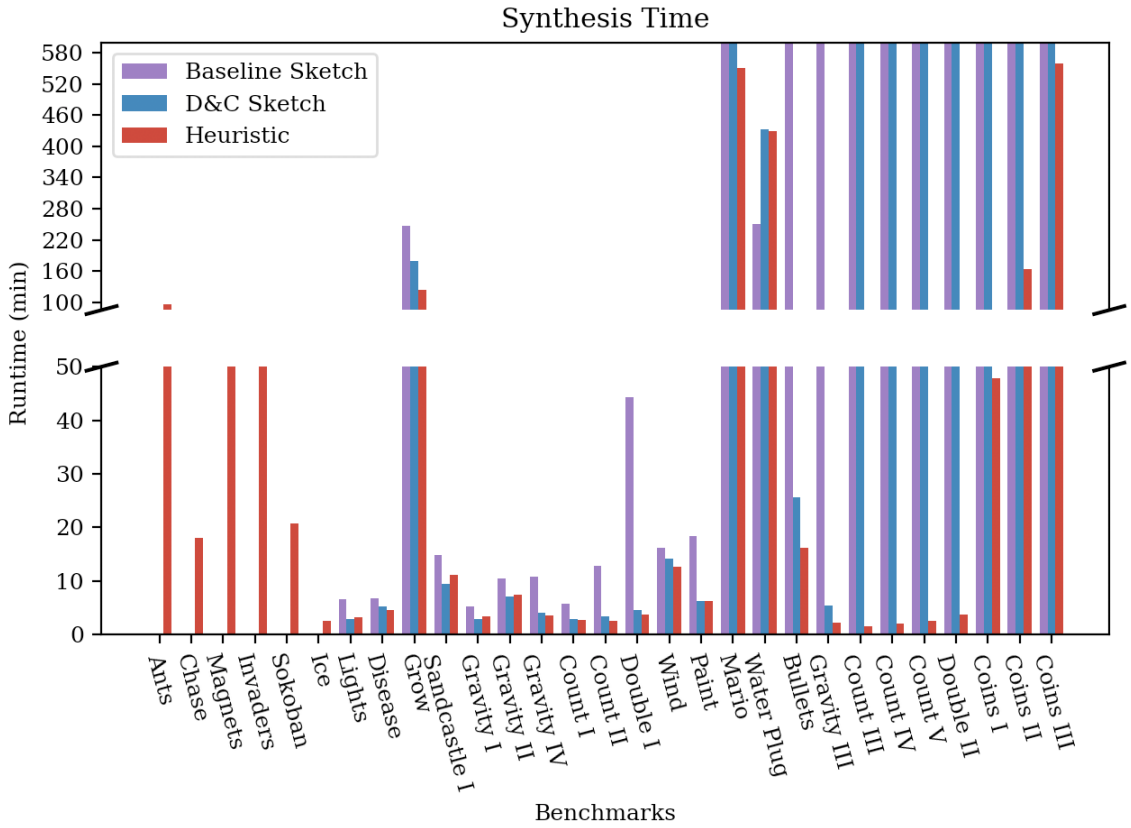


Figure 5-1: Runtimes for the variant AutumnSynth algorithms on each of the benchmark programs solved by at least one algorithm. Note that the first 6 benchmarks (Ants, Chase, Magnets, Invaders, Sokoban, and Ice) all do not contain latent state, so we currently evaluate only one of the algorithms (Heuristic) on them (see Table 5.1 caption for further explanation). We also note that the runtimes that exceed the size of the plot did not finish before the timeout, and that synthesis success is defined as producing a program that matches the observations—not necessarily being semantically equivalent to the ground-truth program. Finally, we note that while these results provide a snapshot of the current state of our project, they are subject to change as we continue to develop our variant algorithms. In particular, yet-to-be-implemented generalizations of the Heuristic method and optimizations to the Sketch-based algorithms could lead to different relative runtimes across the three algorithms (e.g. lower Sketch runtimes and higher Heuristic runtimes) for some benchmarks. See Section 5.1 for a more detailed discussion.

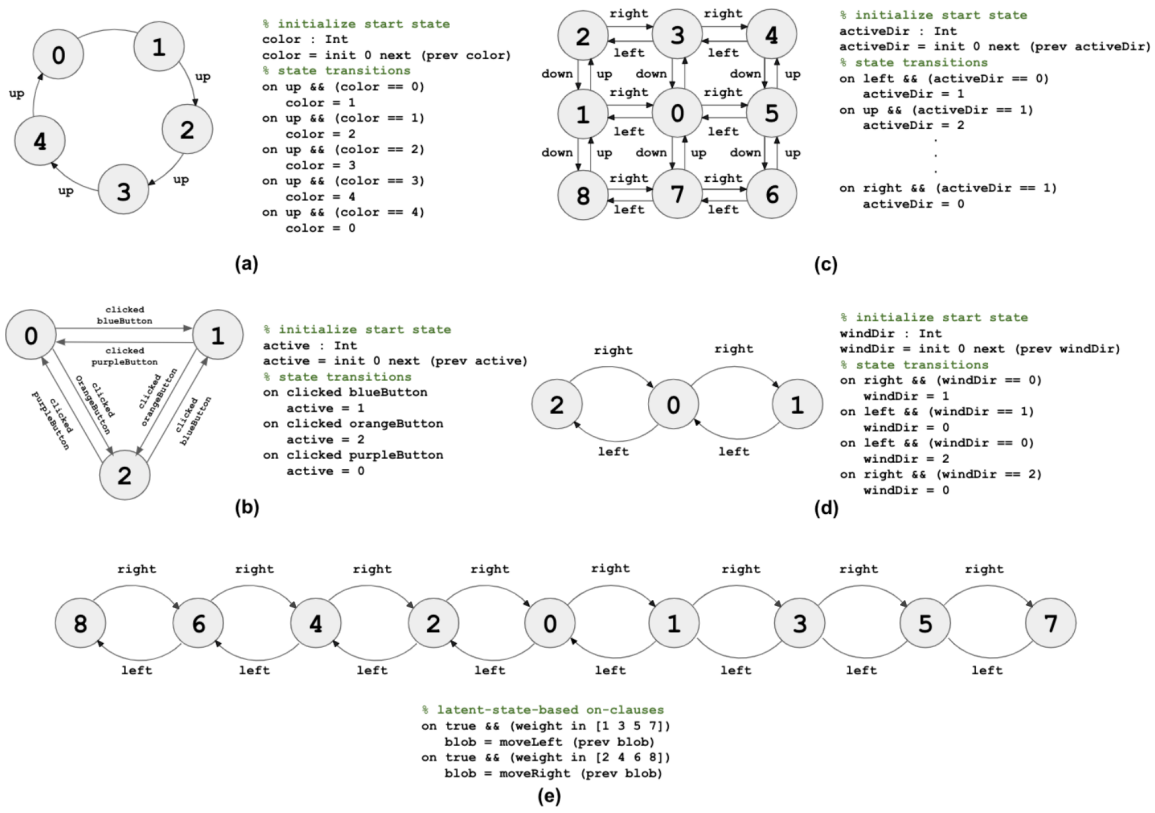


Figure 5-2: Sample latent state automata synthesized by AutumnSynth. **(a)** Paint model. Each state corresponds to a different color, indicating the color of the block added when a user clicks on an empty grid square. Pressing up cycles through the colors. **(b)** Gravity III model. Each state corresponds 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.

need not fit into this format. Finally, they also exhibit a considerable amount of *random* behavior, including dynamics such as objects being randomly added with some small probability at every time.

We have just begun to test how well the AutumnSynth algorithm can synthesize Autumn programs that model EMPA games, and take care to emphasize that our current set of results is very preliminary. In order for the algorithm to be successful in this new domain, we have modified several lower-level details of the algorithm, while maintaining the gists of the high-level phases discussed previously. These modifications include adding a few library functions to the Autumn standard library that more concisely capture some of the behaviors demonstrated in EMPA games, modifying the search space for update function synthesis to use these new library functions and removing some elements that were no longer useful, and modifying the event search spaces by adding and removing some events. We also modified the heuristic algorithms used to perform update function synthesis as well as added new heuristics to handle the kinds of random behavior displayed in the EMPA suite. Finally, since the objects in all but one of the EMPA games consist of 30 pixel by 30 pixel squares, we changed the object parsing heuristic to take advantage of this information. However, the size of these grid frames compared to the CISC grid frames means that additional improvements will be necessary to make the object parsing step sufficiently fast. As a result, for now, we skip the object parsing step in our EMPA evaluation by providing the correct objects through other means, so we can ensure that the other steps of the algorithm are operating first. We will certainly add this phase of the algorithm back in the final version of the paper.

With all of these modifications as well as some other low-level ones (e.g. some minor modifications to the algorithm between different benchmark runs that will ultimately be standardized), we ran AutumnSynth on the first level of each of the 27 games in the EMPA suite. These preliminary results are provided in Table 5.2, where certain cells are marked with *TBD* to designate that a particular experiment has not yet been performed. We find that AutumnSynth, using the heuristic state synthesis algorithm, synthesizes a program for almost all of the 27 games, with runtimes ranging

from about a few minutes to several hours depending on the size of the model (i.e. the length of the manually-constructed input sequence, the number of on-clauses that had to be learned, the number of objects involved, etc.). We note, however, that since most of these games display random behavior and thus the synthesized AUTUMN programs describe random elements, it is difficult to automatically check whether the input observation trace can be produced by the synthesized random program. As a result, we only perform this automatic check for the non-random models (8 out of 27 in the suite), and assess whether the other programs are correct just by inspection. This correctness check will certainly be revised in the final version of this paper to be more systematic, but we emphasize that the current results should thus be taken with a grain of salt: It is possible that some of the synthesized programs are less ideal models of the underlying behavior than they could be or even are incorrect, since we may have overlooked details during manual inspection (though more rigorous testing on one spot-checked random program suggests the method is working).

Nonetheless, the fact that AUTUMNSYNTH seems to work at all on programs in this externally-sourced benchmark is promising. In particular, the Heuristic algorithm was able to synthesize AUTUMN programs containing very large automata, such as that in the Aliens program, which has 14 states and 20 transitions. This is larger than any of the automata in the CISC programs, an impressive feat. Successfully synthesizing a large portion of the EMPA benchmark will concretize the generality of our approach, and we are excited about continuing to pursue this line in the future.

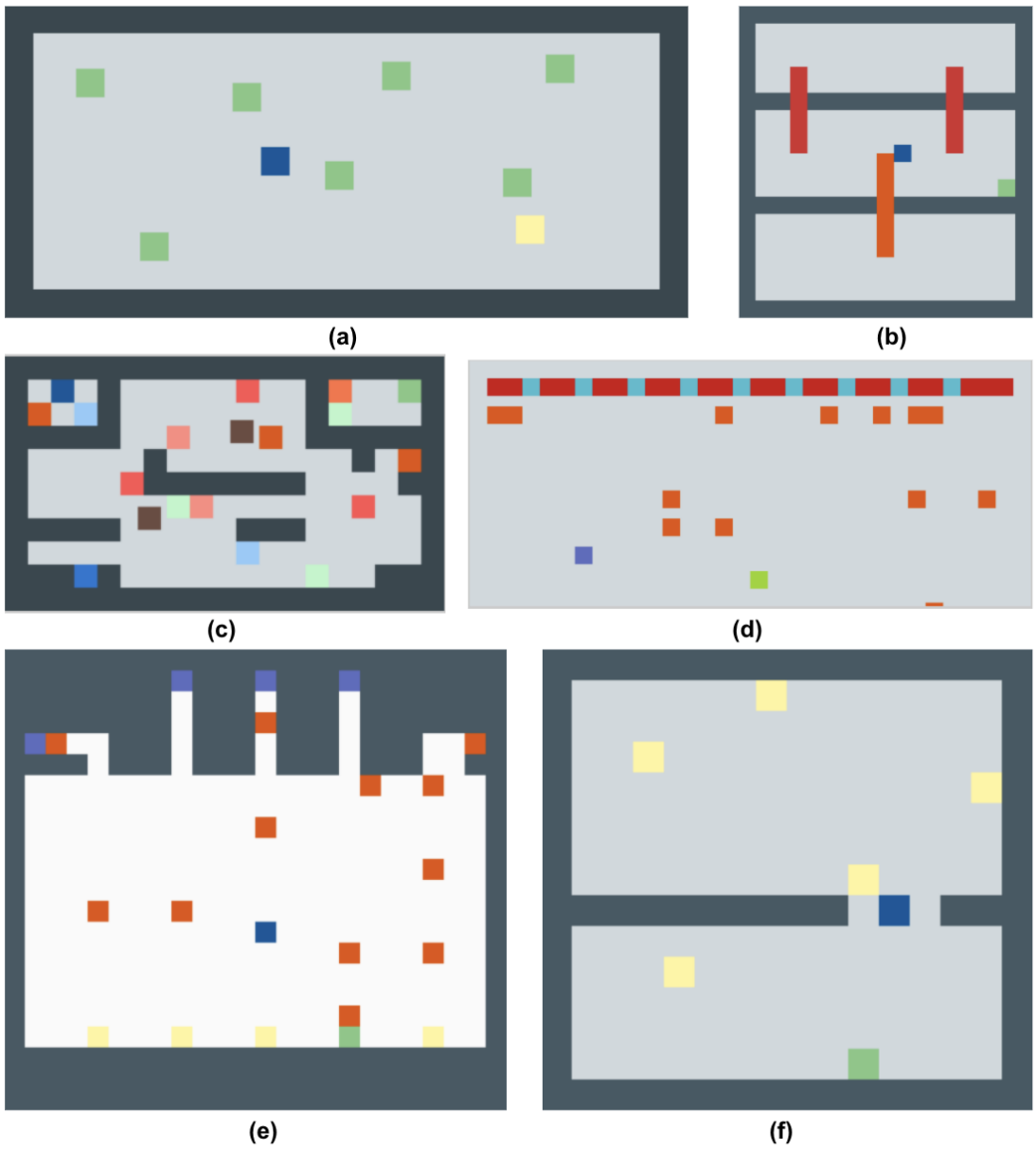


Figure 5-3: Stills from a sample of programs in the EMPA suite, resized to fit neatly into the figure. (a) Avoid George, where the dark blue agent must avoid the yellow enemy, which chases it and the randomly moving green objects. (b) Missile Command, in which the dark blue agent must get to the green goal before the gates close. (c) Portals, in which some blocks teleport the agent to other blocks. (d) My Aliens, in which the agent collects orange and is killed by purple objects. (e) Plaque Attack, in which the agent can shoot at orange enemies before they reach the yellow goals. (f) Bees and Birds, where the randomly moving yellow objects can kill the enemy before it reaches the green goal.

Figure 5-4: The Aliens program from the EMPA 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.

	ID	Model Name	# of Latent Automata	Max # of Latent Automaton States	Max # of Latent Automaton Transitions	# of On-Clauses	Input Length (Frames, Sec)	Output Length (Program Lines)	Heuristic Runtime	Sketch Runtime	D&C Sketch Runtime
No Latent State	1	**Antagonist**	0	0	0	7	186, 9.3s	50	3.8h	N/A	N/A
	2	**Avoid George**	0	0	0	9	100, 5.0s	50	4.7h	N/A	N/A
	3	Bait	0	0	0	8	86, 4.3s	50	43m	N/A	N/A
	4	Bees and Birds	0	0	0	7	30, 1.5s	43	20m	N/A	N/A
	5	Boulder Dash	0	0	0	-	-	-	⊥	N/A	N/A
	6	Butterflies	0	0	0		53, 2.7s	39	1.9h	N/A	N/A
	7	Chase	0	0	0	-	-	-	×	N/A	N/A
	8	Closing Gates	0	0	0	7	159, 8.0s	48	7.5h	N/A	N/A
	9	Explore/Exploit	0	0	0	5	222, 11.1s	36	35m	N/A	N/A
	10	Helper	0	0	0	11	290, 14.5s	51	3.4h	N/A	N/A
	11	Jaws	0	0	0	11	192, 9.6s	60	7.6h	N/A	N/A
	12	Preconditions	0	0	0	6	83, 4.1s	43	9.8m	N/A	N/A
	13	Push Boulders	0	0	0	10	211, 10.6s	58	4.3h	N/A	N/A
	14	Relational	0	0	0	11	179, 9.0s	52	2.6h	N/A	N/A
	15	Sokoban	0	0	0	7	71, 3.6s	41	42m	N/A	N/A
	16	Surprise	0	0	0	9	211, 10.6s	55	5.6h	N/A	N/A
	17	Survive Zombies	0	0	0	27	138, 6.9s	92	11.8h	N/A	N/A
	18	Water Game	0	0	0	10	57, 2.9s	57	49m	N/A	N/A
	19	Zelda	0	0	0	7	98, 4.9s	48	49m	N/A	N/A
Latent State	20	Aliens	3	14	20	37	318, 15.9s	108	28.5h	⊥	⊥
	21	Corridor	-	-	-	-	-	-	×	×	×
	22	Frogs	-	-	-	-	-	-	⊥	⊥	⊥
	23	Lemmings	3	4	12	18	356, 17.8s	78	×	×	×
	24	Missile Command	1	4	12	22	168, 8.4s	76	2.0h	TBD	TBD
	25	My Aliens	1	2	2	11	130, 6.5s	57	7.0h	TBD	TBD
	26	Plaque Attack	3	11	11	23	82, 4.1s	84	6.9h	⊥	⊥
	27	Portals	2	2	2	17	245, 12.3s	86	26.2h	TBD	TBD

Table 5.2: Preliminary results from running AutumnSynth on the EMPA 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.

Chapter 6

Conclusion and Future Work

In this thesis, we have developed a new programming language (Autumn) and benchmark suite (Causal Inductive Synthesis Corpus) for exploring few-shot learning of causal dynamics models, as well as demonstrated a proof-of-concept for a new synthesis approach (AutumnSynth) that succeeds on these problems. Our synthesis algorithm operates by uniting techniques from two disparate communities within the field of programming languages—the functional synthesis and automata synthesis communities—and widens the scope of problems that either community can solve on its own. The key innovation of our method is that symbolic latent state can be tractably synthesized by employing *error-driven search*, where the errors derive from an initial solution attempt using functional synthesis.

While our results so far are promising, we remain most excited by the many potential avenues of future work that are engendered by our current progress. These may be divided into short-term future directions and long-term future directions, each of which we discuss next in turn.

6.1 Short-Term

Beyond the previously described standardizations to our current evaluation setup, short-term extensions to our work include measuring (1) how well programs synthesized by the AutumnSynth algorithm generalize on inputs beyond the given

input sequence, and (2) how well AutumnSynth can synthesize programs from *user-generated* inputs, rather than just *expert-generated* ones.

More precisely, with respect to the former, while we currently declare a success if the synthesis algorithm produces a program that matches the input observation sequence, we are ultimately interested in how well the synthesized program matches the ground-truth program from which the input was produced. If the synthesized program is able to generalize from a relatively short observation sequence to other input sequences as well as humans can, that is strong evidence that the method is likely capturing some aspects of how humans learn these kinds of causal models. We can measure this generalization performance by creating a *test set* of observation sequences from the ground truth, and identifying the fraction of these test sequences on which the synthesized program and the ground-truth program produce the same output. In other words, we can use a traditional train/test split that is generally used in the machine learning community to evaluate our algorithm, where the split is one training sequence and $n - 1$ test sequences.

In addition, we currently select the input sequences fed to our synthesis algorithm manually, in a way that we know is compatible with the heuristics embedded in our algorithm, from the update function synthesis step to the automata synthesis step. We are interested in how well the algorithm performs when fed input traces created by users who are not as familiar with the internals of the procedure. As such, we will run user studies in which participants are asked to interact with Autumn programs via our web interface. We will record these interaction traces under two different experimental contexts: (1) a *demonstration* context, in which participants are asked to first figure out how an Autumn program works by interacting with it and then create a recorded input sequence demonstrating how it works, and (2) an *exploration* context, where the initial exploratory interactions by users are all recorded until they feel they have understood how the model works.

For both of these additional evaluation metrics (train/test split evaluation and user-generated input sequences), generalizations will need to be made to lower-level aspects of the algorithm in order to achieve high performance. These improvements

include changes like *iterating* through the set of co-occurring events for an unsolved update function in order of increasing false positive count, as opposed to always taking the top one and giving up if that choice fails as is currently done, as well as better curating the different event search spaces used in the algorithm so there is less event ambiguity in general. We will also experiment with modifications to the object parsing and mapping heuristics used by the algorithm so that they handle a wider range of scenarios, as well as generalize some details of our heuristic-driven automata synthesis algorithm. With these sorts of modifications, we are hopeful that this expanded evaluation suite will further establish the generality of our high-level approach.

6.2 Long-Term

The original dream that inspired the development of the AUTUMN domain and synthesis algorithm was to create learning algorithms that are able to infer causal models of the world as easily as humans do. The AUTUMNSYNTH method is one step in this direction, but to ultimately create such a versatile artificial learner, much work remains. One direction is determining how to best blend symbolic techniques like those developed in this thesis with deep learning-based approaches, to overcome brittlenesses that have long plagued attempts at building symbolic AI. Some examples of this thrust are applying synthesis procedures like AUTUMNSYNTH over object representations and mappings learned using deep-learning-based object detection and tracking algorithms, or using neural-guided program search to speed up some of the more enumerative aspects of the synthesis process. Another important line of future work is more formally handling uncertainty in the synthesized programs—in other words, inferring *probabilistic programs* from the observation sequences. Yet another crucial direction is developing a synthesis algorithm that produces an output program in an *online* or incremental way, for example by producing a hypothesis program after the first pair of frames and then continually *revising* that hypothesis program with each new frame until the end of the observed sequence. Such an algorithm would

more closely align with the incremental way in which humans arrive at hypotheses about how the world works over time. Progress on all of these directions and more will be needed to create a truly human-like causal model learner, which operates not just in 2D environments, but in the uncertain, messy, and 3D real world as well. We look forward to taking inspiration and lessons from work on the AUTUMN domain as we build towards this future.

Appendix A

Additional Evaluation Details

Instead of completely raw images, we currently send a slightly more processed version of the images to the synthesis engine as input, namely a list of 2D pixel positions with colors. The significance of this representation is that, if two objects overlap at one pixel, the synthesizer does not need to figure out from that pixel's color and transparency value (all AUTUMN renderings are partially transparent) that there are really two overlapping colors there. Instead, the input will already include two elements with the same x-y coordinates and color, e.g. $\{(x, y, color), (x, y, color)\}$. This detangling of pixels with overlap into their individual components can be trivially performed by storing a mapping between all RGBA values formed via overlaps of a finite number of colors, and the lists of colors that compose them. We will implement this procedure in the final version of the algorithm.

Finally, we note that the current benchmark runtimes were measured not just on one machine but across a few machines, so we will standardize our evaluation by running all experiments on one machine in the final paper. We also currently run each model 1-3 times (more for the faster benchmarks and less for the longer/timeout benchmarks), and that a few very minor bug fixes were made to the implementation between some of the experiments, which are very unlikely to affect the relative runtimes of the three algorithms. We will standardize these in the final version of this work as well (e.g. by averaging runtimes over larger number of runs, etc.).

Bibliography

[1] R. Abraham. Symbolic ltl reactive synthesis, July 2021.

[2] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3):158–243, 2021.

[3] B.J. Copeland. *Alan Turing's Electronic Brain: The Struggle to Build the ACE, the World's Fastest Computer*. OUP Oxford, 2012.

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

[5] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on machine learning models. *CoRR*, abs/1707.08945, 2017.

[6] Alison Gopnik. The scientist as child. *Philosophy of Science*, 63(4):485–514, 1996.

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

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

[9] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, January 2011.

[10] Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, and Daniel Kroening. DeepSynth: Automata Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning, 2021.

[11] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray

Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. 596:583–589, 2021.

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

[13] Xuhong Li, Haoyi Xiong, Xingjian Li, Xuanyu Wu, Xiao Zhang, Ji Liu, Jiang Bian, and Dejing Dou. Interpretable deep learning: Interpretations, interpretability, trustworthiness, and beyond. *CoRR*, abs/2103.10689, 2021.

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

[15] Gary Marcus. Deep learning: A critical appraisal, 2018.

[16] Vaishnavh Nagarajan, Anders Andreassen, and Behnam Neyshabur. Understanding the failure modes of out-of-distribution generalization. In *International Conference on Learning Representations*, 2021.

[17] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 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, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *CoRR*, abs/1911.08265, 2019.

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

[20] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[21] Rodrigo Toro Icarte, Ethan Waldie, Toryn Klassen, Rick Valenzano, Margarita Castro, and Sheila McIlraith. Learning reward machines for partially observable reinforcement learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[22] Pedro A. Tsividis, Joao Loula, Jake Burga, Nathan Foss, Andres Campero, Thomas Pouncy, Samuel J. Gershman, and Joshua B. Tenenbaum. Human-level reinforcement learning through theory-based modeling, exploration, and planning, 2021.

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