766 Project Midpoint: Automata from Images

Keith Johnson <keith.johnson@wisc.edu> [kjjohnson32]

April 5, 2022

1 Project Summary

Finite state automata underlie many foundational areas of computer science, including regular language definitions, state machines in programs, model checking, and more. Their simple visual representation makes hand-drawing an automaton on paper or a whiteboard the typical first step of implementation. However, the translation from a visual representation to a working implementation is often tedious and error-prone. This project aims to create a system that takes a hand-drawn automaton, derives logical constraints from the image, and synthesizes an implementation of the desired automaton.

Solving this problem is important because it will reduce the time and potential for bugs when implementing systems based on automata. This reduction enables software developers to focus more on the program logic and less on the minutia of implementation. I am personally interested in this problem because my area of research is in program synthesis, and an image-based program specification is novel in this area. Plus, I have at many times had to do this automaton-image-to-implementation translation manually, and it was time-consuming and error prone. Having a system as proposed would have been beneficial to me.

1.1 Related Work

Recognizing hand-drawn finite automata diagrams have seen recent work. Bresler, Průša, and Hlaváč (2016)[1] studied *on-line* recognition of flow charts

[1] Bresler, M., Průša, D. & Hlaváč, V. Online recognition of sketched arrow-connected diagrams. IJDAR 19, 253–267 (2016). https://doi.org/10.1007/s10032-016-0269-z

and finite automata, where the input is a set of line strokes from a drawing program, and it produces a structural description of the diagram. This work was extended to off-line recognition by deriving line strokes from images and applying the existing techniques for solving on-line recognition. Other approaches, such as that of Schäfer, Keuper, and Stuckenschmidt (2021)[2] use off-line recognition to directly derive diagram features from the images using deep learning. I am less interested in this, as described in Section 2.

I was not able to find prior work for this exact problem, going from an automata image all the way to a synthesized program. Similar approaches either rely on drawing the automaton on the computer (so the system is directly told all constraints and is simply rendering the automaton), or direct input of the automaton constraint. Using an image as a program specification in this manner appears to be mostly[3] novel. Existing approaches either are focused on UI design (e.g., Microsoft's Sketch2Code) or synthesizing image manipulations (e.g., using the actual image data as a specification), as opposed to specifying an executable program through diagrams in an image (e.g., extracting features and text that specify a program). For this particular case of automata, deriving a specification from an image is a natural step forward for usability.

2 Overall Approach

While the overall application is a new approach, my implementation will use existing "classical" computer vision techniques for recognition of the automaton pieces images, as well as standard program synthesis techniques for automata synthesis. Specifically, my goal is to compile the synthesis program into a *semantics-guided synthesis* (SemGuS) problem, which is a general synthesis framework and the focus of my main research. Once compiled to a SemGuS problem, any SemGuS solver can be used to synthesize the automaton implementation.

I am particularly interested in "classical" computer vision techniques (e.g., engineering specific features instead of using deep learning) due to the end result of creating a program specification, which must be exact and precise. I

[2] Schäfer, B., Keuper, M. & Stuckenschmidt, H. Arrow R-CNN for handwritten diagram recognition. IJDAR 24, 3–17 (2021). https://doi.org/10.1007/s10032-020-00361-1

[3] An honorable mention goes to the joke paper "93% of Paint Splatters are Valid Perl Programs" by McMillen and Toady (https://www.mcmillen.dev/sigbovik/2019.pdf).

expect the format of a drawn diagram to be more restricted in my system as opposed to a neural-network-based approach, but I believe the predictability and observability of my approach will increase confidence that the correct program specification is produced, as opposed to one that appears similar but is incorrect.

2.1 Challenges for Computer Vision

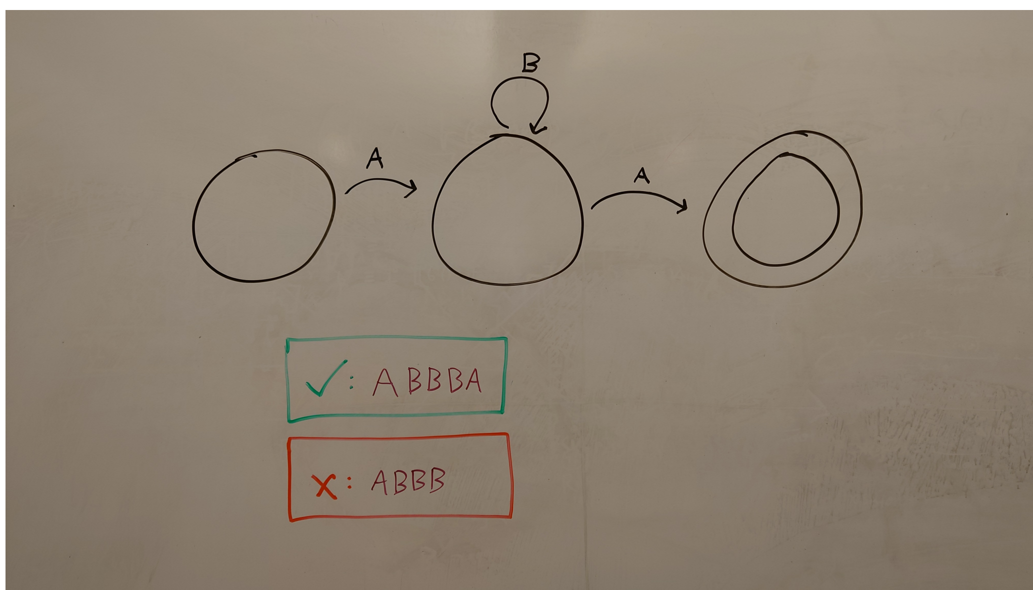

Figure 1: An example simple automaton image.

There are several challenges that this problem poses for computer vision techniques, in particular the following three:

1. The target medium, diagrams on whiteboards, is a visually noisy medium. Consider Figure 1; note the uneven lighting with splotches and stains from previous drawings. This noise necessitates robust image cleaning techniques to separate the features from the background.

2. The diagrams are hand-drawn, meaning circles are only *roughly* circular and lines are only *approximately* straight (Figure 2). Additionally, the marker strokes are uneven and include missing streaks. Together, this makes using standard techniques like Hough transforms and corner

Figure 2: Circles are approximately circular and lines are approximately straight, which proves to be a challenge.

detectors difficult at best. Much of the challenge comes from finding methods that can robustly identify the various diagram pieces despite their imperfections.

3. Diagram pieces have *implied relationships* between each other. For example, transition arrows are associated with both a start and end state, as well as a transition label above or below. These different pieces must be grouped into their logical relations with each other after being recognized.

In summary, while this problem may appear to be a straight-forward exercise in recognizing geometric shapes, the particular medium (dirty whiteboard) and content (imperfectly drawn squiggles) makes this task a challenge. The bulk of the work is handling the mess and imperfection of the real world, and cleaning the incoming information into a usable state.

3 Evaluation

Evaluation of this system will be based on the following questions.

- Can it successfully turn automata images into implementations?
- What sorts of automata can it can handle?

- Under what conditions does it fail?

Evaluation will include creating "benchmark" images of varying difficulty. As the focus of this project is on the computer vision side, the evaluation will be oriented more toward the produced automata specifications rather than the final synthesized programs. In particular, one caveat is the maturity of SemGuS tooling—it is still under active development and may not be stable enough to solve these automata problems.

4 Current Progress

I have made decent progress on this project so far, although I am behind in my timelines. In particular, the challenges mentioned in Section 2.1 have been delaying my progress, though most are solved at this point. Currently, I am successfully recognizing states (including accepting vs. regular states) and positive/negative example boxes, as shown in Figure 3. I expect my progress to accelerate now that the fundamental image cleaning and processing challenges are solved. Specific notes about the current state of my implementation are as follows.

Handling Whiteboard Noise [Complete]. I handle whiteboard noise by using adaptive thresholding to separate the bulk of the content; however, this leaves a lot of noise, specifically in the lower-right corner of Figure 1, but generally present all over. To combat this, I used a sequence of applying a Gaussian blur to the image, thresholding that blur to get a mask, and applying the mask to eliminate noise outside of the strong content.

Handling Squiggly Circles and Lines [Complete]. I spent a large chunk of time attempting to get Hough circle transforms working to detect state circles. However, I was unable to get satisfactory performance. I needed to drastically increase the transform sensitivity in order to recognize the states as circles, but this resulted in a large number of erroneous "circles" being detected across the image. I had similar problems with the rectangles, and attempting to detect corners failed due to lack of well-defined corners.

Instead, I used geometric properties of regions. Circularity measurements are good enough to detect even my egg-shaped circles, and I use a similar technique for rectangles (comparing the area of the bounding box to the actual area of the shape).

Handling Transition Arrows [In Progress]. Still working on a robust way to find arrows. Bresler, Průša, and Hlaváč (2016) have some good notes about detecting arrows, and I am also planning on trying feature detection (e.g., SIFT) against "known" arrows.

Handling Labels [Not Going Well]. I have been having difficulty getting my handwriting recognized. As it is just me doing this project, setting up and training a custom OCR pipeline is probably out-of-scope. I will likely either use feature detection to recognize only a subset of characters (e.g., '0' and '1') or switch to colored balls as the automata alphabet.

5 Revisions and Timeline

The following pieces of my original plan are likely to be revised:

- Sending automata specifications to SemGuS tooling seems dubious at this point, based on said tooling not existing yet. Generating a SemGuS problem file is likely good enough, since that completes the computer vision portion of this project.
- OCRing hand-written labels is unlikely at this point, as mentioned above. Switching to colored balls as the automata alphabet seems the most feasible and retains the "spirit" of the project.

A revised timeline is as follows:

Date	Tasks
Week of Feb. 28th:	*Initial project planning. Set up vision pipeline.*
Week of Mar. 7th:	*Item detection: Hough transforms and failures*
Week of Mar. 14th:	*Image cleaning*
Week of Mar. 21st:	*Item detection: states (and accepting states)*
Week of Mar. 28th:	*Item detection: positive and negative examples.*
Week of Apr. 4th:	Item detection: transitions and labels
Week of Apr. 11th:	Constraint compilation: transition relations.
Week of Apr. 18th:	Create benchmark images and test.
Week of Apr. 25th:	Final touches and presentation.

Figure 3: Recognized states (regular in cyan, accepting in blue) and positive (green) and negative (red) examples

6 Example

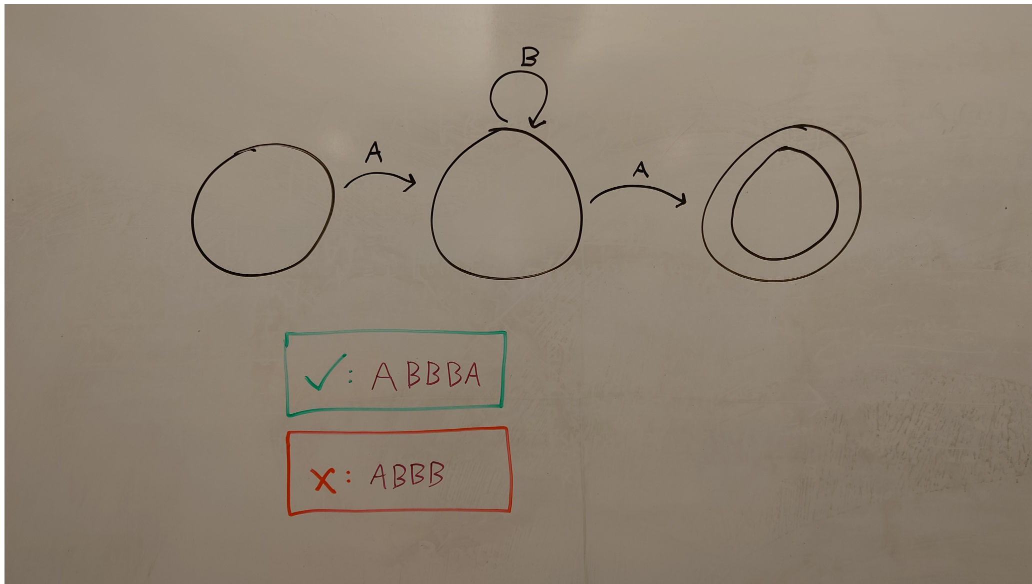

Figure 4: An example simple automaton image, implementing the regular expression AB^*A, as well as including a positive and negative example.

Figure 4 shows an image of an automaton defining the regular expression AB^*A, as well as a positive and a negative example. This image would be compiled into the following constraints, written here as constrained Horn clauses (CHCs). First, three states are defined:

$$\top \implies State(1)$$
$$\top \implies State(2)$$
$$\top \implies State(3)$$

There are additional constraints distinguishing the accepting state (double circle) and non-accepting states:

$$s = 1 \implies Reject(s)$$
$$s = 2 \implies Reject(s)$$
$$s = 3 \implies Accept(s)$$

The transitions (arrows) between states become:

$$
\begin{aligned}
s = 1 \land i = \text{“A”} \land s' = 2 &\implies Transition(s, i, s') \\
s = 2 \land i = \text{“B”} \land s' = 2 &\implies Transition(s, i, s') \\
s = 2 \land i = \text{“A”} \land s' = 3 &\implies Transition(s, i, s')
\end{aligned}
$$

The examples become additional constraints:

$$
\begin{aligned}
\mathit{Execute}(s_0, \text{“ABBBA”}, s') &\implies \mathit{Accept}(s') \\
\mathit{Execute}(s_0, \text{“ABBB”}, s') &\implies \mathit{Reject}(s')
\end{aligned}
$$

where *Execute* is a relation that holds when the automaton is run from initial state s_0 on the given string and halts at state s'. The exact form the examples will take on the image is not yet decided; the presented form of a check or cross followed by the example seems reasonable for now.

Note that these specifications are not a complete description of the system. In particular, no starting state was specified in the image, and there is an implicit failure state for when no transitions apply. While these are necessary for a real implementation, drawn automaton often omit these, and so we use program synthesis to find the most reasonable implementation (aided by the given positive and negative examples).