Gidayu: Visualizing Automaton and Their Computations

Tiago Cogumbreiro
tiago.cogumbreiro@umb.edu
University of Massachusetts Boston
Boston, Massachusetts, USA

Gregory Blike
gregory.blike001@umb.edu
University of Massachusetts Boston
Boston, Massachusetts, USA

ABSTRACT

Generating visualizations of Formal Languages and Automata (FLA) is often a laborious and error prone task. Existing tools either offer the ability to fully customize the appearance of the artifacts, or offer reusability and abstraction, but do not offer both at the same time. In this paper, we introduce a system called GIDAYU for creating mathematical diagrams of automata, of their computations, and of their transformations. Many kinds of automata are supported: (non)deterministic finite automata, generalized nondeterministic finite automata, and (non)-deterministic pushdown automata. GIDAYU fosters experimentation and rapid prototyping, as diagrams are generated automatically. The specification language includes directives to fine tune the presentation of each element; and, users can customize the visual notation used by our system. We discuss various parameters GIDAYU offers to visualize diagrams and their importance in the instruction of FLA.

CCS CONCEPTS

• **Human-centered computing → Visualization toolkits**; **Visualization design and evaluation methods**; • **Theory of computation** → Regular languages.

KEYWORDS

visualization, automata, formal languages, computation diagram

1 INTRODUCTION

The subject, Formal Languages and Automata (FLA), is in the basis of the curriculum of undergraduate computer science [6]. A crucial artifact of the education material of FLA consists of depictions of automata, known as state diagrams, and illustrations of common automata operations, *e.g.*, converting a Nondeterministic Finite Automaton (NFA) into a Deterministic Finite Automaton (DFA). Instructors face the challenge of producing multiple illustrations of theoretical concepts, where the correctness of the depiction depends on its accuracy and consistency.

Mathematical diagrams are a powerful learning tool. While there are multiple options to produce high quality visualizations of FLA, we lack systems that encourage rapid prototyping and customization. Image editors only offer low-level graphic primitives and lack the ability to specify a visual notation language. Generating correct and consistent artifacts with image editors is error prone and laborious [8]. Interactive FLA editors, such as [5, 9, 14], require the user to manually drag and drop elements (such as states and transitions) to create and relate them, which can then be used for an array of educational tasks (*e.g.*, acceptance tests). Interactive FLA editors produce mathematical diagrams with a hard-coded notation, so reusing such artifacts in an educational context can be challenging. The visual notation cannot be adapted to match the artifacts of a textbook or of the instructor's slides, which can introduce confusion for students and inconsistencies in production. Educators need to devote time to clarify the different conventions found in the artifacts produced by interactive editors.

This paper introduces the first tool to visualize the computation of automata. The state of the art has limited support to explain the computation of an automaton, rendering the execution as a series of tables. Computation diagrams, *e.g.*, [16, pp. 50], visualize all reachable runtime states (configurations) of an automaton for a particular input. Computation diagrams are useful pedagogical device to understand automata: students can visualize in one diagram the various possible configurations and the relationships among them. In such diagrams, nondeterminism is visually evident as a form of branching, and accepting an input can be explained as a search algorithm (*i.e.*, reaching an accepting configuration). Furthermore, manual generation of computation diagrams, say with an image editor, scales poorly. The upper bound of the size of a computation diagram is given by the number of states times the length of the input. Additionally, optimizing the layout of large diagrams to best fit a certain area is a nontrivial task.

We present a tool called GIDAYU[1] [3] to visualize specifications of multiple kinds of automata as well as their computations. To the best of our knowledge, our tool is the first to visualize computations. Our approach is to separate the definition of an automaton (and of a computation) from its presentation (*i.e.*, the state diagram, the computation diagram), while giving full control over the visual notation. The user can refine visual elements, by highlighting, hiding, or changing the colors and shape of any element.

Our contributions consist of using GIDAYU to:

- Section 2, on regular languages: visualizations of DFAs, NFAs, and Generalized NFAs (GNFAs); visualizations of computations; automata transformations from NFA to GNFA, and from GNA to a regular expression.
- Section 3, on context-free languages: visualizations of pushdown automata (PDAs) and their computations.
- Section 4, on customizing visualizations: styling automata and their computations, which allows for changing the appearance of diagrams.

The structure of the remainder of the paper is as follows. Section 5 reports on GIDAYU in a classroom setting. Section 6 discusses related work and Section 7 concludes.

2 VISUALIZING REGULAR LANGUAGES

[1] GIDAYU is named after a form of musical narration that accompanies Japanese puppet shows.

This paper appears in the Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1 (ITiCSE 2022).

Listing 2.1: An NFA declaration that generates Figure 1.

```
type: nfa
states:
  q1: {label: q_1, initial: true}
  q2: {label: q_2}
  q3: {label: q_3, final: true}
transitions:
- {src: q1, actions: [b], dst: q2}
- {src: q1, actions: [a], dst: q1}
- {src: q1, actions: [null,b], dst: q3}
- {src: q2, actions: [null,b], dst: q3}
- {src: q3, actions: [null], dst: q1}
```

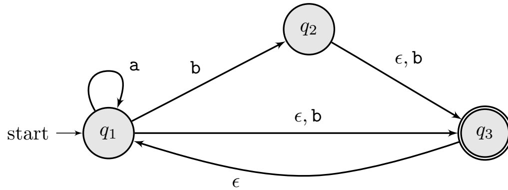

Figure 1: Gidayu's visualization of Listing 2.1.

We introduce Gidayu by showing visualizations generated from descriptions of automata. The user specifies an automaton using our domain-specific language (DSL), *e.g.*, Listing 2.1. From this, Gidayu can produce two forms of diagrams: a visualization of the automaton itself (*e.g.*, Figure 1), and a visualization of how that automaton processes an input (*e.g.*, Figure 4).

To showcase the generality of our tool, we discuss the two kinds of diagrams discussed in [16, Chapter 1: Regular Languages]: state diagrams (Section 2.1), that have specializations to DFAs, NFAs, and GNFAs; and computation diagrams (Section 2.2), that visualize the semantics of NFAs. A GNFA is an NFA where single transitions are defined on regular expressions rather than on single letters, used to show that every NFA can be represented by an equivalent regular expression [16, Lemma 1.60]. Finally, in Section 2.3, we describe how to combine automata with common operations on regular languages. Gidayu can visualize *all* diagrams and operations presented in [16, Chapter 1: Regular Languages].

2.1 State Diagrams

A state diagram depicts a finite automaton as a directed graph, where nodes represent the automaton's states, arcs represent the automaton's transitions.

NFAs. Our finite automata DSL is simple. In Listing 2.1, we find a section to declare the type of automaton (here `nfa`), its states, and transitions. The `states` section, in Lines 2 to 5, declare the identifier and attributes each state. For instance, in Line 3, we declare a state with a unique identifier `q1`. The first attribute of `q1` is a `label` which gives the human-readable name of the state (encoded as LaTeX). The attribute `initial` sets `q1` as the initial state of the automaton. The `transitions` section, in Lines 6 to 11, declares the transitions of our automaton. A transition consists of the identifier of the source state (`src`), and the identifier of the target state (`dst`), and a list of symbols (each encoded as a string) consumed. The keyword

Listing 2.2: A GNFA example.

```
type: gnfa
states:
  s: {label: s, initial: true}
  q1: {label: q_1}
  q2: {label: q_2}
  q3: {label: q_3}
  a: {label: a, final: true}
transitions:
- {src: s, dst: q1, actions: [[]]}
- {src: q1, actions:[{char: b}], dst: q2}
- {src: q1, actions:[{char: a}], dst: q1}
- {src: q1, actions:[{union: {left:[], right:{char: b}}}], dst:
    q3}
- {src: q2, actions:[{union: {left:[], right:{char: b}}}], dst:
    q3}
- {src: q3, actions:[[]], dst: q1}
- {src: q3, dst: a, actions: [ [] ] }
```

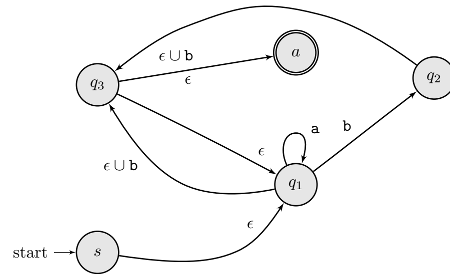

Figure 2: Gidayu's visualization of Listing 2.2.

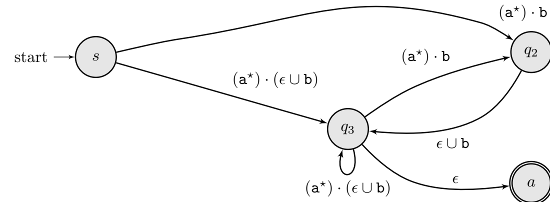

Figure 3: The visualization of the GNFA that results from automatically removing one state from Listing 2.2.

`null` denotes an ϵ-transition. For instance, in Line 9 we declare a transition from q_1 into q_3 that consumes either ϵ or `b`.

DFAs. Gidayu also visualizes DFAs. The automaton description for DFAs only introduces a new type called `"dfa"`. DFAs share the visual notation of NFAs, so we skip giving an example illustration.

GNFAs. Gidayu enforces the usual semantic constraints given to GNFA, such has having exactly one accepting state, no incoming transitions to the initial state, no outgoing transitions from the accepting state, and no parallel transitions. Gidayu typesets the regular expressions in the transitions of GNFAs. Listing 2.2 lists an example of a specification of a GNFA which Gidayu renders as Figure 2.

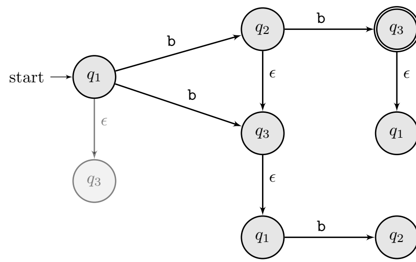

Figure 4: Computation diagram of input "bb" (Listing 2.1).

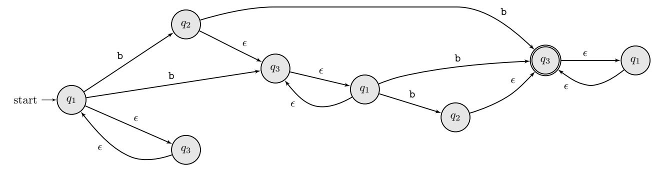

Figure 5: Computation diagram of input "bb" (Listing 2.1) that includes cycles.

2.2 Computation diagrams

Formal Background on NFA computations. Let $N = (Q, \Sigma, \delta, q_i, F)$ be an NFA, where Q is the set of states, Σ is the set of available inputs, $\delta\colon Q \times \Sigma \cup \{\epsilon\} \to \mathcal{P}(Q)$ is a transition function, q_i is the initial state, $F \subseteq Q$ is the set final states, and $\mathcal{P}(\cdot)$ is the power set. Let $w \in \Sigma^\star$ range over inputs. We can define the computations of an NFA as a transition system. Let a $C \in Q\times\Sigma^\star$ be a *configuration* of N, (q_i, w) is an *initial configuration*, for any input w, and (q, ϵ) is an *final configuration* for any $q \in Q$, where $N = (Q, \Sigma, \delta, q_i, F)$. Define the *yields* relation $\vDash$ over configurations as: $(q, cw) \vDash (r, w)$ if, and only if $r \in \delta(q, a)$; and, $(q, w) \vDash (r, w)$ if, and only if $r \in \delta(q, \epsilon)$. A *computation* is a sequence of configurations related by $\vDash$, *e.g.*, $C_1 \vDash C_2 \vDash \cdots \vDash C_n$, where C_1 is an initial configuration. Let $\vDash^\star$ denote the reflexive and transitive closure of $\vDash$. Let the set of *reachable configurations* of an NFA N processing an input w be defined as $\{C \mid (q_i, w) \vDash^\star C\}$ where $N = (Q, \Sigma, \delta, q_i, F)$.

A *computation diagram* depicts all computations that start from a given initial configuration. Gidayu can use the specification of an automaton, *e.g.*, Listing 2.1, to visualize the execution of a particular input, rendered as a computation diagram, *e.g.*, Figure 4. The nodes in the diagram are configurations and the edges represent the yields relation ($\vDash$). Observe that we omit rendering the input of each configuration, yet the input of each configuration can still be inferred by considering the character labeling each edge. For instance, in Figure 4, there are three nodes q_1, which represent from left to right the configurations (bb, q_1), (b, q_1), and (ϵ, q_1), respectively. A deterministic computation can be represented as a linked sequence of configurations (a path). A nondeterministic computation can be represented as a tree, or generally, as a directed graph. Computation diagrams help with understanding nondeterministic computations in two significant ways. First, because there may exist many ways to accept a certain input, and visually they are trivial to identify all the paths that reach certain nodes (configurations) in a graph (computation). Second, in order to show that an automaton rejects a certain input, we must show that no path reaches an accepting configuration, which is also easier to visualize than textually enumerating all possibilities.

Computations visualizes as a tree. Computations can be visualized in multiple ways. In a tree-based diagram (*e.g.*, [16, pp. 50]), the children of a node are given by the yields relation. The same configuration may appear multiple times in a tree visualization. In a graph-based diagram, the nodes are configurations and the edges are the yields relation; each configuration is a unique node of that graph. Our tool employs a graph-based diagram. For instance, in Figure 4, we observe that there are two paths that merge into configuration (q_3, b): $(q_1, bb) \vDash (q_2, b) \vDash (q_3, b)$ and $(q_1, bb) \vDash (q_3, b)$. A tree-based diagram renders configuration (q_3, b) twice. A graph-based visualization decreases the overall size of the depiction. In the worst case, each level of a tree grows quadratically with the number of states, while each level of a graph is bounded by a constant (the total number of states). Furthermore, redundancy in a computation diagram can confuse students, as they may try to find differences in two identical paths.

Visually grouping ϵ-transitions. Computations interleave processing one ϵ-transition with one non-ϵ transition. Gidayu can group each of these phases of computation automatically. In our experience, grouping ϵ-transitions helps with understanding the graph at the cost of requiring more space for the depiction. Figure 4 illustrates a grouping layout which positions vertically every configuration with the same string being consumed.

Visualizing cycles caused by ϵ-transitions. ϵ-Transitions do not consume any input, and consequently, they may introduce cycles in the computation. Gidayu allows for controlling whether or not to show cycles. Hiding cycles simplifies the understanding, as there is less information on screen, and simultaneously, the acyclic version preserves acceptance and rejection. We can compare the acyclic version (Figure 4) with the cyclic version (Figure 5).

2.3 Operations on Regular Languages

Gidayu implements multiple operations on automata:

- convert an NFA (input) into a GNFA (output), *e.g.*, Listing 2.1 yields Listing 2.2.
- remove a state from a GNFA (input) which yields another GNFA (output), *e.g.*, Listing 2.2 yields a GNFA which is then visualized as Figure 3.
- convert an NFA (input) into a DFA (output)
- given two NFAs (input) generates the union NFA, or the concatenation NFA (output)
- given one NFA (input) generates the star NFA (output)

3 VISUALIZING CONTEXT-FREE LANGUAGES

We now discuss the visualization of PDAs and their computations using Gidayu. Our tool supports both state diagrams and computation diagrams with minimal syntax for describing automata.

To assess the generality of our tool, we discuss the two kinds of diagrams discussed in [16, Chapter 2: Context-Free Languages]: parse trees to visualize the computation of grammars and state diagrams to visualize PDAs. Parse trees have no characteristic visual

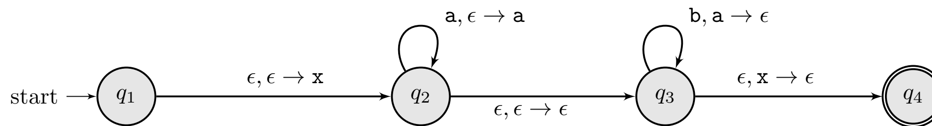

Figure 6: Visualizing the PDA in Listing 3.1. A transition $a, b \to c$ means read a, pop b and push c.

Listing 3.1: A PDA that recognizes language $\{a^n b^n\}$.

```
1  type: pda
2  states:
3    q1: {initial: true, label: q_1}
4    q2: {label: q_2}
5    q3: {label: q_3}
6    q4: {label: q_4, final: true}
7
8  transitions:
9   - {src: q1, push: x, dst: q2}
10  - {src: q2, read: a, push: a, dst: q2}
11  - {src: q2, dst: q3}
12  - {src: q3, read: b, pop: a, dst: q3}
13  - {src: q3, pop: x, dst: q4}
```

notation, so they can be rendered with general-purpose graph-drawing tools, such as GraphViz. Computation diagrams of PDAs are absent from [16, Chapter 2: Context-Free Languages].

State diagrams. Gidayu can be used to visualize nondeterministic pushdown automata (PDAs) and deterministic pushdown automata (DPDAs). In Gidayu's syntax, we must specify the type as `pda`, and the transitions now have additional, optional, parameters: a `push` symbol (given as a string), a `pop` symbol, and a `read` symbol. Our tool renders Listing 3.1 as Figure 6.

Computation diagrams. The computations of PDAs can be visualized using computation diagrams. At runtime, a PDA configuration consists of a triple with the current state, the sequence of symbols that remain to be processed, and a stack of symbols in memory. Gidayu renders the computation diagram of string `"aabb"` as Figure 7. We note that some computations of PDAs are infinite. For instance, a PDA with a self-loop of the form $q \xrightarrow{\epsilon,\epsilon \to \mathtt{x}} q$ would yield an infinite ϵ-closure, which would then produce an infinite computation when reaching state q. In practice, the problem of infinite computations does not affect the usual PDAs discussed in classes — Gidayu implements a cut-off when generating computations to cope with this problem. The problem of visualizing infinite computations remains open.

4 CUSTOMIZING VISUALIZATIONS

Gidayu lets the user change the appearance of their visualization at two different levels. Locally, the user can override the styling of a particular state/transition with *styling annotations*. Globally, the user can override the styling of all visual elements (states and transitions) by providing a *style sheet*. Configuring the appearance of state diagrams and computation diagrams is quite similar, so we restrict our discussion to state diagrams.

Styling an automaton. Listing 4.1 shows a variation of our running example Listing 2.1, where we configure the visualization to illustrate the various capabilities. A user can highlight certain states

Listing 4.1: An NFA declaration that generates Figure 8, we emphasize in olive the keywords used to fine tune states and transitions.

```
1  type: nfa
2  states:
3    q1: {label: q_1, initial: true, style: [fill=green]}
4    q2: {label: q_2, highlight: true}
5    q3: {label: q_3, final: true, hide: true}
6  transitions:
7  - {src: q1, actions:[b], dst: q2, hide: true}
8  - {src: q1, actions:[a], dst: q1, topath: [loop below]}
9  - {src: q1, actions:[null, b], dst: q3, highlight: true}
10 - {src: q2, actions:[null, b], dst: q3, style: [dotted]}
11 - {src: q3, actions:[null], dst: q1}
```

or transitions of a diagram to visually emphasize certain elements. For instance, in Line 4, we highlight state q_2 (rendered in a yellow background); in Line 9, we highlight transition $q_1 \to q_3$ (rendered with a thicker edge, in red). Users can also hide states/transitions, which by default set the opacity to 20%. For instance, in Line 5, we hide state q_3, and, in Line 7, we hide transition $q_1 \to q_2$. Gidayu allows overriding highlight/hide status through the command line without changing the source file, *e.g.*, to generate an animation where we change the state being highlighted, or to partially disclose states/transitions. Gidayu exposes the underlying graphing toolkit to change the appearance of a state/transition. For instance, in Line 3 we change the fill color of a state with the `style` annotation; in Line 10, we change the style of transition $q_2 \to q_3$ to be rendered as a sequence of dots. A user can also change the positioning of transitions, which is useful to cope with undesired decisions by the layout algorithm. In Line 8, we use the `topath` annotation to render the self-loop below the state, rather than above.

Customizing the style sheet. The default visual appearance of diagrams can be configured with a file, called the style sheet. We can configure the default styling of states and transitions, as well as the appearance of each modifier (highlighted and hidden). We can customize the colors, the line width of each element, the line style, the shape of arrows, among others. Style sheet configuration reuses the styling keywords of configuring a single automaton whenever possible. In Listing 4.2, we show an excerpt of a style sheet. Section `state` configures the appearance of states; in Line 2, we set the default background color of states to be yellow. Section `transition` configures the appearance of transitions; in Line 4, we set the default color of transitions to be blue. Section `format` configures the text of the label, which can be used to set the notation of transitions. For instance, the transitions in PDAs sometimes denoted as $a, b/c$ (rather than $a, b \to c$). The code in Lines 6 to 8 achieves this change, by ranging over each parallel transition with a for-loop and then

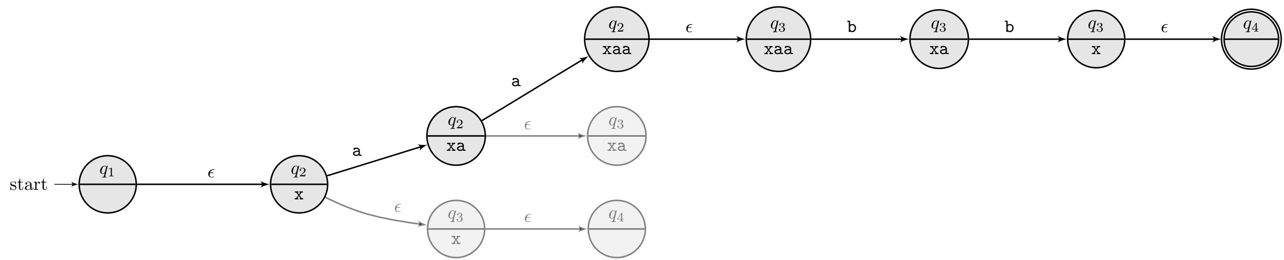

Figure 7: The PDA in Listing 3.1 processes input aabb. Depictions of a PDA configuration place the state above the horizontal line, and the stack below.

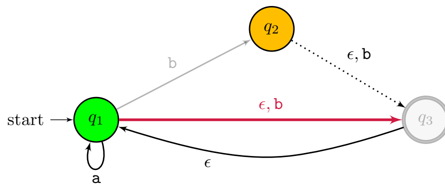

Figure 8: Gidayu's visualization of Listing 4.1.

Listing 4.2: A declaration of the visual notation for transitions (edges).

```
state:
  default: [fill=yellow]
transition:
  default: [blue]
  format: |
    {% for x in actions %}
      {{ x.read_char }},{{ x.pop_char }}/{{ x.push_char }}
    {% endfor %}
```

renders each PDA-transition as $a, b/c$. Format sections use a domain-specific language, called Jinja [11], that features a Python-inspired syntax, basic control flow structures (loops and conditionals), and common string operations. Figure 9 visualizes the PDA specified in Listing 3.1 using an alternative style sheet that resembles the popular JFLAP tool. Notice that besides changing the default colors of states and transitions, we changed the appearance of the initial state and the shape of the arrow tips. Our style sheet file allows more control over the appearance with extensions written in LaTeX, which is the underlying imaging toolkit that Gidayu uses.

5 GIDAYU IN THE CLASSROOM

In this section, we discuss using Gidayu in a classroom environment, we discuss how computation diagrams can be used to discuss visual proofs, and we discuss the interoperability of our tool.

Pedagogy. We have been using Gidayu for 3 semesters at University of Massachusetts Boston. Students generally find state diagrams intuitive. The following quotes come from informal discussions with students and should be regarded as anecdotal evidence. Some students found computation diagrams surprising, as they were unsure about the semantics of NFAs: "at first [I was] confused trying to understand how the automata 'knew' which path to take such that it would lead to an end state." Computation diagrams helped students better understand the semantics, by gradually disclosing the computation diagram: "I definitely like seeing a progression, with nodes highlighted, as this would serve to clarify the ambiguity." To achieve this, we show the computation diagram and state diagrams side-by side. We then visualize how the computation unfolds by showing/hiding configurations in the computation diagram and highlighting the state diagram. These multiple visualizations can all be achieved from the same automaton, and using the command line options to highlight/hide states/transitions/configurations.

Computation diagrams as visual proof. The first author has used computation diagrams in class to introduce the notion of a *visual proof*, and to showcase the difference between the existential and universal quantifiers. We instruct students that a computation diagram serves as a visual proof of input acceptance and of input rejection, as it lists exhaustively all possible computations for a given input. Verifying the correctness of a proof consists of verifying the correctness of the computation diagram. When proving acceptance, we require the student to identify at least one final configuration in the computation diagram. When proving rejection, we require the student to state that there are no final configurations in the diagram. We have also used computation diagrams to serve as an education device to distinguish existential quantification (acceptance) from universal quantification (rejection). Hannah and Sidoli survey using visual representations as proofs in [4].

Interoperability. The syntax of our DSL to specify automata (and style sheets) can be given in either JSON or YAML (all of the examples in this paper are written in YAML). JSON and YAML are popular interchange file formats, which encourages third-party integration of our visualizations.

6 RELATED WORK

Table 1 summarizes this section, divided in terms of its user interface. In this table, we compare each work in terms of whether there is support for a customizable notation/visualization, and for visualizing an automaton and its computations.

Graphical User Interface (GUI). We discuss related work that offer a GUI to visualize FLA. JFLAP [5, 14] and OpenFLAP [9] are widely used in education as a workbench to visualize automata, and to exercise automaton inputs and its transformations. Users cannot customize the visual notation of state diagrams and there is no support for visualizing computation diagrams. JFLAP renders

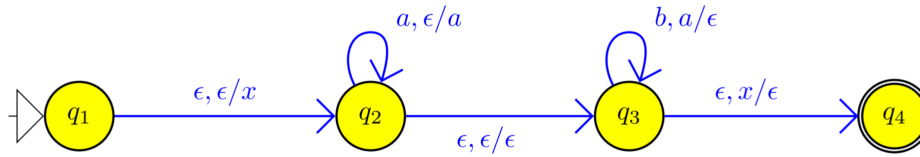

Figure 9: Gidayu's visualization of Listing 3.1 using a different style sheet.

Table 1: Feature comparison, where G represents Gidayu, "Custom. viz." means customizable visualization, "Comp. diagram" means computation diagram.

	[14]	[9]	[1]	[15]	[7]	[19]	G
UI	GUI	GUI	GUI	API	API	DSL	DSL
Custom. viz.	✗	✗	✓	✗	✗	✓	✓
State diagram	✓	✓	✓	✓	✓	✗	✓
Computation	✓	✗	✗	✓	✓	✗	✓
Comp. diagram	✗	✗	✗	✗	✗	✗	✓

computations textually: the user can step through a series of tables of configurations, one table per letter of the input. There is no way to visualize the relationship between all configurations, as computation diagram provide. GUItar [1] provides a GUI to interactively draw state diagrams and export the automaton to multiple well known file formats. This tool allows customizing the visual notation of state diagrams, but lacks support for testing inputs and visualizing computations.

Application Programming Interface (API). There are many works for symbolic manipulation of FLA with an API [2, 10, 12, 13, 17, 18]. Here, we detail related work to visualize FLA with an API. PyFormlang [15] provides an API in Python to study FLA. Users cannot customize the visual notation of state diagrams and there is no support for computation diagrams. PyFormlang lacks support for computations; it only performs membership checks (*i.e.*, whether or not an input is accepted). visual-automata [7] provides an API in Python to study FLA. Users cannot customize the visual notation of state diagrams. visual-automata renders computations textually as a table and extends state diagrams. A state diagram is annotated with additional edges: whenever one configuration yields another configuration, draw an edge from the state of the source configuration to the state of the target configuration. For instance, in Figure 4, we have the following steps $(b, q_2) \vDash (\epsilon, q_3)$ and $(b, q_2) \vDash (b, q_3)$—a step is a pair of reachable configurations. visual-automata would draw two edges between state q_2 and state q_3. Such an approach does not scale with the computation size, since any two configurations that refer to the same pair of states, adds another parallel edge to the state diagram. Additionally, the information about the string being processed is lost. In contrast, the processed inputs are readily available in Figure 4. This leaves the student to explore the progression of string recognition mentally, rather than visually, as the computational unfolds.

Language-based Interface. In closing, we cover related work that offers a DSL to visualize diagrams. Penrose [19] is a general system for creating mathematical diagrams. Gidayu and Penrose follow the same approach of separating content from presentation with a language-based interface. Penrose does offer primitives to specify the visual notation of a certain diagram, but does not support FLA artifacts, such as state diagrams or computation diagrams.

7 CONCLUSION

We introduce Gidayu, a system to visualize automata, computations, and automata transformation. We discuss how to use Gidayu to visualize regular languages (via DFA, NFA, or GNFA) and context-free languages (via PDA or DPDA). To show case the generality of our tool, we show that our tool supports most diagrams presented in a well known textbook on FLA [16]. Gidayu allows configuring the visualization of an automata, the color, shape, and notation of each element. Our tool also includes a small DSL to configure the default appearance of the various diagrams. Our DSLs are based on YAML, a well known data-serialization language, to facilitate integration with 3rd-party tools and batching.

Future work includes supporting more kinds of automata and their computations. We would like to support more kinds of automata (*e.g.*, Mealy machines, Moore machines, and Turing machines). Many of these kinds of automata can already be rendered, by encoding them as an NFA. However, such an encoding does not support visualizing computations. We would also like to explore techniques for visualizing large computations. A possible direction is to support interactive sampling of computations. Visualizing infinite PDA computations remains an interesting theoretical challenge, but in our experience such computations are rarely studied in the context of undergraduate FLA.

REFERENCES

[1] André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. 2009. FAdo and GUItar: Tools for Automata Manipulation and Visualization. In *Proceedings of CIAA*, Sebastian Maneth (Ed.). Springer, 65–74. https://doi.org/10.1007/978-3-642-02979-0_10

[2] J. M. Champarnaud and G. Hansel. 1991. AUTOMATE: a Computing Package for Automata and Finite Semigroups. *Journal of Symbolic Computation* 12, 2 (1991), 197–220. https://doi.org/10.1016/S0747-7171(08)80125-3

[3] Tiago Cogumbreiro and Greg Blike. 2022. *Gidayu source code snapshot.* https://doi.org/10.6084/m9.figshare.19660689.v1 Project repository available at: https://gitlab.com/umb-svl/gidayu.

[4] Gila Hanna and Nathan Sidoli. 2007. Visualisation and proof: a brief survey of philosophical perspectives. *ZDM Mathematics Education* 39, 1 (2007), 73–78. https://doi.org/10.1007/s11858-006-0005-0

[5] Ted Hung and Susan H. Rodger. 2000. Increasing Visualization and Interaction in the Automata Theory Course. In *Proceedings of SIGCSE*. ACM, 6–10. https://doi.org/10.1145/330908.331800

[6] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.* ACM. https://doi.org/10.1145/2534860

[7] Lewi Lie Uberg. 2021. *Visual Automata.* https://github.com/lewiuberg/visual-automata

[8] Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. 2020. How Domain Experts Create Conceptual Diagrams and Implications for

Tool Design. In *Proceedings of CHI*. ACM, 1–14. https://doi.org/10.1145/3313831.3376253

[9] Mostafa Mohammed, Clifford A. Shaffer, and Susan H. Rodger. 2021. Teaching Formal Languages with Visualizations and Auto-Graded Exercises. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 569–575. https://doi.org/10.1145/3408877.3432398

[10] Anders Møller. 2021. dk.brics.automaton: Finite-State Automata and Regular Expressions for Java. `http://www.brics.dk/automaton/`.

[11] The Pallets organization. 2021. Jinja: A a full featured template engine for Python. https://web.archive.org/web/20211206222815/https://palletsprojects.com/p/jinja/. Accessed December.

[12] Darrell Raymond and Derick Wood. 1994. Grail: A C++ Library for Automata and Expressions. *Journal of Symbolic Computation* 17, 4 (1994), 341–350. https://doi.org/10.1006/jsco.1994.1023

[13] Michael Riley, Cyril Allauzen, and Martin Jansche. 2009. OpenFst: An Open-Source, Weighted Finite-State Transducer Library and Its Applications to Speech and Language. In *Proceedings of NAACL-Tutorials*. ACL, USA, 9–10.

[14] Susan H. Rodger, Bart Bressler, Thomas Finley, and Stephen Reading. 2006. Turning Automata Theory into a Hands-on Course. In *Proceedings of SIGCSE*. ACM, 379–383. https://doi.org/10.1145/1121341.1121459

[15] Julien Romero. 2021. PyFormlang: An Educational Library for Formal Language Manipulation. In *Proceedings of SIGCSE*. ACM, 576–582. https://doi.org/10.1145/3408877.3432464

[16] Michael Sipser. 2012. *Introduction to the theory of computation*. Cengage Learning.

[17] Alley Stoughton. 2008. Experimenting with Formal Languages Using Forlan. In *Proceedings of FDPE*. ACM, 41–50. https://doi.org/10.1145/1411260.1411267

[18] Margus Veanes and Nikolaj Bjørner. 2012. Symbolic Automata: The Toolkit. In *Proceedings of TACAS*, Cormac Flanagan and Barbara König (Eds.). Springer, 472–477. https://doi.org/10.1007/978-3-642-28756-5_33

[19] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: From Mathematical Notation to Beautiful Diagrams. *ACM Transactions on Graphics* 39, 4, Article 144 (2020), 16 pages. https://doi.org/10.1145/3386569.3392375