# FSM Builder: A Tool for Writing Autograded Finite Automata Questions


Eliot Wong Robson
University of Illinois
Urbana-Champaign
Urbana, IL, USA
erobson2@illinois.edu

Sam Ruggerio
University of Illinois
Urbana-Champaign
Urbana, IL, USA
samuelr6@illinois.edu

Jeff Erickson
University of Illinois
Urbana-Champaign
Urbana, IL, USA
jeffe@illinois.edu



## Abstract

Deterministic and nondeterministic finite automata (DFAs and NFAs) are abstract models of computation commonly taught in introductory computing theory courses. These models have important applications (such as fast regular expression matching), and are used to introduce formal language theory. Undergraduate students often struggle with understanding these models at first, due to the level of abstraction. As a result, various pedagogical tools have been developed to allow students to practice with these models.

We introduce the FSM Builder, a new pedagogical tool enabling students to practice constructing DFAs and NFAs with a graphical editor, giving personalized feedback and partial credit. The algorithms used for generating these are heavily inspired by previous works. The key advantages to its competitors are greater flexibility and scalability. This is because the FSM Builder is implemented using efficient algorithms from an open source package, allowing for easy extension and question creation.

We discuss the implementation of the tool, how it stands out from previous tools, and takeaways from experiences of using the tool in multiple large courses. Survey results indicate the interface and feedback provided by the tool were useful to students.



## CCS Concepts

**• Theory of computation → Regular languages**; **• Social and professional topics → Computing education**; **• Applied computing → Computer-assisted instruction**.

## Keywords

discrete mathematics; theory education; autograder; finite automata

**ACM Reference Format:**
Eliot Wong Robson, Sam Ruggerio, and Jeff Erickson. 2024. FSM Builder: A Tool for Writing Autograded Finite Automata Questions. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653599




*ITiCSE 2024, July 8–10, 2024, Milan, Italy*



ACM ISBN 979-8-4007-0600-4/24/07…$15.00
https://doi.org/10.1145/3649217.3653599


## 1 Introduction

Formal languages and automata are a core topic in introductory computing theory courses [10]. Undergraduate students often struggle with these concepts, as these courses are usually the first exposure these students have to abstract models of computation. The simplest of these models are Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA), which are used to define regular languages. DFAs and NFAs are classified as *finite-state machines* (FSMs), or (equivalently) finite automata.

Many assignments require students to design FSMs to match English descriptions of regular languages or regular expressions. In our experience, students have frequently asked for more of these types of practice problems, along with feedback to their solutions. In a large introductory theory course, providing meaningful one-on-one feedback for a multitude of practice problems is infeasible, and the time needed to provide this feedback prevents many students from engaging fully with these topics. Thus, providing practice with automated feedback represents a significant opportunity to increase student engagement while also easing the workload of course staff.

To meet these demands, we developed the FSM Builder as a successor to previous finite automata tools, implementing similar key features while also providing functionality to aid in development of new exercises at scale for large courses. This tool was developed for easy integration with the PrairieLearn platform [20], but the implementation is self-contained and may be used elsewhere.

## 1.1 Finite Automata Background

We briefly review standard terminology for FSMs [15]. A finite state machine consists of an input alphabet (the characters that can be used in input strings), a set of states, a set of accepting states, a start state, and a transition function. The transition function defines a state to transition to from every state and every character from the alphabet (unless explicitly stated that missing characters go to an implicit dump state, which by convention results in rejection). The set of accepted strings is the language accepted by the FSM, and there may be multiple different FSMs that accept the same language.

Of note is that DFAs cannot have multiple transitions leaving a given state on the same character, while NFAs can (as this is how nondeterminism is incorporated). Additionally, NFAs may contain epsilon transitions.

## 1.2 Organization

The rest of the paper is organized as follows. We first review the features and limitations of other tools in Section 2. We outline design considerations for the FSM Builder in Section 3 and provide background for the course and development of the tool in Section 4. In Sections 5 and 6, we discuss the features of the user interface and backend autograder respectively, including the specific algorithms used to generate partial credit and feedback. In Section 7 we present survey results from using the FSM Builder in an undergraduate algorithms course with over 300 students. Section 8 provides details on how to adopt the tool, and Section 9 concludes with a discussion of future directions.

## 2 Related Work

There has been substantial prior work on software tools enabling automated teaching of automata theory, with a specific focus on finite automata [5]. In this section, we discuss the strengths and weaknesses of previous tools and compare to those of the FSM Builder.

### 2.1 JFLAP

One of the earliest such tools is the JFLAP software package [13, 14], which allows interactive exploration of automata by students, including the simulation of input strings. The original aim of JFLAP was to give students an interactive way to explore course content, as the original package does not provide mechanisms for feedback or evaluation by instructors. Moreover, the original package required a local installation, making scaling to a large course difficult. JFLAP has support for more types of automata than just finite automata [4], which is beyond the scope of the FSM Builder.

*2.1.1 DAVID Extension* A follow-up work on JFLAP analyzed the impact of automated feedback on the student experience. Specifically, Bezáková et al. [3] developed the DAVID extension, which provides automated feedback through counterexamples, and analyzed the effect on student performance. This feedback, called a *witness string*, is the shortest string where the student submission and reference solution have differing behavior. The feedback mechanism in the FSM Builder is an extension of this idea, providing multiple such strings if possible.

Although the DAVID extension used an autograder in addition to the feedback provided, this autograder did not provide partial credit.

*2.1.2 OpenFLAP* More recently, the functionality in JFLAP was expanded with an autograder in OpenFLAP [11] as part of an effort to develop an eTextbook for automata theory. This tool can be integrated with existing learning management systems, but cannot be integrated with PrairieLearn easily. In addition, the autograding algorithm is not as robust as that of the DAVID extension, using test strings instead of analyzing the FSMs for equivalence.

### 2.2 Automata Tutor

More recently, Automata Tutor has emerged as a similar tool to JFLAP, providing a web-based interface with a greater emphasis on graded assessments [7]. In particular, the most recent version of Automata Tutor provides automated feedback for a number of different question types related to automata (not just FSMs), with a focus on large courses [6]. However, this tool cannot be integrated with PrairieLearn easily, and does not seem to allow for custom grader code the way the FSM Builder does.

### 2.3 FSM Designer

The FSM Designer [18] is a graphical user interface for typesetting automata and does not feature any kind of simulation or autograding. Although the FSM designer has a narrower scope than other tools, it has an intuitive interface, freely available source code, and was already a popular tool many students were familiar with. The user interface of the FSM Builder is based on that of the FSM Designer to facilitate easy adoption by students, and encourage students to typeset their homework by giving them familiarity with the FSM Designer.

### 2.4 Partial Credit

Alur et al. [2] analyzed different partial credit schemes for DFA construction questions. Although not directly compared to other partial credit schemes, the density difference partial credit scheme proved to be the easiest to integrate into the FSM Builder.

## 3 Design Considerations

Some desirable features of the previously discussed tools incorporated into the FSM Builder are the following:

(1) A simple, modern graphical user interface for students. See Section 5.1.
(2) A robust grading algorithm that can efficiently check whether a student submission is equivalent to a reference solution and give partial credit. See Sections 6.2 and 6.3.
(3) A string-based feedback mechanism that can generate counterexamples to student submissions. See Section 6.4.
(4) The ability to create questions and practice assignments.

To distinguish itself from other tools, the FSM Builder incorporates the following unique features:

(1) Compatibility with PrairieLearn, the course content hosting platform.
(2) Reinforcement of course conventions for designing more human-readable FSMs. See Section 6.1.
(3) Fast creation of practice problems from existing reference solutions with minimal developer overhead (no need for custom grader code if using the included autograder). See Section 5.2.
(4) Use of the graphical interface independently of the autograder (for questions with custom grading algorithms).
(5) A self-contained, modular, open source implementation in Python and JavaScript. Importantly, all code from the FSM Builder can be integrated into other content hosting platforms if desired.

The above lend to the scalability and flexibility of the FSM Builder, which are the key characteristics distinguishing it from prior work.

## 4 Course Context

The FSM Builder was developed and pilot tested as part of the introductory algorithms and models of computation course in the computer science department at the University of Illinois Urbana-Champaign. Most students are second year computer science majors taking the course as a degree requirement. This is a large course, with enrollment between 370 and 400 students each semester. The models of computation portion of the course spends a considerable amount of time on finite state machines and regular languages. The first author of this paper has been the lead developer for the course since Fall 2021, and the second author has been a developer since Spring 2022. The third author has been a lead instructor for the course and written much of the course material.

One of the most common questions asked of students on finite automata is, given an English description of a language, prove this language is regular by providing a finite state machine accepting this language. This question structure is commonly used in homework and exam questions throughout the course, and accordingly, most of the existing practice questions for regular languages follow this structure. We designed the interface for question writing to facilitate efficient creation from existing solutions to these types of problems.

The course also uses the PrairieLearn platform [19] to deliver other types of course content, so any tool used needed to be compatible with this platform. See [8] for more details on content development for our course.

### 4.1 Tool Development

Prior to the development of the FSM Builder, our course used a system for autograding FSMs that required programmatic input from both students and question writers, hosted on PrairieLearn. Students were required to write a short Python program defining a finite automata, and this was checked against the reference solution using an equality checking algorithm based on the product construction for DFAs [15]. During these semesters, a common sentiment was the desire for a visual editor that did not require writing Python code.

The key motivation for the creation of the FSM Builder, rather than using existing tools, was desire for integration with PrairieLearn. With the design considerations outlined in Section 3, during the summer of 2022, we wrote the first proper version of the FSM Builder. This included the full graphical interface based on the FSM designer, a simple interface for question writers, and string based feedback.

In summer of 2023, we updated the FSM Builder with another round of feedback from course instructors, incorporating the language-similarity based partial credit scheme, and more detailed feedback for incorrect answers by students. With these new features in place, the tool has seen adoption in other courses, including the introductory discrete mathematics course within the same department.

## 5 User Interface

The FSM Builder is integrated within PrairieLearn, providing an intuitive user experience for students and an interface for question writers that requires minimal knowledge of automata theory. Of particular importance is the ease of question creation using the included autograder, as this facilitated the production of a large number of exercises in a short period of time from existing reference solutions.

### 5.1 Student Interface

As alluded to in Section 2.3, the interface for students is based on the FSM Designer by Wallace [18]. Students are given a canvas where they can add new states and transitions to their finite state machine by clicking. States can be moved once created and transitions between states can be repositioned to keep the drawing clear. States can be given labels consisting of any characters, and transitions can be labeled with characters from the language's alphabet. An instance of the FSM Builder is shown in Figure 1a.

Students receive automated feedback from their response both on the canvas and feedback box below once they select "Save & Grade".

### 5.2 Instructor Interface

In PrairieLearn, an instructor can create a new question using the custom HTML element for the FSM Builder. This HTML element takes a JSON object defining a DFA or NFA used as the correct answer. This JSON object contains the states, alphabet, transitions, initial state, and accepting states. The following is an example JSON object used for the reference solution in Figure 1:

```
{
  "states": ["0", "1", "2", "3"],
  "input_symbols": ["0", "1"],
  "transitions":{
      "0":{"0": "1", "1": "0"},
      "1":{"0": "2", "1": "1"},
      "2":{"0": "3", "1": "2"},
      "3":{"0": "3", "1": "3"}
  },
  "initial_state": "0",
  "final_states": ["3"]
}
```

Instructors can either write JSON directly when creating new questions, or use the FSM Builder itself to generate the JSON for the reference FSM. Unlike the previous solution we had for building questions with Python, there is no scripting necessary to use the autograder, only static HTML elements. For general information on custom elements, see [17].

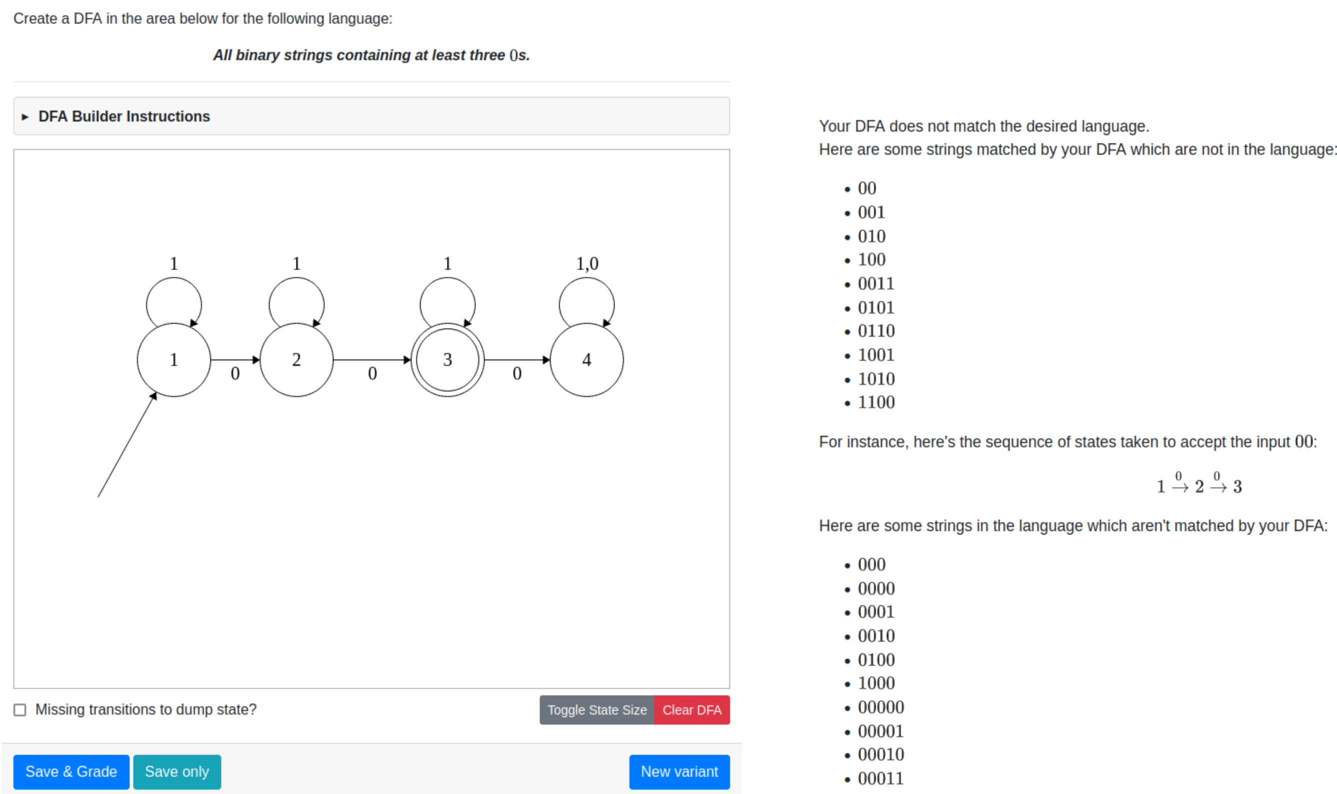
## 6 Autograder

The FSM Builder uses autograding features that both reinforce clear writing conventions for designing FSMs, and check for correctness against a reference solution using finite automata based algorithms. Note that the autograder is tailored to the most common question format used by our course, asking students to provide an FSM that accepts a given target language.

### 6.1 Enforcing FSM conventions

Before any in-depth feedback on the language of an input automaton is given by the grading algorithm, the student submissions are checked that they define a valid automaton (DFA or NFA) according to the following conventions:

(1) All provided state names are nonempty and unique.
(2) Exactly one start state is marked.
(3) The FSM does not have any non-accessible states.
(4) All transitions are defined on valid characters in the alphabet.
(5) The automaton must have transitions leaving every state for every character, unless specified that missing transitions go to a dump state.




**(a) The student interface for the FSM Builder. Note the student submission in the figure does not match the desired language.**

**(b) Feedback shown to the student for the submission on the left.**

**Figure 1: An incorrect student submission in the FSM Builder and corresponding feedback.**

(6) For a DFA, the submission should not have multiple transitions leaving a given state on the same character.

Using these checks ensures that the FSM is well defined, and helps promote clear writing. Where possible, parts of the submissions that must be corrected according to the above rules are highlighted in red on the canvas. This gives personalized feedback on specific errors in a submission, similar to what students might get from course staff.

## 6.2 Checking correctness of an automaton

Once the student submission has been validated based on the above criteria, it is converted by the grader code into a Python object representing the automaton. The grader code creates a similar object for the automaton provided as a reference solution, then compares them for language equality to award full credit. Note that this type of comparison is robust to different underlying FSMs, meaning that any student submission accepting the same language as the reference solution will be given full credit by the autograder.

This comparison is done with an optimized version of the Hopcroft-Karp algorithm [1, 12]. Importantly, this algorithm has nearly-linear runtime in the size of the input, and is easy to implement using standard data structures, making it very well-suited for this application. The runtime was of particular importance, as the size of student submissions necessitated the use of an algorithm more efficient than the standard product construction algorithm [15]. This is the same comparison algorithm used by Bezáková et al. [3].

## 6.3 Partial Credit

If the student submission was not awarded full credit by the equivalence algorithm, we use the language-based partial credit scheme described by Alur et al. [2, Section 3.3], called the *approximated density difference*. In detail, for two regular languages $L_1, L_2 \subset \Sigma^*$, this quantity is defined as

$$\text{A-}\textsc{Den-Dif}(L_1, L_2) := \frac{1}{2k+1} \sum_{n=0}^{2k} \frac{|(L_1 \oplus L_2) \cap \Sigma^n|}{\max(|L_2 \cap \Sigma^n|, 1)}$$

where $k$ is the number of states in the minimal DFA representing $L_2$. In our grading algorithm, $L_2$ is the language of the reference solution. The expression contained in the sum is the number of strings of length $n$ misclassified by $L_1$ scaled by the number of strings of length $n$ accepted by $L_2$. Intuitively, this is the size of the discrepancy between $L_1$ and $L_2$ for a given length $n$, which is then summed over all lengths up to $2k$.

This partial credit algorithm was chosen because it was practical to implement efficiently using the primitives provided by the automata package [9], and conclusions by Alur et al. [2] stating that this algorithm performed well in cases where a student answer misclassified a small, finite number of strings (for example, only misclassifying the empty string).

### 6.4 Feedback

In addition to the partial credit scheme, the grading algorithm also generates feedback from strings misclassified by the student's FSM. All strings up to length 8 are checked whether they were incorrectly accepted or incorrectly rejected by the student's FSM, and then the (lexicographically) first 10 are given as feedback to the student. If no misclassified strings are found during this search, then a minimal length misclassified string is generated using an extension of the DFA equality algorithm [12]. For the first string incorrectly accepted by the student's FSM, the sequence of states taken to reach an accepting state is shown. An example of this feedback is shown in Figure 1b.

This can be viewed as an expanded version of the feedback system used by the DAVID extension [3], where instead of a single string witnessing that the student's submission is not equal to the reference solution, we give multiple such witness strings as feedback if possible. This expanded feedback allows students to identify patterns in misclassified strings that can more quickly lead to finding a correct solution.

### 6.5 Implementation

The autograder is written in Python and uses the automata package [9] to implement to the main grading and feedback algorithms. The package provides robust primitives for efficient manipulation of regular languages, making it easy to write custom grader code for other types of assessments involving finite automata. This means the FSM Builder is very self-contained (the automata package is the only major dependency) and more extensible than previous tools.

In particular, the package provides optimized algorithms for converting regular expressions to NFAs, NFAs to DFAs, minimizing DFAs, enumerating the strings belonging to the language of a DFA, and the product construction for DFAs. These subroutines were critical in writing the grading and feedback algorithms for the FSM Builder, and represent the most technically challenging parts of the implementation. The availability of these subroutines opens the door for the creation of more custom questions using the tool.

## 7 Evaluation

### 7.1 Scalability

A key motivating factor in the development of FSM Builder was scalability to both large courses and large numbers of questions. The FSM Builder has been very successful on these fronts, as in our course, we were able to convert our entire backlog of finite automata questions (over 50) to automated practice problems without the need to write custom grader code for any individual question (only using the autograding functionality described in Section 6). The tool scaled well to the questions themselves, as we did not encounter any efficiency issues with the grading or feedback algorithms, and the tool integrated cleanly with PrairieLearn. This scalability has been observed in other courses, as the large introductory discrete mathematics course in our department has also begun using the FSM Builder.

The FSM Builder has also proven to be extensible and flexible, as we were able to develop multiple questions using custom grading algorithms.

### 7.2 Student Response

To evaluate student responses to the FSM Builder, we conducted a voluntary, fully anonymous online survey distributed to students in the introductory computing theory course during the Fall 2023 semester. The survey consisted of Likert scale questions about the experience of using the tool. All students had interacted with the tool as part of a short (required) homework question. Of the 383 enrolled students, 246 students had additionally used the tool as part of optional review content, and 196 responded to the survey. Students were incentivized by granting the entire course a small amount of extra credit if over half of students in the course completed the survey. The survey platform prevented students from submitting responses more than once.

The University of Illinois Urbana-Champaign IRB office gives this survey a non-human subjects research designation, as all of the data collection was completely anonymous.

The goal of the survey was to assess general impressions of the user interface and feedback of the tool in comparison to feedback given by course staff. The questions were developed to assess the general sentiment towards different aspects of the tool, rather than doing detailed comparisons with other tools. Importantly, the FSM Builder was not being used to replace any existing course content, so our evaluation focused on whether students found the content delivered through the tool useful and engaging.

The results of the survey are shown in Figure 2, and overall feedback to the FSM Builder was very positive. Notably, there were more students who agreed the FSM Builder provided useful feedback (statement 8, 131 positive responses) than students who agreed written homework graded by a TA provided useful feedback (statement 9, 125 positive responses). While not conclusive evidence feedback by the tool was superior, this demonstrates students were generally satisfied with the automated feedback, indicating the FSM Builder is able to provide valuable feedback to students at scale.

The statement most disagreed with was number 9, that it is easier to use the FSM Builder than to construct an automaton on paper. However, more respondents agreed than disagreed, and this mixed response indicates that using the tool is still advantageous for large courses, as grading automata on paper does not scale well among many students. Despite this, 67% of respondents (131 out of 196) agreed the user interface of the FSM Builder was intuitive (statement 6).

Students also responded positively to statement 10 (61% agreed, 119 out of 196), that the tool made it easier to typeset written homework. This provides some positive evidence for the choice to use the FSM Designer as the basis for the user interface.

## 8 Adopting the FSM Builder

To try the FSM builder, look at the DFA and NFA practice assessments available in our public course instance [1].

To use the FSM Builder in your course, start by following the onboarding instructions for PrairieLearn [16]. Next, follow the

[1]https://www.prairielearn.org/pl/course_instance/129595

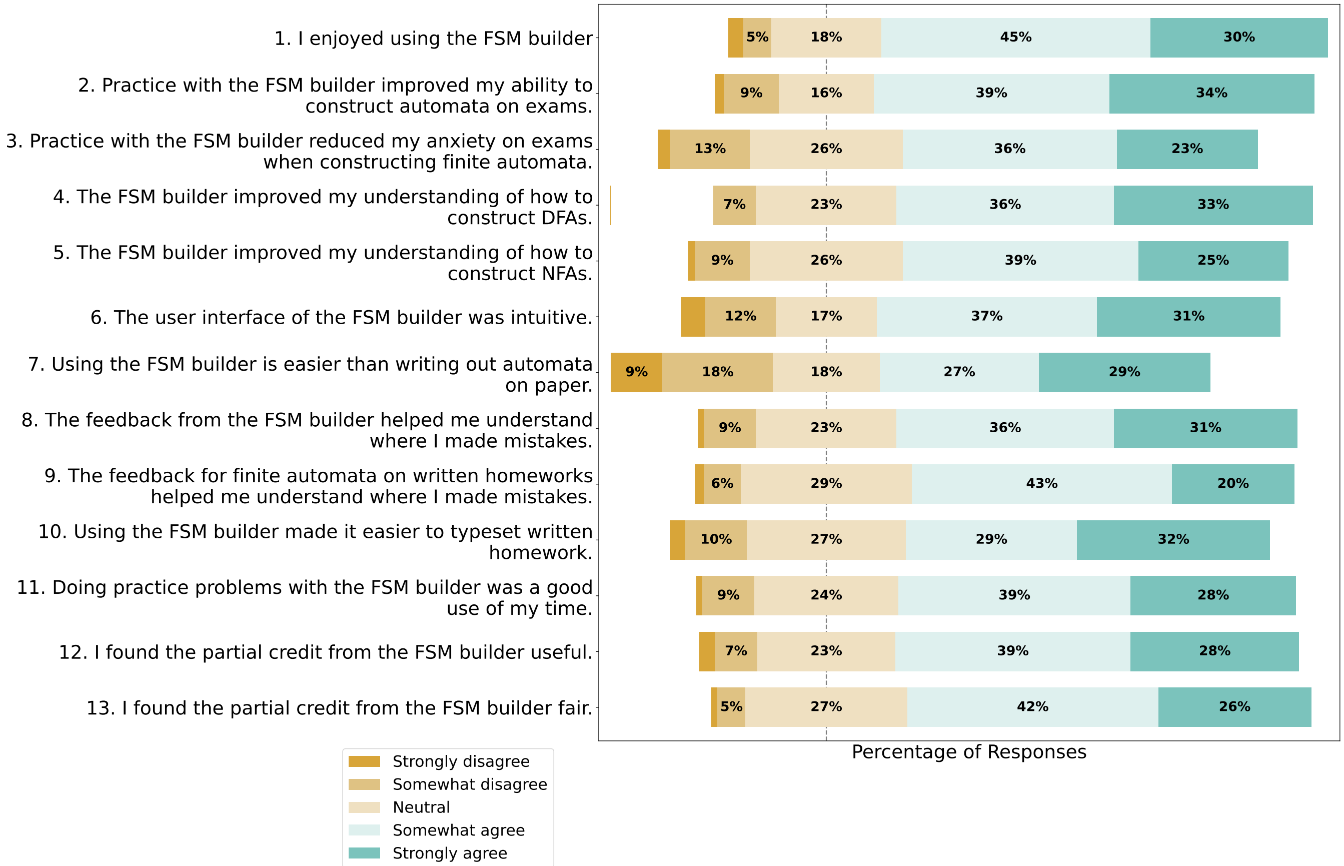



**Figure 2: Responses to the Likert scale questions on the survey. There were 196 total responses.**

directions in this repository [2] on integrating the FSM builder with your course. Note that the code for this implementation is self-contained.

## 9 Limitations and Future Work

The key limitation of this work is the scope for the evaluation of the FSM Builder. Most of the questions students practiced on were part of optional review content and of very similar format, and completion of the survey was voluntary. The results from the survey were positive, but the nature of the evaluation means we can only draw general conclusions. Although we found this acceptable for an initial investigation, more detailed data is required to draw stronger conclusions on the effectiveness of the FSM Builder.

With a variety of different independent components, the FSM Builder provides a number of avenues for future research. Specifically, the graphical interface shown to students and the backend grader code are modular components, and further work could explore the student response to different types of partial credit, different feedback schemes, and potentially a different student interface to that of Wallace [18].

Furthermore, the backend grader code used by the FSM Builder can be easily adapted to work with regular expressions, using subroutines in the automata package [9]. We have such a tool in our course, providing nearly identical feedback to that of the FSM Builder. Possible future work could examine the effectiveness of this regular expression tool.

Orthogonal to altering the FSM Builder itself, most of the problems completed by students were in very similar assessment contexts. A direction for further work could examine student responses to the partial credit and feedback schemes when used for different types of questions (including questions that involve randomization and automated generation of question prompts), and in different assessment contexts, such as exams.

## Acknowledgments

We thank Seth Poulsen and Yael Gerter for suggestions and feedback on the survey questions we used. We would also like to thank Morgan Fong, Yael Gerter, Seth Poulsen, and the Computers & Education research group at the University of Illinois Urbana-Champaign for feedback on earlier versions of this paper

Funding for this work was partially provided by the Strategic Instructional Innovations Program at the University of Illinois Urbana-Champaign.

[2] https://github.com/eliotwrobson/FSMBuilder

## References


[1] Marco Almeida, Nelma Moreira, and Rogério Reis. 2009. Testing the Equivalence of Regular Languages. *J. Autom. Lang. Comb.* 15 (2009), 7–25. https://api.semanticscholar.org/CorpusID:9014414

[2] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated Grading of DFA Constructions. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, Francesca Rossi (Ed.). IJCAI/AAAI, 1976–1982. http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6759

[3] Ivona Bezáková, Kimberly Fluet, Edith Hemaspaandra, Hannah Miller, and David E. Narváez. 2022. Effective Succinct Feedback for Intro CS Theory: A JFLAP Extension. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 976–982. https://doi.org/10.1145/3478431.3499416

[4] Ryan Cavalcante, Thomas Finley, and Susan H. Rodger. 2004. A visual and interactive automata theory course with JFLAP 4.0. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2004, Norfolk, Virginia, USA, March 3-7, 2004*, Daniel T. Joyce, Deborah Knox, Wanda P. Dann, and Thomas L. Naps (Eds.). ACM, 140–144. https://doi.org/10.1145/971300.971349

[5] Pinaki Chakraborty, P. C. Saxena, and C. P. Katti. 2011. Fifty Years of Automata Simulation: A Review. *ACM Inroads* 2, 4 (dec 2011), 59–70. https://doi.org/10.1145/2038876.2038893

[6] Loris D'Antoni, Martin Helfrich, Jan Kretínský, Emanuel Ramneantu, and Maximilian Weininger. 2020. Automata Tutor v3. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 3–14. https://doi.org/10.1007/978-3-030-53291-8_1

[7] Loris D'Antoni, Matthew Weavery, Alexander Weinert, and Rajeev Alur. 2015. Automata Tutor and what we learned from building an online teaching tool. *Bull. EATCS* 117 (2015). http://eatcs.org/beatcs/index.php/beatcs/article/view/365

[8] Jeff Erickson, Jason Xia, Eliot Wong Robson, Tue Do, Aidan Tzur Glickman, Zhuofan Jia, Eric Jin, Jiwon Lee, Patrick Lin, Steven Pan, Samuel Ruggerio, Tomoko Sakurayama, Andrew Yin, Yael Gertner, and Brad Solomon. 2023. Auto-graded Scaffolding Exercises For Theoretical Computer Science. In *Proc. 2023 ASEE Annual Conference & Exposition*. https://doi.org/10.18260/1-2--42347

[9] Caleb Evans and Eliot W. Robson. 2023. automata: A Python package for simulating and manipulating automata. *Journal of Open Source Software* 8, 90 (Oct. 2023), 5759. https://doi.org/10.21105/joss.05759

[10] 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*. Association for Computing Machinery, New York, NY, USA.

[11] 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

[12] Daphne Norton. 2009. Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP. (04 2009).

[13] Susan H. Rodger. 2006. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., USA.

[14] Susan H. Rodger, Eric Wiebe, Kyung Min Lee, Chris Morgan, Kareem Omar, and Jonathan Su. 2009. Increasing Engagement in Automata Theory with JFLAP. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 403–407. https://doi.org/10.1145/1508865.1509011

[15] M. Sipser. 2012. *Introduction to the Theory of Computation*. Cengage Learning. 45–47 pages.

[16] PrairieLearn Team. 2021. PrairieLearn Documentation. https://prairielearn.readthedocs.io/en/latest/ Accessed: April 2024.

[17] PrairieLearn Team. 2021. PrairieLearn Question Element Documentation. https://prairielearn.readthedocs.io/en/latest/devElements/ Accessed: April 2024.

[18] Evan Wallace. 2015. Finite State Machine Designer. Github repository. https://github.com/evanw/fsm

[19] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning. *ASEE Annual Conference and Exposition, Conference Proceedings* 122nd ASEE Annual Conference and Exposition: Making Value for Society, 122nd ASEE Annual Conference and Exposition: Making Value for... (2015). https://doi.org/10.18260/p.24575 2015 122nd ASEE Annual Conference and Exposition ; Conference date: 14-06-2015 Through 17-06-2015.

[20] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. ASEE Conferences, Seattle, Washington, 26.1238.1–26.1238.14. https://peer.asee.org/24575.