Problem Generation for DFA Construction

Alexander Weinert

May 18, 2014

Abstract

Teaching the construction of DFA to computer science students relies in great part on practice problems, in which the student is asked to construct an automaton for some given language. Nowadays, these practice problems are constructed by humans in order to teach certain general concepts that, once understood, can be reused in the construction of other DFA. A problem arises if the student finishes all their practice problems, but still has not understood the underlying concept. We present a method to generate practice problems using one concrete example problem as the input. During the construction we make sure that the resulting problem has the same level of difficulty and exercises the same concepts as the original problem. We also present an evaluation of our algorithm on 20 examples from a well known textbook on automata theory.

1 Introduction and Motivation

Nearly every student of computer science is taught the concept of Deterministic Finite Automata, or DFA for short, at some point during their studies. The teaching of this subject usually consists of a formal definition, some examples of automata that recognize simple string languages, and some exercises for the students, which are often of the form "Construct a DFA $\mathcal{A}$ that recognizes the regular language L". L is most often defined in English.

It is then usually assumed that the student understands the techniques that are used when constructing automata. However, this may not be the case. It may be that the student would like to have more practice problems that exercise the same principles. In this case the student usually has to rely on problems from previous semesters or from other courses for more exercise. This poses a set of new difficulties, since the older problems may use different notations or teach concepts in a different order, which would further complicate the learning process.

We present a technique that takes a regular language L as input and outputs a set of regular languages in such a way that the minimal automata that recognize the output languages use the same concepts as the minimal automaton that recognizes L and are of similar complexity.

Solving problems of this form, i.e., constructing the automaton for a given language L is very simple if the language L is defined as a formula in Monadic Second Order Logic, or MSOL for short. For every MSOL formula we can construct an automaton that recognizes exactly the language that is defined by the formula, and vice versa [Tho97]. Grading student solutions for these problems has been investigated in [ADG$^+$13], which forms the basis for most of our work in this project.

In [SGR12], the authors research the automated generation of algebra problems from a given problem. A more general treatment of problem generation can be found in [SSG12], where the topic is investigated for a massively open online course in embedded systems. Problem generation has also been investigated in [AGK13] in the context of natural deduction. Generating feedback for faulty student solutions has been not

been researched for DFA construction exercises, but there exists an approach for introductory programming assignments in [SGSL13].

2 Problem Definition

We only concern ourselves with tasks of the form "Construct a DFA that recognizes the language L", where L is some regular language. Our goal is to construct a set of tasks of the same form, such that the student exercises the same principles for the construction of automata when solving the new problems. Since we only concern ourselves with DFA, we are going to use the terms "automaton" and DFA interchangeably for the rest of this report.

Since the only possibility for variations in this kind of tasks is the choice of the regular language L, the task reduces to the following: Given some regular language L, construct a new regular language L', so that the automata that recognize L and L' use the same concepts.

In order to make the approach feasible, we only consider the minimal automata that recognize L and L'. We also assume that the language L is given as a formula in MOSEL. This logic provides syntactic sugar over the well known Monadic Second Order Logic, which is in turn equivalent to finite automata. Thus, MOSEL formulas correspond to regular automata as well. The full definition of MOSEL as well as the transformation between automata and MOSEL formulas is detailed in [ADG+13].

Thus, our final problem is as follows: Given some MOSEL formula φ, construct another MOSEL formula φ', such that the corresponding minimal automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\varphi'}$ use the same concepts for recognizing their respective language and are of similar complexity.

3 Approach

Our approach works in three steps: Abstraction, Concretization and Filtering. In the first step, we *abstract* the given MOSEL formula into a CHOICE-MOSEL formula, which represents a set of MOSEL formulas, including the original one. We then *concretize* the CHOICE-MOSEL formula, i.e., we construct the set of all MOSEL formulas that are represented by the CHOICE-MOSEL formula. In a final step we *filter* the resulting MOSEL formulas in order to remove formulas whose minimal automaton differs too much from the minimal automaton of the original formula.

Since we are exclusively dealing with the parse tree of a formula in this project, we use the terms "formula" and "parse tree of a formula" interchangeably. This enables us, for example, to say that we traverse the nodes of a formula, when we mean to traverse the nodes of the corresponding AST.

3.1 Abstraction

Our main idea for this first step of the construction of tasks is to transform a given MOSEL formula ϕ into a CHOICE-MOSEL formula ϕ_c, which represents a number of MOSEL formulas. The syntax of CHOICE-MOSEL is defined in figure 1.

In order to define the abstraction of a MOSEL formula, we first consider the type of a node in a MOSEL formula. We note that the syntax of MOSEL as defined in [ADG+13, figure 2] features some primitives, namely string and integer constants, first- and second order quantifiers, boolean operators and integer comparators

MOSEL formulas are made up of a MOSEL-predicate at top level, which is represented by the non-terminal ϕ in the MOSEL-grammar. This predicate may in turn refer to positions and sets, represented

$$
\begin{array}{ccc}
n \rightarrow (0 \ldots 9)^* & c \rightarrow (a \ldots z) & str \rightarrow (a \ldots z)^* \\
CInt \rightarrow \texttt{ChoiceInt}(n) & CChar \rightarrow \texttt{ChoiceChar}(c) & CStr \rightarrow \texttt{ChoiceString}(s)
\end{array}
$$

$$
\begin{array}{rl}
CPred \rightarrow & \texttt{simBoolOp}(CPred, CPred) \mid \texttt{comBoolOp}(CPred, CPred) \mid \texttt{FOQuant}(str, CPred) \mid \texttt{SOQuant}(str, CPred) \mid \\
& \texttt{neg}(CPred) \mid \texttt{boolConst}() \mid \texttt{posComp}(CPos, CPos) \mid \texttt{atPos}(CChar, CPos) \mid \texttt{atSet}(CChar, CSet) \mid \\
& \texttt{in}(CPos, CSet) \mid \texttt{setCard}(CSet, CInt) \mid \texttt{setCardMod}(CSet, CInt, CInt) \mid \texttt{startEnd}(CStr) \mid \texttt{isEmpty}() \\
CPos \rightarrow & \texttt{FOVar}(str) \mid \texttt{endPos}() \mid \texttt{incDec}(CPos) \mid \texttt{occ}(CStr) \\
CSet \rightarrow & \texttt{SOVar}(str) \mid \texttt{indOf}(CStr) \mid \texttt{setOp}(CSet, CSet) \mid \texttt{all}() \mid \texttt{posComp}(CSet)
\end{array}
$$

Figure 1: The definition of CHOICE-MOSEL

by the nonterminals P and S, respectively. Thus, we define the set of primitive types as $primTyp := \{\textbf{int}, \textbf{string}, \textbf{FOquant}, \textbf{SOquant}, \textbf{boolOp}, \textbf{intComp}, \textbf{pred}, \textbf{pos}, \textbf{set}\}$. We can then define the type of a node as the tuple of types of its arguments. For example, the type of the node $(|S|\%m \text{ CMP } n)$ would be $(\textbf{set}, \textbf{int}, \textbf{intComp}, \textbf{int})$, whereas the type of $(\phi \text{ C } \phi)$ would be $(\textbf{pred}, \textbf{boolOp}, \textbf{pred})$.

Our abstraction is guided by this idea of types. Two MOSEL-nodes can only be represented by the same CHOICE-MOSEL-node if they have the same type. The converse, however, is not true. Consider, for example, the two formulas $\phi_1 \vee \phi_2$ and $\phi_1 \Rightarrow \phi_2$. If we abstracted these formulas with the same node, it would mean that we could not distinguish between $\phi_1 \Rightarrow \phi_2$ and $\phi_1 \vee \phi_2$ after the abstraction anymore. Constructing the minimal DFA for the former formula requires the student to understand both the concepts of negation as well as the union of automata. The construction of the minimal DFA for the latter formula requires only the concept of intersection, however. Thus, we map these two nodes to different nodes in the abstraction.

$\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are abstracted with the same node, however, since they require the concept of intersection and union, however, which are very similar to each other and can thus be interchanged. The transformation function $abstract$: MOSEL $\rightarrow$ CHOICE-MOSEL is defined in figure 2.

$$
\begin{array}{ccc}
abs(n) \mapsto CInt(n) & abs(c) \mapsto CChar(c) & abs(s) \mapsto CStr(s)
\end{array}
$$

$$
\begin{array}{lll}
\left.\begin{array}{l} \phi_1 \wedge \phi_2 \\ \phi_1 \vee \phi_2 \end{array}\right\} & \mapsto \texttt{simBoolOp}(abs(\phi_1), abs(\phi_2)) & \\
\left.\begin{array}{l} \phi_1 \Rightarrow \phi_2 \\ \phi_1 \Leftrightarrow \phi_2 \end{array}\right\} & \mapsto \texttt{comBoolOp}(abs(\phi_1), abs(\phi_2)) & \\
\neg\phi & \mapsto \texttt{neg}(abs(\phi)) & \\
\left.\begin{array}{l} \exists x.\phi \\ \forall x.\phi \end{array}\right\} & \mapsto \texttt{FOQuant}(x, abs(\phi)) & \\
\left.\begin{array}{l} true \\ false \end{array}\right\} & \mapsto \texttt{boolConst}(x, abs(\phi_1)) & \\
P_1 \ CMP \ P_2 & \mapsto \texttt{posComp}(abs(P_1), abs(P_2)) & \\
a@P & \mapsto \texttt{atPos}(abs(a), abs(P)) & \\
a@S & \mapsto \texttt{atSet}(abs(a), abs(S)) & \\
P \in S & \mapsto \texttt{in}(abs(P), abs(S)) & \\
|S|\%m \ CMP \ n & \mapsto \texttt{setCardMod}(abs(S), abs(m), abs(n)) & \\
|S| \ CMP \ n & \mapsto \texttt{setCard}(abs(S), abs(n)) & \\
\left.\begin{array}{l} begWt(s) \\ endWt(s) \end{array}\right\} & \mapsto \texttt{startEnd}(abs(s)) & \\
isEmpty & \mapsto \texttt{isEmpty}(abs(S), abs(n)) &
\end{array}
$$

$$
\begin{array}{lll}
x & \mapsto \texttt{FOVar}(x) & \\
\left.\begin{array}{l} fst \\ last \end{array}\right\} & \mapsto \texttt{endPos}() & \\
\left.\begin{array}{l} P+1 \\ P-1 \end{array}\right\} & \mapsto \texttt{incDec}(abs(P)) & \\
\left.\begin{array}{l} fstOcc(s) \\ lastOcc(s) \end{array}\right\} & \mapsto \texttt{occ}(abs(s)) & \\
X & \mapsto \texttt{SOVar}(X) & \\
indOf(s) & \mapsto \texttt{indOf}(abs(s)) & \\
\left.\begin{array}{l} S_1 \cap S_2 \\ S_1 \cup S_2 \end{array}\right\} & \mapsto \texttt{setOp}(abs(S_1), abs(S_2)) & \\
all & \mapsto \texttt{all}() & \\
\left.\begin{array}{l} psLt(P) \\ psLe(P) \\ psGt(P) \\ psGe(P) \end{array}\right\} & \mapsto \texttt{posComp}(abs(P)) &
\end{array}
$$

Figure 2: The definition of abs

3.2 Concretization

In this step our goal is to reverse the abstraction; We are given a CHOICE-MOSEL formula ϕ_c and want to construct the set of all MOSEL formulas ϕ that can be abstracted to ϕ_c. More formally, we want to construct the set $Conc(\phi_c) := \{\phi \mid abs(\phi) = \phi_c\}$.

3.2.1 Idea

The basic idea of concretization is very simple. It is simple to reverse the abstraction by walking over the tree in postorder. If the current node is a leaf, we return all nodes that can be abstracted to this node. Due to our construction of abs, we know that all nodes that can be abstracted to a leaf node are leaf nodes themselves. Thus, we can simply construct them without arguments.

This is not the case for literals, since all integer- and string literals are abstracted to the same node `CInt` and `CStr`, respectively. Thus, the set of concretizations of any node of type `CInt` and `CStr` is infinitely large. However, since we are eventually only interested in the MOSEL formulas whose corresponding minimal DFA is of similar complexity to the minimal DFA for the original formula, we restrict the concretization of these two nodes as follows: `CInt(n)` is concretized to all integers m in the interval $[\max(0, \lfloor m \cdot (1-p) \rfloor), \lceil m \cdot (1+p) \rceil]$ for some p. In our implementation we chose $p = 0.5$. `CStr(s)` is concretized to all strings of the same length as s. This is based on the observation that changing the length of a string constant in a MOSEL formula usually changes the complexity of the corresponding automaton quite drastically.

If the node is an inner node of the formula, we have already concretized all of its arguments. Due to our constraint that two MOSEL nodes can only be abstracted to the same CHOICE-MOSEL node if they have the same type, we know that all possible concretizations take the same number and type of arguments. Thus, we have concretized all possible arguments for all possible concretizations of the current node, which enables us to use them in the concretization of the parent node.

Note that this concretization uses exponential space, since we construct all subtrees recursively. In order to avoid this, we now describe an alternative method of concretization that relies on SMT-constraints, which also allows us greater control over the concretization.

3.2.2 Constraint Generation

The alternative idea for the concretization of a CHOICE-MOSEL-formula ϕ_c is to convert ϕ_c into SMT-constraints, such that each model of these constraints corresponds to one concretization of ϕ_c. We then use a SMT-solver to enumerate all models and reconstruct the corresponding MOSEL-formula for each model. This allows us to enumerate all concretizations using less memory and also enables us to abort the enumeration of concretizations at any point. Furthermore, we are able to impose additional constraints on the resulting formulas. This in turn allows us to exclude formulas whose minimal automaton differs too much in complexity from the original automaton already at this stage.

The general construction of the constraints is shown in figure 3. The function `Number-of-concretizations(CNode)` returns the number of MOSEL-nodes that can be abstracted to the given CHOICE-MOSEL type. Note that the results of this function for all CHOICE-MOSEL-nodes can be precomputed using the definition of abs from figure 2. For example `Number-of-concretizations(comBoolOp)` would be 2, since there are two MOSEL-nodes that can be abstracted into `comBoolOp`.

The transformation from a model to a MOSEL-formula can then be easily performed using a postorder traversal of the CHOICE-MOSEL-formula, where each visited node constructs the MOSEL-node associated

function CONC(CHOICE-MOSEL-formula $\phi_c = \texttt{CNode}(arg_1, \ldots, arg_n)$)
    $x \leftarrow$ `Fresh-variable`
    $numConc \leftarrow$ `Number-of-concretizations(CNode)`
    $C \leftarrow \emptyset$
    $C \leftarrow C \cup \{0 \leq x, x \leq numConc\}$
    for all $1 \leq i \leq n$ **do**
        $C \leftarrow C \cup conc(arg_i)$
    end for
end function

Figure 3: Algorithm for construction of SMT-constraints from a CHOICE-MOSEL-formula

with the value of its unique variable.

There are two additional constraints on the concretization of constants we can now impose easily. First, we observe that, in order to preserve the concepts used in constructing the minimal DFA for the original formula, it proved to be beneficial to preserve equality of literals. If, for example, a string literal s appears at two places in the original formula, we do not want two different string literals to be concretized at these two places, but we want to use the same literal s' at both places. The same holds for integer literals. This can easily be achieved by reusing the same variables for equal literals instead of getting a fresh variable for every node in the first step of CONC.

Another observation is that is beneficial to also preserve inequality of literals. Consider, for example the formula $begWt(a) \wedge endWt(b)$, i.e., the formula describing the language of all strings that start with an a and end with a b. When constructing the automaton for this formula, the student does not have to consider words of length 1, since no word of this length can start and end with different symbols. If we were to concretize the formula $begWt(a) \wedge endWt(a)$, the student would have to consider this new corner case, which would make the task of constructing the automaton harder. This constraint can easily be imposed by collecting all variables that are used by nodes of type `CInt`, `CChar` and `CStr`, respectively, and impose the additional constraint $x_i \neq x_j$ for each pair of these variables.

These two additional constraints of preserving equality and inequality are not an integral part of our technique. They exist merely as an optimization that allows us to filter out undesirable concretizations already at this early stage which in turn improves the runtime of the complete algorithm. These optimizations might also prevent desirable MOSEL-formulas from being concretized in some cases. Our experiments have shown, however, that the benefit of a shorter runtime greatly outweighs the downside and that the algorithm still produces a sufficient number of good MOSEL-formulas.

3.3 Filtering

We have shown how to produce a set of MOSEL-formulas that are similar in structure to given MOSEL-formula. However, once we finish abstraction and concretization, there are still some formulas that we do not want to give to a student, either because their corresponding automaton is too simple, or because the automaton differs too much from the original automaton. In order to remove these formulas from the output, we filter the result of the concretization step using two different filters. The first one removes those formulas from the output whose automaton is probably too simple to construct. The second one removes those whose automaton differs too much from the original automaton. In order to keep the runtime of these filters low, both filters work heuristically.

3.3.1 Triviality Filter

Since we only manipulate the MOSEL-formulas syntactically, we end up with unsatisfiable formulas in many cases. Consider again the formula $begWt(a) \wedge endWt(b)$. The concretization results, among others, in the formulas $begWt(a) \wedge begWt(b)$ and $endWt(a) \wedge endWt(b)$, both of which are unsatisfiable, i.e., they describe the empty language. In order to exactly recognize this kind of formulas, we would have to construct the automaton for each formula and then check if it consists only of a single state.

Since the construction of the DFA is expensive in terms of runtime, we instead use a method similar to the estimation of the language density in [ADG+13]: We evaluate the concretized MOSEL-formula on all strings up to a certain length. If the formula evaluates to false on all these strings, we assume that it evaluates to false on all strings, i.e., that it describes the empty language, and remove it from the output. With the same reasoning we also remove all formulas that evaluate to true on all strings up to a certain length from the output. In our experiments, we tested each formula on all strings up to length 4, which proved to be a good trade-off between runtime and effectiveness of the filter.

3.3.2 Complexity Filter

After we filter out the trivial formulas, the output might still contain some formulas whose minimal automaton differs too much from the minimal automaton of the original formula. In order to assess the difference between the automaton of the generated formula and the automaton of the original formula correctly, we would have to construct both automata and then compute the minimal edit distance between them, as detailed in [ADG+13]. Since the computation of the minimal edit distance is a computationally very expensive operation in our implementation, we instead only compare the number of states of both automata. If the number of states differs by more than some percentage p, then we reject the formula. In our experiments, we chose $p = 0.2$, which showed to be a good trade-off between rejecting as many undesirable formulas and keeping as many formulas as possible.

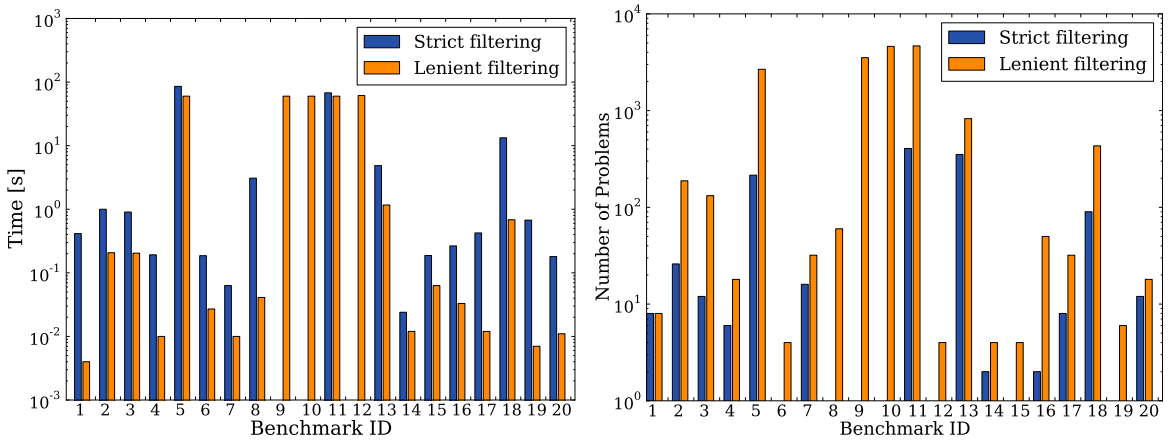
4 Results

We implemented our method using the tool available at `www.automatatutor.com` and tested it using 20 examples of regular languages taken from [Hop01]. In these experiments we chose to preserve both equality and inequality of literals as described in section 3.2.2. We used two different filter configurations. In the first configuration, which we call "lenient filtering", we only applied the triviality filter described in section 3.3.1. In the second configuration, we applied this filter first, and subsequently applied the complexity filter from section 3.3.2. The results of these benchmarks are shown in figure 4.

We see that we were able to concretize all formulas in less than 10 seconds for most examples. The outliers are mostly languages taken from a task of the form "Construct a NFA that recognizes the regular language L". This leads to a formula of higher complexity, which in turn slows the algorithm down. We can also observe that the number of generated problems is sufficiently high for most of the benchmarks. The few cases in which no formulas were generated were for exercises for the construction of NFA instead of DFA again.

We have also investigated the relation between the number of variables used in the SMT constraints and the runtime of the complete algorithm. The results are presented in figure 5. Note the logarithmic scale on the y-axis of this figure.

We can see that the runtime of the algorithm increases rapidly if we use more variables in the SMT query, i.e., if we have a larger input formula. The increase in runtime does, however, not appear to be exponential



(a) Runtime of the algorithm

(b) Number of problems generated

Figure 4: Performance of the algorithm on benchmarks

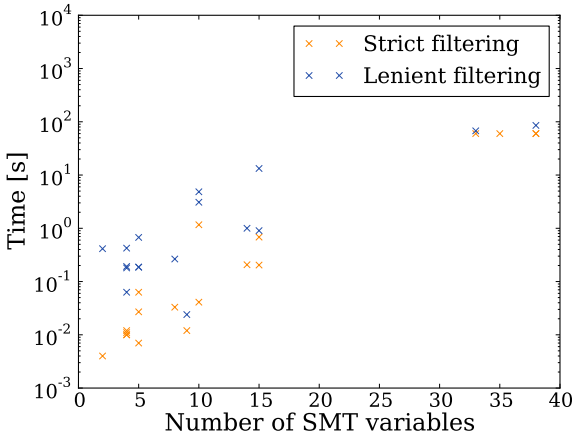


Figure 5: Relation between the runtime and the number of SMT variables used

in the number of variables. This is owed to the fact that the SMT query is of a very simple form, since there are no interdependencies between the constraints and all constraints are of the form $var \leq const$.

5 Conclusions and Future Work

We have presented an algorithm that takes a formal description of a regular language as input and returns a set of regular languages for which the construction of the minimal DFA is of similar difficulty. We have also presented an evaluation of this technique on multiple examples of regular languages taken from one of the most well-known textbooks on automata theory. This evaluation showed that our algorithm produces a sufficient number of regular languages in a reasonably long time that allows for direct interaction with the tool instead of overnight computation.

There are multiple viable directions for future work on this topic. The results in section 4 show that the algorithm produces a sufficient number of new problems in a reasonable time, but they state nothing about the quality of the resulting problems. Even though we have inspected the problems manually and found them to be of similar difficulty as the original problem, a more rigorous investigation should define an objective metric of quality of the results and measure it. One possible approach would be to compare the resulting problems with the original one using the grading metrics from [ADG$^+$13].

Another challenge is that the evaluation shows that, while our algorithm works well on DFA, it times out when constructing new problems of the form "Construct a NFA that recognizes the language L". This is owed to the fact that, since MOSEL can be seen as a deterministic description language, the mere formulation of these problems in MOSEL transforms the nondeterministic choice into an enumeration of all possibilities. Take, for example, the language of all strings that start and end with the same letter. The MOSEL description for this language is $(begWt(a) \wedge endWt(a)) \vee (begWt(b) \wedge endWt(b))$ This also leads to the problem that the resulting problems are neither well suited for DFA construction nor for NFA construction. However, this could be alleviated by the introduction of nondeterministic operators into MOSEL, which would be an interesting direction for further investigation.

Finally, we approached problem generation using the logical description of a regular language. There are, however, multiple ways to represent such a language. It might feel more natural to manipulate the DFA corresponding to a regular language directly and generate new problems that way. One problem that poses itself, however, would be to ensure that this mutation does not change the concepts that are used for constructing that DFA. Furthermore, a minor change to a minimal automaton might result in an automaton that can be minimized quite drastically, resulting in a construction task of greatly different difficulty. Thus, the generation of tasks using manipulation of DFA would have to be investigated as well.

6 Acknowledgements

The author would like to thank Prof. Sanjit Seshia, the students of CS298-98 and Loris D'Antoni for continuous feedback on this project. Thanks also go to Rajeev Alur, Loris D'Antoni, Björn Hartmann, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan, who developed `www.automatatutor.com`, which was used as the basis for the implementation of this project.

References

[ADG+13] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of DFA constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 1976–1982, 2013.

[AGK13] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. Automatically Generating Problems and Solutions for Natural Deduction. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 1968–1975, 2013.

[Hop01] John Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, 2001.

[SGR12] Rohit Singh, Sumit Gulwani, and Sriram K Rajamani. Automatically Generating Algebra Problems. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 1620–1627, 2012.

[SGSL13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26. ACM, 2013.

[SSG12] Dorsa Sadigh, Sanjit A. Seshia, and Mona Gupta. Automating Exercise Generation: A Step towards Meeting the MOOC Challenge for Embedded Systems. In *Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*, pages 2:1–2:8, 2012.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, pages 389–455. Springer Berlin Heidelberg, 1997.