Teaching Automata Theory and Formal Languages with Dafny

Ran Ettinger

Abstract

This talk describes a mandatory second-year undergraduate course on *Automata Theory and Formal Languages* for computer science and software engineering students, taught using the verification-aware programming language *Dafny*. The course closely follows *Automata, Computability and Complexity: Theory and Applications* by Elaine Rich. It covers finite state machines and regular expressions, pushdown automata and context-free grammars, ending with Turing machines and decidability. By encoding the formal definitions and key algorithms from this textbook in Dafny, we transform a traditionally abstract mathematics course into an interactive, hands-on learning experience where students receive immediate feedback on both their machine constructions and their reasoning about formal languages.

Foundational Definitions: Strings and Languages. The course begins with foundational definitions formalized directly in Dafny's type system. We define symbols as characters, strings as sequences of symbols, and formal languages as potentially infinite sets of strings. This use of infinite set types is central to our approach, as it allows us to define languages declaratively through set comprehension. Students immediately work with standard string operations: the empty string epsilon, concatenation of two strings, replication that produces w^n, string reversal, and counting occurrences of a symbol. We prove properties such as associativity of concatenation and that epsilon is the identity element, using both direct assertions and Dafny's calculation construct for equational reasoning.

A key early example is a lemma establishing that the reverse of a concatenation equals the concatenation of the reverses in opposite order: reversing wx yields the same result as concatenating the reverse of x with the reverse of w. The proof proceeds by induction on the second string and uses a calculation block to present a chain of equalities, each justified by either definitional unfolding, previously proven lemmas, or the induction hypothesis. Students observe how Dafny verifies each step mechanically, providing confidence that no logical gaps exist.

Operations on languages are similarly defined using infinite set comprehension. Language union and intersection leverage Dafny's built-in set operations, while language concatenation is defined as the set of all strings that can be formed by concatenating a string from the first language with a string from the second. The Kleene star is particularly elegant: it comprises all strings that can be formed by concatenating any finite sequence of strings from the given language. Students work through exercises establishing properties such as "concatenating a language with the singleton set containing only epsilon yields the original language" using Dafny's proof by calculation features.

The interplay between string operations and language operations culminates in lemmas such as one asserting that reversing the concatenation of two languages yields the concatenation of their reverses in opposite order. Its proof uses a calculation block to transform the left-hand side through a series of set comprehension manipulations, with one critical step justified by invoking the previously proven lemma about string reversal for all pairs of strings drawn from the two languages. This universal quantification construct demonstrates how lemmas on individual strings compose to build results about entire languages. Students see that mathematical reasoning is not merely symbolic manipulation but a structured process that can be mechanically verified, and they practice this themselves in assignments where they define specific languages using set comprehension and prove membership or non-membership of particular strings.

Finite State Machines and Regular Languages. The automata portion of the course begins with the formalization of deterministic and nondeterministic finite state machines. Each machine is represented as a tuple comprising a set of states, an alphabet, a transition specification (a total function from state and symbol to state for DFSMs, a set of triples containing source state, symbol-or-epsilon, and target state for NDFSMs), a start state, and a set of accepting states. Most importantly, we leverage Dafny's expressive type system and support for infinite sets to define the formal language accepted by a machine as an infinite set comprehension: a string is in the language if and only if, starting from the initial state, the machine can reach an accepting state after consuming the entire input. This declarative specification allows students to reason about language acceptance at a high level of abstraction while remaining completely precise and machine-checkable.

For DFSMs, the acceptance condition is straightforward: we recursively compute the final state reached after processing the entire input string, and accept if that state is in the accepting set. For NDFSMs, the formalization is more subtle due to epsilon transitions. We define an epsilon-closure function that, given a state, computes the set of all states

Dafny Workshop '26, Rennes, France
2026.

reachable via epsilon transitions alone. The acceptance condition for NDFSMs processes the input recursively: starting from the epsilon-closure of the initial state, each input symbol transitions to a new set of states, computed by following all possible labeled transitions and then taking the epsilon-closure of each resulting state. The machine accepts if, after processing the entire input, any state in the final set is accepting.

Students study canonical examples such as a DFSM that accepts all binary strings with an odd number of ones. For this machine, we work through a complete correctness proof in Dafny, establishing that the formal language accepted by the machine equals the set of binary strings where the count of ones modulo two equals one. The proof proceeds by induction on the input string and defines separate lemmas for each state of the machine, relating the parity of ones in the remaining input to whether the machine will reach an accepting state from that particular state. Students observe how Dafny's verifier confirms each step of the reasoning and learn to write assertions and invoke lemmas to guide the verification process. This example serves as a template for the kinds of arguments students encounter throughout the course.

We provide executable simulators for both DFSMs and NDFSMs, implemented as verified methods with preconditions and postconditions that ensure the result correctly reflects membership in the formal language. These simulators are imperative interpreters taken nearly literally from the textbook algorithms: they use a while loop to iterate over the input string character by character, maintaining the current state (for DFSMs) or current set of states (for NDFSMs) as a mutable variable. The proof of correctness is established through loop invariants that connect the simulator's intermediate state to the recursive acceptance definitions, ensuring that after processing i characters, the remaining computation on the suffix starting at position i will reach an accepting configuration if and only if the original input is in the language. Students are encouraged to use these simulators in assignments to test their machine constructions, stepping through computations to observe state transitions and debug incorrect designs.

In assignments, students construct FSMs for various languages specified in mathematical notation. For example, they build a DFSM accepting all strings over alphabet $\{a, b\}$ where every occurrence of a is immediately preceded and followed by b, or a DFSM for strings containing neither substring ab nor bb. They encode their machines as Dafny tuples, defining the state set, alphabet, transition function (using either named functions or lambda expressions), start state, and accepting states. After defining a machine, students may test it using the provided simulator on representative strings to verify acceptance and rejection behavior. Some assignments include optional proofs of correctness relating the machine's accepted language to the mathematical specification, which earn bonus points but are not required.

For NDFSMs, students work with examples involving epsilon transitions and nondeterminism. They manually trace computations to determine whether specific strings are accepted, sometimes writing detailed calculation-style proofs that step through the computation one transition at a time, invoking auxiliary lemmas to justify each step.

Regular expressions are formalized as an inductive datatype with constructors for the empty language, epsilon, single symbols, union, concatenation, and Kleene star. The language denoted by a regular expression is defined recursively as an infinite set. We present a constructive proof of Kleene's theorem showing that for every regular expression there exists an equivalent NDFSM, building the machine recursively by composing smaller machines corresponding to subexpressions. While the construction algorithm is fully implemented, most correctness proofs remain unverified, serving as examples of the correspondence between regular expressions and finite automata.

Students also encounter the pumping lemma for regular languages. The lemma states that if a language is regular, then there exists a pumping length k such that any string in the language of length at least k can be decomposed as xyz where $|xy| \leq k$, y is non-empty, and for all $q \geq 0$, the string $xy^q z$ is also in the language. We present a proof in Dafny that observes if a string's length exceeds the number of states, then during the computation some state must be visited twice, creating a loop that can be repeated or skipped. The proof relies on three auxiliary lemmas that remain unverified, serving as optional advanced exercises.

More importantly, students use the pumping lemma in proofs by contradiction to establish that certain languages are not regular. For example, to show that the language $\{a^n b^n | n \geq 0\}$ is not regular, we assume it is regular and apply the pumping lemma to obtain a pumping length k. We then consider the string $a^k b^k$, which is in the language and has length $2k \geq k$. The pumping lemma guarantees a decomposition xyz where $|xy| \leq k$, y is non-empty, and all pumped strings $xy^q z$ remain in the language. Since $|xy| \leq k$, the substring xy lies entirely within the a^k prefix, so y consists only of a's. Pumping to $q = 2$ yields $xyyz$, which has more a's than b's and thus cannot be in the language—a contradiction. This proof structure, with several unverified lemmas handling the technical details, appears in tutorials and assignments, teaching students how to apply the pumping lemma systematically to prove non-regularity.

Context-Free Grammars. Dafny datatypes are used for the formulation of context-free grammars. A grammar is represented as a tuple containing sequences of variables,

terminal symbols, production rules, and a start symbol. Variables are either nonterminals (identified by name) or terminals (individual symbols). A production rule pairs a nonterminal name with a sequence of variables representing its right-hand side. We define derivation steps as relations on sequences of variables: one sequence derives another if a nonterminal can be replaced according to a production rule. The language generated by a grammar is defined as an infinite set comprehension: the set of all terminal strings for which there exists a valid derivation sequence starting from the start symbol and ending with that string.

Rather than proving properties of grammars formally, students work with an executable derivation generator that we provide. This generator, written as an unverified Dafny method, performs a breadth-first exploration of all possible derivations up to a specified depth, printing each derivation step and collecting all generated terminal strings. The tool maintains a queue of partially-derived variable sequences, systematically applying production rules to nonterminals and tracking derivation depth to avoid infinite exploration. This allows students to test their grammar designs immediately and observe how different rule applications lead to different strings in the language.

In assignments, students construct context-free grammars for various languages specified mathematically. For example, they design a grammar for balanced parentheses involving multiple bracket types (curly braces, square brackets, and round parentheses), where strings must be properly nested. They also construct grammars for languages with arithmetic constraints, such as strings of the form $a^i b^j$ where $2i = 3j+1$, requiring careful analysis to ensure the grammar generates exactly the right ratio of symbols. Other assignments include languages where the count of one symbol is exactly twice the count of another (allowing arbitrary interleaving), and palindromes over a given alphabet. Students also study ambiguous grammars, such as a grammar for arithmetic expressions that allows multiple parse trees for the same string, exploring how different derivation orders can yield the same terminal string.

After defining each grammar as a Dafny tuple specifying its components, students run the simulator on their grammar with various depth limits, observing which strings are generated. They verify that expected strings appear in the output and that strings outside the language are not generated within the exploration depth. For instance, when testing a balanced parentheses grammar, students confirm that properly nested strings are derived while malformed strings with mismatched brackets do not appear. This hands-on experimentation reinforces the connection between formal grammar definitions and the concrete languages they generate, helping students develop intuition about how production rules combine to define complex syntactic patterns.

Pushdown Automata. A pushdown automaton, extending a finite state machine with a stack for unbounded memory, is formalized in Dafny as a tuple comprising states, input alphabet, stack alphabet, transition relation, start state, and accepting states. Each transition specifies a source state, an input symbol (or epsilon for non-consuming moves), a string to pop from the stack, a target state, and a string to push onto the stack. This design allows multiple symbols to be manipulated in a single transition, providing flexibility in modeling stack operations. A PDA configuration is a triple consisting of the current state, the remaining unread portion of the input string, and the current contents of the stack.

We provide a recursive nondeterministic simulator that explores possible computation paths through the PDA's state space. When multiple transitions are applicable from a configuration, the simulator systematically tries each option, backtracking when a path reaches a dead end (no applicable transitions, or the input is consumed but acceptance conditions are not met). The simulator accepts a string if any explored path leads to an accepting configuration, defined as being in an accepting state with both the input completely consumed and the stack empty. The recursive implementation maintains a history of previously visited configurations to detect and avoid cycles, though for some inputs the exploration could theoretically be non-terminating.

Students construct PDAs for various context-free languages in their assignments. These include the canonical example of strings with equal numbers of a's and b's in sequence ($a^n b^n$), balanced parentheses with multiple bracket types (curly braces, square brackets, and round parentheses requiring proper nesting), languages with arithmetic constraints between symbol counts (such as $a^i b^j$ where $2i = 3j + 1$), languages where one symbol appears exactly twice as often as another (allowing arbitrary interleaving), and palindromes over a given alphabet. For palindromes, students must design PDAs that use nondeterminism to guess the middle of the string, pushing symbols onto the stack during the first half and popping to verify mirror symmetry during the second half.

After encoding their PDAs as Dafny tuples, students may test them using the provided simulator, which prints detailed execution traces showing the sequence of configurations visited during computation. Each configuration display includes the current state, the remaining unread suffix of the input, and the current stack contents. For accepted strings, students observe the path from initial configuration to final accepting configuration. For rejected strings, they see where the computation gets stuck or fails to reach acceptance, helping them debug incorrect transition specifications. The type system ensures that PDAs are well-formed before simulation, catching errors such as undefined states, invalid symbols, or transitions referencing non-existent stack symbols.

Turing Machines and Decidability. Turing machines represent the culmination of computational models in the

course. Using Dafny, we formalize both the machine definition and its operational semantics, sticking to deterministic machines only. Our Turing machine comprises a set of states, an input alphabet, a tape alphabet (which includes a blank symbol), a transition function, a start state, and a set of halting states. The transition function maps each pair of non-halting state and tape symbol to a triple specifying the next state, the symbol to write, and the direction to move the read/write head (left or right).

A Turing machine configuration is represented as a quadruple consisting of the current state, the tape contents to the left of the head, the symbol currently under the head, and the tape contents to the right of the head. We enforce invariants that tape segments contain no leading or trailing blanks (except possibly the symbol under the head), ensuring a canonical representation that avoids redundant blank symbols at the tape extremes.

We provide an iterative simulator that repeatedly applies the transition function, updating the configuration at each step, until the machine reaches a halting state. The simulator begins with the input string on an otherwise blank tape, positions the head at the leftmost cell (containing a blank), and then executes transitions until halting. At each step, the simulator writes the specified symbol, moves the head left or right, and transitions to the new state. Special handling manages tape expansion when the head moves beyond current tape boundaries, automatically introducing blanks as needed. The simulator prints both the complete tape contents and the detailed configuration at each step, displaying the current state, tape left of the head, symbol under the head, and tape right of the head, making the computation trace fully visible.

Students study example Turing machines that perform various computational tasks. These include machines that recognize specific languages (such as verifying that a string has a particular pattern), machines that transform their input (such as replacing marked symbols with originals to reconstruct the input after verification), and machines that implement basic computations. Through detailed manual traces of these machines' execution on sample inputs, students observe how Turing machines use the tape both as input medium and as rewritable memory, enabling them to perform computations beyond the capabilities of finite automata or pushdown automata. The course concludes with discussions of decidability, the Church-Turing thesis, and the theoretical limits of computation, though these topics are explored conceptually rather than through formal development in Dafny.

Simulators and Visualization. The simulators for all three automaton types (FSMs, PDAs, and Turing machines) are implemented entirely in Dafny. These simulators provide a textual interface: when executed, they print the sequence of configurations visited during computation, displaying state transitions, stack contents (for PDAs), or tape modifications (for Turing machines) as lines of text output. Students run these simulators from the command line or within their development environment, observing the printed trace to understand how their machines process input strings. While effective for debugging and understanding computational behavior, this textual interface requires students to mentally reconstruct the computation flow from sequential text output.

To make the learning experience more interactive and visual, web-based visualization tools were recently developed as a complement to the Dafny simulators. Compiling the Dafny code to Python, the generated Python modules are used as the backend for these visualizers. The applications allow users to input their own machines (FSMs, PDAs, or Turing machines), specify test strings, and step through computations interactively with graphical displays. For FSMs, the visualizer shows the current state highlighted in a state diagram, with arrows indicating transitions as symbols are consumed. For PDAs and Turing machines, it displays both the current state and the stack/tape contents evolving step by step.

These visualization tools have yet to be used in the course but represent a promising aid for the future. They transform how users interact with automaton designs, providing immediate graphical feedback alongside the textual traces from the Dafny simulators. In lectures and tutorials, such visualizers might be helpful in demonstrating complex examples that would be tedious to trace by hand. The combination of Dafny implementations with interactive visualization bridges the gap between abstract formal definitions and concrete computational behavior.

Pedagogical Benefits and Reflections. The adoption of Dafny for teaching automata theory offers several significant advantages. Students receive immediate feedback on their constructions: syntax errors, type mismatches, and well-formedness violations are caught before execution, while runtime errors during simulation are quickly detected. This tight feedback loop accelerates learning. The executable nature of our programs makes abstract concepts concrete—students implement machines as code, run them on test inputs, and observe results. When a machine incorrectly accepts or rejects a string, students trace the computation to identify and fix errors. This bridges the gap between theory and practice.

The use of Dafny enables partial automation in grading. Submitted machines may be run against test suites. Machines passing all tests demonstrate significant correctness even without formal proofs. Additionally, Dafny's support for both ghost specifications and executable code creates a smooth continuum from high-level reasoning to concrete implementation.

From a technical perspective, Dafny's features align well with automata theory. Tuples represent machines as structured data, inductive datatypes model grammars and regular expressions, infinite set comprehensions define formal

languages, and ghost predicates capture non-executable concepts like language membership. The verifier checks that definitions are consistent and implementations respect contracts.

Challenges remain. Full formal verification of complex algorithms requires expertise beyond what we expect from second-year undergraduates. Many lemmas in our materials remain unverified, marked for future work or as optional advanced exercises. Nevertheless, even incomplete proofs serve pedagogical value by demonstrating proof structure and the kinds of properties one must establish.

The Dafny portion of the course material could be extended with additional examples, further proofs, and more algorithms. The visualization tools could be improved and tailored to become useful in lectures and tutorial sessions. The materials, which closely follow Rich's textbook while leveraging Dafny's unique combination of specification, verification, and code generation, have been made publicly available[1] to support other educators interested in teaching automata theory with modern verification tools.

[1] https://github.com/ranger71/automata-and-formal-languages