Automated Grading of Automata with ACL2s

Ankit Kumar, Andrew Walter & Panagiotis Manolios

Northeastern University

{ankitk,atwalter,pete}@ccs.neu.edu

Almost all Computer Science programs require students to take a course on the Theory of Computation (ToC) which covers various models of computation such as finite automata, push-down automata and Turing machines. ToC courses tend to give assignments that require paper-and-pencil solutions. Grading such assignments takes time, so students typically receive feedback for their solutions more than a week after they complete them. We present the Automatic Automata Checker (A2C), an open source library that enables one to construct executable automata using definitions that mimic those found in standard textbooks [26]. Such constructions are easy to reason about using semantic equivalence checks, properties and test cases. Instructors can conveniently specify solutions in the form of their own constructions. A2C can check for semantic equivalence between student and instructor solutions and can immediately generate actionable feedback, which helps students better understand the material. A2C can be downloaded and used locally by students as well as integrated into Learning Management Systems (LMS) like Gradescope to automatically grade student submissions and generate feedback. A2C is based on the ACL2s interactive theorem prover, which provides advanced methods for stating, proving and disproving properties. Since feedback is automatic, A2C can be deployed at scale and integrated into massively open online courses.

1 Introduction

In Theory of Computation (ToC) courses, students study models of computation, including Deterministic Finite Automata (DFAs), Push-Down Automata (PDAs) and Turing machines (TMs). When constructing automata, students benefit from getting immediate, automatic feedback. We present Automatic Automata Checker (A2C) [1], an open source library based on the ACL2s theorem prover [4, 12, 7]. For the purposes stated above, A2C provides convenient forms for defining, executing, testing and reasoning about automata.

Automata defined using A2C are not only executable, but are also a formal model of computation that can be reasoned about. Instructors can use ACL2s functionality to specify and check properties over these models, or use the full power of the ACL2s theorem prover to implement custom checks. This advantage is usually missing from visual or XML based representations, such as those used in Automata Tutor V3 [13] or JFLAP [23]. Furthermore, access to a powerful theorem prover allows an instructor to test their constructions, prove properties or even add new theories or models. A2C has built-in support for grading student-submitted automata and outputting results as JSON, making it easy to integrate into existing learning management systems. It comes with out-of-the-box support for Gradescope, a widely-used online grading system.

To appear in EPTCS.

When a student submits an automata construction to Gradescope, it is checked for consistency and semantic equivalence with solutions provided by the instructor. Students receive immediate feedback, *e.g.*, if their solution is incorrect, they are provided with inputs which their automata incorrectly classify. This is a significant improvement over existing tools like JFLAP [23] and OpenFLAP [21] which do not support personalized, high quality feedback. Based on the feedback they receive, students are then able to update and resubmit their automata.

A2C currently supports 3 models of computation – DFAs, PDAs and TMs, in a single package. Input validation, equivalence testing and property testing are supported across all three models. We provide a uniform testing interface across all supported models of computation, unlike Automata Tutor V3 or JFLAP, whose level of testing capabilities depend heavily on the kind of automata being checked.

A2C is built on top of ACL2s, which gives access to several useful features like a powerful macro system to define forms convenient for declaration of automata (our forms faithfully reflect their corresponding textbook [26] descriptions), the *Defdata* [10] framework to define automata components as types, theorem proving to prove equivalence of types, property based testing for checking equivalence between automata constructions and to test if such constructions satisfy certain properties and counter-example generation to produce helpful feedback, if automata constructions do not satisfy properties. Since A2C is open source, it can be easily extended to support other models or theories, for example, recursive function theory which has built-in support in ACL2s.

Our contributions. We make the following contributions with A2C: (i) an open source library to construct executable models of DFAs, PDAs and TMs, (ii) ability to reason about constructed automata using properties and test cases, (iii) instant feedback generation to guide students towards a correct construction, (iv) ability to run either locally or on compatible LMS like Gradescope and (v) ability to extend with more models of computation, with a little experience working with ACL2s. To the best of our knowledge, there exists no other tool with all of these capabilities.

Paper Outline. Section 2 presents related work in the field of automated grading of automata. Section 3 explains our choice of the ACL2s theorem prover. Section 4 illustrates the kinds of helpful feedback A2C generates on student submissions. Section 5 describes implementation considerations of the system. Section 6 shows the system architecture and details integration with Gradescope. Section 7 discusses our experiences using A2C to autograde assignments and exams. Section 8 lists some limitations of A2C and Section 9 concludes.

2 Related Work

Several tools and techniques for automatically grading or providing feedback for student submitted automata exist. For brevity, we will describe some of the most commonly used tools, and touch on the possibility of using specialized algorithms for automata equivalence checking.

The JFLAP (Java Formal Language and Automata Package) is a visualization and teaching tool for formal languages. It supports DFAs, PDAs, and TMs in addition to several other models of computation and parsing algorithms. JFLAP does not have built-in support for grading (as it is a tool intended for students to run themselves while writing up their homework solutions), though several extensions to JFLAP have been made that attempt to add such features [25, 22]. Both of these extensions only support DFAs, and the work of Shekhar *et al.* simply performs bounded checking of words to check equivalence between the student's submission and the instructor's solution. Neither of these extensions allow for

integration with an external LMS. JFLAP claims to be open-source, though we could not find any way to acquire the JFLAP source from its official website.

A more modern incarnation of FLAP called OpenFLAP is a component of OpenDSA [24], an open-source and interactive eTextbook system. OpenFLAP provides support for auto-graded DFA, PDA, and TM construction exercises, but student submissions are evaluated solely on whether they pass concrete test cases provided by the instructor. OpenDSA can be integrated with the Canvas LMS.

Automata Tutor v3 [13] is a closed-source online platform that automatically grades and provides feedback on automata. It has been used to teach thousands of students at over 30 universities, with high reported satisfaction among both students and instructors [13], which highlights the benefits of automated grading systems in undergraduate level theory of computation classes. It supports DFAs, PDAs and TMs in addition to several other models of computation. Instructors can create assignments inside of Automata Tutor that include automata construction exercises. Students then use Automata Tutor's interface to graphically construct the relevant automaton. Automata Tutor automatically grades exercises, and in the case of DFAs will provide "descriptive hints" that aim to help students understand how their solutions are incorrect [14]. For PDAs and TMs, Automata Tutor will only provide counterexamples (if it can find them). These counterexamples are generated by testing randomly generated words up to a configurable length, given certain resource limits [13]. A disadvantage of a closed-source monolithic platform like Automata Tutor v3 is that, it can not be reliably used when it is experiencing technical problems or is down for maintenance, as was the case when we last checked on 12^{th} November, 2022.

Checking equivalence of TMs or (nondeterministic) PDAs is undecidable, but checking the equivalence of DFAs is decidable and algorithms exist that can check equivalence and generate a witness (a counterexample to the equivalence of the two DFAs, e.g. a word that one of the DFAs accepts but the other rejects) with worst-case runtime complexity "nearly linear" in the total number of states in the two DFAs [22]. A complete procedure to check equivalence of DFAs would be a good future addition to A2C.

3 ACL2 Sedan

A2C is written in ACL2 Sedan (ACL2s) [4, 12, 7], which is an extension of A Computational Logic for Applicative Common Lisp (ACL2) [2, 15]. ACL2 is an industrial strength system for integrated modeling, simulation, and inductive reasoning. It comes from the Boyer-Moore family of theorem provers and is capable of reasoning about statements in the first order logic with mathematical induction. ACL2s extends ACL2 with automation and user friendly features like an advanced data definition framework Defdata, a powerful termination analysis based on calling context graphs [20] and ordinals [17, 18, 19], a property based modeling and reasoning framework for theorem proving, and the *cgen* framework [8, 9, 11] for generating counter-examples for invalid properties.

A2C utilizes all of these features to facilitate writing executable automata constructions, reasoning about them using properties, and getting helpful feedback in the form of counter-examples, all in the same system.

4 Illustrative Examples

Before describing the details of our system, we will walk through a few illustrative examples that highlight the kinds of feedback that A2C provides to a student. These example problems are also available

on Gradescope as homeworks, which can be accessed using instructions provided in [1].

4.1 Checking Deterministic Finite Automata

Consider a homework problem that requires a student to construct a DFA that can recognize words in $\{0,1\}^*$ consisting of an odd number of ones. A DFA M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is a start state and $F \subseteq Q$ is a finite set of accept states. Suppose the student submits an ACL2s form as shown:

```
(gen-dfa
 :name     student-dfa
 :states   (e1 e2 o1 o2)
 :alphabet (0)
 :start    e1
 :accept   (o1 o2)
 :transition-fun (((e1 0) . e1) ((e1 2) . o1) ((e2 0) . e2) ((e2 2) . e2)
                  ((o1 0) . o2) ((o1 2) . e2) ((o2 0) . o1) ((o2 2) . e1)))
```

`gen-dfa` is a macro that generates data definitions for a DFA construction. ACL2s has a powerful macro system that can be used to create representations of automata equivalent to those in the book. Motivated users define their own macros and syntax. The `gen-dfa` form above defines a DFA with the name `student-dfa`. The order of components appearing in the form does not matter, *e.g.*, swapping the order of `:states` and `:accept` yields the same DFA. On reading a `gen-dfa` form, A2C performs several checks, including ensuring (1) that the name is new, (2) that all components of a DFA are provided and (3) checking the consistency of each component, *e.g.*, that the transition function is a total function from the appropriate domain to the appropriate co-domain.

Given the form above, A2C reports that `:transition-fun` (δ) is not a function with domain $Q \times \Sigma$. This is because `:alphabet` (Σ) specified in the form consists only of `0`, whereas the alphabet in `:transition-fun` consists of both `0` and `2`. After receiving this feedback, the student can update `:alphabet` as follows.

```
:alphabet       (0 2)
```

The updated form passes all of our system's checks, indicating that the student's submitted form represents a valid DFA. What remains is to check whether this construction is correct. The specification for a correct automaton is a solution provided by the instructor:

```
(gen-dfa
 :name           instructor-dfa
 :states         (even odd)
 :alphabet       (0 1)
 :start          even
 :accept         (odd)
 :transition-fun (((even 0) . even) ((even 1) . odd) ((odd 0) . odd) ((odd 1) . even)))
```

Testing for correctness reduces to testing equivalence between the student's solution `student-dfa` and the instructor's solution `instructor-dfa`. Part of checking equivalence of automata includes checking for equivalence of alphabet. So, in the running example, A2C tries to prove the equivalence of both alphabets, fails and generates the following feedback:

```
Incorrect alphabet provided.
```

This is because the student's alphabet does not match with the one provided in the instructor's DFA. The student corrects their alphabet and updates their transition function as appropriate.

```
:alphabet (0 1)
:transition-fun (((e1 0) . e1) ((e1 1) . o1) ((e2 0) . e2) ((e2 1) . e2)
                 ((o1 0) . o2) ((o1 1) . e2) ((o2 0) . o1) ((o2 1) . e1))
```

But `student-dfa` is still incorrect, as is pointed out by A2C in the following feedback:

```
Transition function error. The following words are misclassified:
('(0 1 1 1) '(1 1 1 0) '(1 1 1))
```

Each list in the feedback represents a word that is misclassified by `student-dfa`. With this feedback, the student learns something new about their construction and can run their DFA on the generated words as shown:

```
(run-dfa student-dfa '(0 1 1 1))
```

The `run-dfa` form runs `student-dfa` on the input word `'(0 1 1 1)` generating the following output:

```
E2
```

This helps the student realize that any transition from `e2` leads back to `e2`. The student corrects this mistake as shown:

```
:transition-fun (((e1 0) . e1) ((e1 1) . o1) ((e2 0) . e2) ((e2 1) . o2)
                 ((o1 0) . o2) ((o1 1) . e2) ((o2 0) . o1) ((o2 1) . e1))
```

The updated solution is finally accepted by the autograder, which outputs:

```
student-dfa is correct.
```

4.2 Checking Push Down Automata

Consider another homework problem, one that requires a student to construct a a PDA that can recognize the language $\{0^n1^n | n \geq 0\}$. Recall that a PDA is a 6-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the input alphabet, Γ is the stack alphabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow \mathscr{P}(Q \times (\Gamma \cup \{\varepsilon\}))$ is a transition function, $q_0 \in Q$ is a start state and $F \subseteq Q$ is a finite set of accept states.

Consider a student's submission for this problem:

```
(gen-pda
 :name student-pda
 :states (q1 q2 q3)
 :alphabet (0 1)
 :stack-alphabet (0 z)
 :start-state q1
 :accept-states (q3)
 :transition-fun (((q1 0 :e) . ((q1 0)))
                  ((q1 1  0) . ((q2 :e)))
                  ((q2 1  0) . ((q2 :e)))
                  ((q2 :e z) . ((q3 :e)))))
```

The `gen-pda` form shown above defines a PDA with the name `student-pda`. Similar to the `gen-dfa` form described earlier, the order of components in this form does not matter. We use `:e` to represent the empty word ε. We could have chosen to use the unicode character for ε as well, but we did not want students to deal with potential issues due to lack of unicode support. Requiring only ASCII characters as input allows A2C to support online courses where students might use a plethora of development environments to write their solutions. Most of the checks related to validity of states, input alphabet and stack alphabet are similar to those for DFAs. Even though we do not require transitions from every possible tuple of

state, alphabet and stack symbol (we assume empty set by default), we do include a check for the presence of a transition from the start state on the empty word `:e`. This is required to add a base stack symbol. The submission shown above fails this test and hence the student receives the following feedback:

```
Starting transition from (Q1 :e :e) missing from the transition function.
```

This is fixed by adding a new start state q0 and updating the transition function as shown:

```
(gen-pda
 :name student-pda
 :states (q0 q1 q2 q3)
 :alphabet (0 1)
 :stack-alphabet (0 z)
 :start-state q0
 :accept-states (q3)
 :transition-fun (((q0 :e :e) . ((q1 z)))
                  ((q1 0  :e) . ((q1 0)))
                  ((q1 1   0) . ((q2 :e)))
                  ((q2 1   0) . ((q2 :e)))
                  ((q2 :e  z) . ((q3 :e)))))
```

The updated solution passes all checks for a valid PDA. It is now time to check whether it matches the specifications of the problem. After checking if the submitted PDA is correct, the autograder reports

```
Transition function error. The following words were misclassified :
(:e)
```

This feedback suggests that `student-pda` does not correctly classify the input word ε. Indeed, this word should be accepted, but is not accepted by `student-pda`. This can be fixed either by modifying the set of accept states:

```
:accept-states (q0 q3)
```

or by modifying the transition function to allow an ε transition from the start state to the accept state:

```
:transition-fun (((q0 :e :e) . ((q1 z) (q3 :e)))
                 ((q1 0  :e) . ((q1 0)))
                 ((q1 1   0) . ((q2 :e)))
                 ((q2 1   0) . ((q2 :e)))
                 ((q2 :e  z) . ((q3 :e))))
```

This updated solution is finally accepted upon submission.

```
student-pda is correct.
```

4.3 Checking TMs

Consider a problem where a student is required to submit a TM that flips 0s to 1s and vice-versa on its tape. A TM is defined as a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where Q is a finite set of states, Σ is the input alphabet, Γ is the tape alphabet, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, $q_0 \in Q$ is the start state, q_{accept} is the accept state and q_{reject} is the reject state. We require that the execution of the TM on the given input w end with the head at the end of the output on its tape. We ignore any symbols occurring to the right of the head after the end of execution. We also trim all blank symbols occurring between the end of the output and the head.

Suppose that a student submits the following TM for the above problem:

```
(gen-tm
 :name student-tm
 :states (q0 q1 q2 q3)
 :alphabet (0 1)
 :tape-alphabet (0 1)
 :start-state q0
 :accept-state q1
 :reject-state q2
 :transition-fun (((q0 1) . (q0 0 R))
                  ((q0 0) . (q0 1 L))
                  ((q0 nil) . (q3 nil R))
                  ((q3 nil) . (q1 nil L))))
```

The `gen-tm` form shown above defines a TM with the name `student-tm`. Similar to the previously seen `gen-` forms, the order of components in this form does not matter. After submitting this form, the following feedback is produced by the autograder:

```
Blank tape symbol nil missing from tape-alphabet.
```

The blank tape symbol is represented using the keyword `nil`. The `tape-alphabet` component of `gen-tm` is required to include the blank tape symbol `nil` and the input alphabet has to be a subset of the tape alphabet. This is fixed by modifying `tape-alphabet` :

```
:tape-alphabet (0 1 nil)
```

After editing and resubmitting, the autograder gives more feedback:

```
Incorrect output produced when running submitted TM on the following words :
('(1 0) '(0) '(0 1 1))
```

Running their TM on words provided in the feedback, the student realizes a mistake in their transition function, and corrects it as shown:

```
:transition-fun (((q0 1) . (q0 0 R))
                 ((q0 0) . (q0 1 R))
                 ((q0 nil) . (q3 nil R))
                 ((q3 nil) . (q1 nil L)))
```

The updated solution is finally accepted by the autograder.

5 Implementation

As already mentioned, our tool is based on the ACL2s theorem prover [12, 4], an extension of ACL2 [15, 2] which consists of a programming language, a logic for the language and an interactive theorem prover.

5.1 Solution format

A2C provides a well defined input format for specifying automata. A solution file in this format consists of one or more `gen-x` forms where *x* may be one of `dfa`, `pda` or `tm`. Each such form provides a declarative description of its corresponding automaton. The description is faithful to its textbook definition [26]. Hence the input format can be naturally explained and is easy to understand.

5.2 Validating automata

A2C validates a `gen-x` form by checking whether all of the following conditions hold:

- all required components of the automaton are provided,
- the start state is one of the given states,
- the set of accept states is a subset of the set of states,
- the domain of the transition function is of the right type,
- the co-domain of the transition function is of the right type,
- additional model-specific checks for each of PDAs and TMs, *e.g.*, the blank tape symbol does not appear in the alphabet of a TM, but should appear in the tape-alphabet.

5.3 Checking correctness

A2C uses the defdata data definition framework to convert an automaton description into corresponding definitions of states, alphabets, transition-functions and functions which make it executable.

For property based testing, instructors can either use `property` forms (for testing as well as theorem proving) or `test?` forms (meant solely for testing). Property based testing depends on the cgen framework for generating counter-examples to invalid properties. A2C makes use of the `interface` [27] library to query ACL2s with `test?` forms to test equivalences and extract counter-examples.

Cgen framework: Property based testing using ACL2s' `test?` forms depends on cgen, a counter-example generation facility which combines random testing with theorem proving in a synergistic fashion. On the one hand, it uses the full power of the theorem prover to simplify a given conjecture formula, which makes finding a counter-example easier. On the other hand, if random testing reveals a generalization of the given formula to be false, it forces the theorem prover to backtrack which enhances theorem proving. The number of tests generated for property based testing is configurable. For use in our undergraduate-level ToC class, we used the default number of test cases for `test?` forms, which is 1000, and are not aware of any false-positives.

Testing equivalence: To test if two automata are equivalent, we first check if their alphabets match:

```
(defdata-equal instructor-x-alphabet student-x-alphabet)
```

Here `instructor-x-alphabet` and `student-x-alphabet` are definitions generated by either of `gen-dfa`, `gen-pda` or `gen-tm`, (depending on `x` being either of `dfa`, `pda` or `tm`) for the alphabet of the instructor and the student automata respectively. `defdata-equal` checks whether two data types are identical. If not, the defdata framework [10] will generate a value that is in one of the data types and not in the other. In this case, if the student and instructor alphabets differ, defdata will return a symbol that is in one alphabet but not the other. To complete the check of equivalence, we check the following property.

```
(test?
  (=> (instructor-x-wordp w)
      (== (accept-x w *student-x*)
          (accept-x w *instructor-x*))))
```

where `x` is one of `dfa, pda` or `tm`.

Note that whenever a `gen-x` form successfully validates and generates data definitions for each component of a given construction named `student-x`, behind the scenes, it generates a constant `*student-x*` to refer to a list consisting of the given components. The (`accept-x *student-x* w`) and (`accept-x *instructor-x* w`) forms check whether the student's automaton and the instructor's automaton accept the word `w` of type `instructor-x-word` respectively. Any counter-example generated by this `test?` form indicates a word accepted by exactly one of the automata. The number of steps PDAs and TMs are allowed to run for on an input is bounded and can be configured by the instructor. In case of TMs, we utilize another test for equivalence, one that checks whether the two TMs agree on their output.

```
(test?
   (=> (instructor-tm-wordp w)
       (== (remove-final-nils (left-of-head (run-tm w *student-tm*)))
           (remove-final-nils (left-of-head (run-tm w *instructor-tm*))))))
```

where `left-of-head` of the output of `run-tm` is the part of the tape to the left of the head, where we expect our output to reside. `remove-final-nils` removes blank symbols occurring between the end of the output and the head.

Automata constructions are executed using functions, whose definitions depend on the automaton `x`:

- Given a initial state q_0, a transition function δ and a word w, `run-dfa` returns a state $s \in Q$. Starting with q_0 and the first letter in w, it queries δ to get a new state. This process is repeated until all letters in w are exhausted and s is reached.

  Since δ is complete (all pairs in $Q \times \Sigma$ belong to the domain of δ), and since the length of w is finite, `run-dfa` is terminating.

- Consider a tuple $t \in Q \times \Gamma^* \times \Sigma^*$ denoting a state reached, contents in a stack and a suffix of the input word left to be consumed, respectively. We call this an exec-tuple.

  `run-pda` is a function that accepts a bound (a natural number), a PDA and a set of exec-tuples, and returns a boolean: t or nil. The execution trace of `run-pda` is a tree such that each of its nodes is an exec-tuple. The root of this tree is (q_0, ε, w) *i.e.*, a tuple consisting of the start state, an empty stack and the input word. At each step of the execution, new leaves of the tree may be generated. Recall that $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathscr{P}(Q \times (\Gamma \cup \{\varepsilon\}))$. For a leaf $l = (q_l, (t \ldots), (c \ldots))$ where $q_l \in Q$, t is the top of its stack and c is the first letter in the rest of the word, we have the following tuples : $(q_l, t, c), (q_l, \varepsilon, c), (q_l, t, \varepsilon)$ and $(q_l, \varepsilon, \varepsilon)$. If any of these tuples exists in the domain of δ, we say that leaf l is *active* and the children of l can be generated using δ. In case none of these tuples exist in the domain of δ, l remains a leaf node.

  An exec-tuple is accepted when it is of the form $(q_f, \ldots, \varepsilon)$ where $q_f \in F$. `run-pda` is executed until either an acceptable exec-tuple is generated (in which case it returns t) or when all active leaves of the tree are at a depth greater than n, in which case it returns nil.

  The inherent non-determinism of PDAs coupled with the possibility of making $\varepsilon-$transitions does not allow guaranteed termination of `run-pda`. Hence, the execution of `run-pda` needs to be bounded by a positive integer n, the maximum depth of an execution tree decided by the instructor.

- Given a TM $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ and a word w, `run-tm` returns a tuple $(Q \times \Gamma^* \times \Gamma^*)$ whose first element is the current state of the TM and the second and third elements are the contents of the tape to the left and to the right of the TM's head, respectively. Note that tape symbols to the left of the head are stored in reverse. The head of a TM moving left on its tape is simulated by

removing the first element from the tape on the left of head and attaching it at the start of the tape on the right of head.

Since the halting problem is undecidable for a Turing machine, we need to set a maximum number of steps the TM can run for before stopping it.

Testing properties: Since checking the equivalence of PDAs and TMs is undecidable, it is useful for the instructor to have a collection of properties to check a students' constructions. For example, in context of our running DFA example,

```
(property no-odd1s-in-ww (w :instructor-dfa-word)
  :proofs? nil
  (! (accept-dfa *student-dfa* (append w w))))
```

checks that *`student-dfa`* does not accept words of the form `ww`. This is because such words can not have an odd number of `1`s. Such properties represent an infinite class of test cases a submission can be evaluated against. Of course, ground expressions such as unit tests can also be defined. Property forms in ACL2s can be configured to perform as much testing as one requires. In our example, we turn off theorem proving locally for this property using `:proofs? nil` to focus exclusively on testing in hopes of finding counter-examples. It is possible to disable theorem proving globally, for all properties as well, as shown:

```
(set-acl2s-property-table-testing? t)
(set-acl2s-property-table-proofs? nil)
```

Now that theorem proving has been disabled and testing enabled globally for property forms, we have

```
(property accept-w->accept-0w1 (w :instructor-pda-word)
  (=> (accept-pda *student-pda* w)
      (accept-pda *student-pda* (app '(0) w '(1)))))
```

which checks that if a word w is accepted by *`student-pda`*, so is $0w1$.

Finally, for the TM example, we could use the following property to test a student submission, after it has been validated:

```
(property involution (w :student-tm-word)
  (== (remove-final-nils (left-of-head (run-tm
        (remove-final-nils (left-of-head (run-tm w *student-tm*))))))
      w))
```

which checks that if running *`student-tm`* on the input word w produces w' in which the 0s and 1s are flipped, then running *`student-tm`* again on w' should produce w as output.

Notice that each property is specified programmatically in the ACL2s language in terms of executable automata, making the process of creating tests effortless.

Unit testing: An instructor can also make use of `check=` forms to check their own construction for specific test cases. For example, from our running examples,

```
(check= (accept-pda *instructor-pda* '(0 0 0 1 1 1)) t)
```

checks whether `*instructor-pda*` accepts `'(0 0 0 1 1 1)`. Similarly,

```
(check= (remove-final-nils (left-of-head (run-tm '(1 0 1 1 1 0 1 0) *instructor-tm*)))
   '(0 1 0 0 0 1 0 1))
```

checks whether running `*instructor-tm*` on input `'(1 0 1 1 1 0 1 0 1 0)` does actually flip 0s to 1s and vice-versa.

6 System Description

A2C can be packaged as an executable and deployed on any online grading platform that can run the executable on student submissions and can use the output generated by our tool to grade assignments. It is written inside of ACL2s, making it easy to extend for anyone with working knowledge of ACL2s.

6.1 ACL2s and external tools

As previously discussed in Section 3, ACL2s extends the ACL2 theorem prover with several automation and user-friendly features. ACL2 provides a full-featured programming language, so it is possible to implement and integrate additional functionality inside of ACL2 without needing to modify or understand its internals. ACL2 is built on top of Common Lisp, and is designed in such a way that one can write Common Lisp code that interacts with ACL2 code and vice versa. This means that it is also relatively easy to integrate external tools with ACL2.

ACL2s provides a feature that allows a user to save the state of a running ACL2s process as an executable file. We use this feature to generate an executable for each assignment that an instructor would like to autograde. The state of such an executable includes libraries that we wrote that provide the forms required for testing student constructions and integrating with Gradescope in addition to the instructor's version of any automata being graded.

6.2 Libraries

Our Gradescope-based automatic grader for the ToC course uses the following libraries:

- `A2C`: forms needed to define, validate, run and check equivalence between user-provided automata
- `gradescope-acl2s` : interacts with Gradescope by generating JSON files consisting of scores and feedback for autograded submissions.
- `interface` : provides an interface with ACL2s' theorem proving and counterexample generation functionality.

These libraries are designed to be reusable in other contexts besides A2C. Students can use the A2C library locally to inspect their automata before submitting. For example, a student who is so inclined can define an automata and then use ACL2s to check both concrete test cases and properties that they believe their automata should satisfy. Instructors may also find this useful to check their work while developing solution automata. Defining properties requires some experience working with the ACL2s theorem proving system. Freshman computer science students generally learn ACL2s as part of Northeastern University's Logic and Computation class.

Figure 1: Technical stack for autograding ToC assignments

6.3 Gradescope Integration

Gradescope is a LMS used at several universities. It provides autograding functionality, and can be configured to allow students to submit the same assignment multiple times and review autograder feedback after each submission.

A Gradescope autograder is a zip-archive consisting of code that accepts student submissions and generates JSON files consisting of points received and feedback for each autograded solution. Student submissions are run in Docker containers, which are setup according to specifications provided in the autograder zip-archive. Since Gradescope Docker containers do not support ACL2s by default, we use a custom Docker image [3]. In most cases, an instructor can simply use one of our provided examples, just updating the problems and solutions as necessary, to publish an autograded assignment.

Our autograder can be adapted to support any LMS that supports autograding using a Docker image or an executable. To do this, one would need to write a library analogous to `gradescope-acl2s` that handles whatever I/O is appropriate for that LMS.

7 Experiences and Observations

We deployed A2C on Gradescope for autograding assignments in a undergraduate-level ToC class taught at Northeastern University, consisting of about 50 students. Students were introduced to the input format for specifying each automata before the corresponding assignment was released.

7.1 Ease of use

Setting up autograded assignments using A2C was easy. Solutions were easy to specify, since they were simply the instructor's solution. A portion of the final exam consisted of autograded problems. For example, a question similar to : "Write a TM to insert a 1 in the input tape after the third occurrence of symbol # from the left" was asked in the final exam. TMs submitted for this problem were fairly complex (*i.e.* they had several states and transitions). Autograding helped us save about 16 hours of TA time grading this problem. In addition to this, usually one in thirty students would have sent regrade requests, which would have taken another 2 hours. Hence it was useful for the students as well.

7.2 Anecdotal evidence

We observed that in comparison to manually graded assignments: (1) autograded assignments had a significantly higher number of resubmissions, (2) students received higher grades in the autograded assignments and (3) student feedback regarding the autograding was overwhelmingly positive due to the immediate feedback which allowed students to find trivial errors and helped them better understand the course material. On average, more than 95% of students got full credit on autograded problems, whereas less than 20% got full credit on manually graded problems.

8 Limitations

Our approach has few limitations, which we discuss here.

Firstly, our tool does not provide a graphical interface for constructing or inspecting automata. Although the format we use to submit automata is similar to that used in Sipser's classic textbook [26], students may still need to learn how to read and write S-expressions to write automata descriptions. This was not an issue in our class since most students had already taken the "Logic and Computation" class, where they were introduced to programming in ACL2s. However, in case they do not understand S-expressions, students may require some additional training before they can use our tool.

Adding visualizations of automata to our tool would be relatively straightforward - since our tool is written in ACL2s, it is easy to develop code that will produce output suitable for visualization tools. Adding a UI for constructing automata may be somewhat more difficult and will not integrate with Gradescope.

Though A2C does not require its users to have any ACL2s experience, some functionality does require ACL2s experience. In particular, writing an extension of our tool requires the user to write ACL2s functions. Users may also need to understand ACL2s to some extent to make use of the property system.

Additionally, even in cases where the automata equivalence problem is decidable, the method we use to check for automata equivalence is not complete, which is a result of having a uniform testing interface across all supported models of computation. Put another way, our tool may return false positives, e.g. report that two automata are equivalent when they are not. While using our tool in a Theory of Computation course, we performed spot checks of our tool's results and did not find any issues. It may be the case that the mistakes that students make tend to be easy to exploit – that is, they are easy to find counterexamples. We expect that more complex automata and automata that differ in their classification of a small number of words would be more prone to false positives.

To mitigate the issue of incompleteness, an instructor using our tool can provide both test cases and properties to check tricky errors that the automata equivalence checker might miss. If they have some ACL2s expertise, the instructor can use a variety of techniques to improve ACL2s' ability to find witnesses to non-equivalence. However, adding more properties should be sufficient in most cases.

Overall, we believe that trading off incompleteness for ease of use is worthwhile, as this allows instructors and TAs to spend more time working with students.

9 Conclusion and Future work

We present A2C, a library based on the ACL2s theorem prover to check and provide automatic feedback to students about automata they construct. Such constructions are executable and generate a formal model to reason about in the context of a theorem prover, by checking for equivalence with the instructor's construction, testing properties and test cases. It generates immediate and helpful feedback. A2C can either be run locally or on a compatible LMS. It has been integrated into Gradescope to grade assignments and exams. All of these abilities make A2C ideal for use in massively open online courses, where instructor-student ratio is generally too low to allow highly available, high quality and personalized feedback by instructors. We believe that a similar approach of theorem prover based feedback generation can be used for non-programmable assignments in other subjects like logic and discrete math.

In the future, we would like to add the capability to generate visualizations of automata in order to make learning more intuitive and effective for students. We would also like to add decision procedures for decidable problems, like checking equivalence of DFAs. However, this is an example of an extension

which can be implemented by end users as well, due to A2C being open source.

Acknowledgements

We sincerely thank Mirek Riedewald and Jason Hemann, the instructors for the Theory of Computation course at Northeastern University in Fall 2020 and Fall 2021 respectively.

References

[1] *A2C : Automatic Automata Checker*. https://github.com/ankitku/A2C.

[2] *ACL2 web-page*. https://www.cs.utexas.edu/users/moore/acl2/.

[3] *ACL2s docker image*. https://hub.docker.com/r/atwalter/acl2s_gradescope_autograder/tags.

[4] *ACL2s web-page*. http://acl2s.ccs.neu.edu/acl2s/doc/.

[5] *Gradescope*. https://www.gradescope.com.

[6] Rajeev Alur, Loris D'Antoni, Sumit Gulwani & Dileep Kini (2013): *Automated Grading of DFA constructions*. In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Available at http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6759.

[7] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon (2011): *The "ACL2" Sedan Theorem Proving System*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, doi:10.1007/978-3-642-19835-9_27.

[8] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University.

[9] Harsh Raju Chamarthi, Dillinger Peter C., Matt Kaufmann & Panagiotis Manolios (2011): *Integrating testing and interactive theorem proving*. In: *International Workshop on the ACL2 Theorem Prover and its Applications (ACL2)*, doi:10.4204/EPTCS.70.1.

[10] Harsh Raju Chamarthi, Dillinger Peter C. & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. In: *International Workshop on the ACL2 Theorem Prover and its Applications (ACL2)*, doi:10.4204/EPTCS.152.3.

[11] Harsh Raju Chamarthi & Panagiotis Manolios (2011): *Automated specification analysis using an interactive theorem prover*. In: *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Available at http://dl.acm.org/citation.cfm?id=2157665.

[12] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *International Conference on Software Engineering (ICSE)*, doi:10.1016/j.entcs.2006.09.018.

[13] Loris D'Antoni, Martin Helfrich, Jan Kretinsky, Emanuel Ramneantu & Maximilian Weininger (2020): *Automata Tutor v3*. In: *International Conference on Computer Aided Verification (CAV)*, doi:10.1007/978-3-030-53291-8_1.

[14] Loris D'Antoni, Dileep Kini, Rajeev Alur, Sumit Gulwani, Mahesh Viswanathan & Björn Hartmann (2015): *How Can Automatic Feedback Help Students Construct Automata?* In: *ACM Transactions on Computer-Human Interaction (TOCHI)*, doi:10.1145/2723163.

[15] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2012): *Computer-Aided Reasoning: An Approach*. Springer US. Available at https://link.springer.com/book/10.1007/978-1-4615-4449-4.

[16] Matt Kaufmann & J Strother Moore: *ACL2: An industrial strength version of Nqthm*. In: *Conference on Computer Assurance (COMPASS)*, doi:10.1109/CMPASS.1996.507872.

[17] Panagiotis Manolios & Daron Vroon (2003): *Algorithms for Ordinal Arithmetic*. In: *19th International Conference on Automated Deduction (CADE)*, doi:10.1007/978-3-540-45085-6_19.

[18] Panagiotis Manolios & Daron Vroon (2004): *Integrating Reasoning about Ordinal Arithmetic into ACL2*. In: *Formal Methods in Computer-Aided Design (FMCAD)*, doi:10.1007/978-3-540-30494-4_7.

[19] Panagiotis Manolios & Daron Vroon (2005): *Ordinal Arithmetic: Algorithms and Mechanization*. *Journal of Automated Reasoning*, doi:10.1007/s10817-005-9023-9.

[20] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In: *Computer Aided Verification (CAV)*, doi:10.1007/11817963_36.

[21] Mostafa Mohammed, Clifford A. Shaffer & Susan H. Rodger (2021): *Teaching Formal Languages with Visualizations and Auto-Graded Exercises*. In: *ACM Technical Symposium on Computer Science Education (SIGCSE TS)*, doi:10.1145/3408877.3432398.

[22] Daphne Norton (2009): *Algorithms for testing equivalence of finite automata, with a grading tool for JFLAP*. Available at `https://scholarworks.rit.edu/theses/6939`.

[23] Susan H Rodger & Thomas W Finley (2006): *JFLAP: an interactive formal languages and automata package*, Computer Scienc ed. edition. Jones and Bartlett Publishers, Inc. Available at `https://www.jflap.org/jflapbook/jflapbook2006.pdf`.

[24] Clifford A. Shaffer (2016): *OpenDSA: An Interactive ETextbook for Computer Science Courses*. In: *ACM Technical Symposium on Computing Science Education (SIGCSE TS)*, doi:10.1145/2839509.2850505.

[25] Vinay S. Shekhar, Anant Agarwalla, Akshay Agarwal, B. Nitish & Viraj Kumar (2014): *Enhancing JFLAP with automata construction problems and automated feedback*. In: *International Conference on Contemporary Computing (IC3)*, doi:10.1109/IC3.2014.6897141.

[26] Michael Sipser (2013): *Introduction to the Theory of Computation*, Third edition. Course Technology.

[27] Andrew T. Walter & Panagiotis Manolios (2022): *ACL2s Systems Programming*. In: *International Workshop on the ACL2 Theorem Prover and its Applications (ACL2)*, doi:10.4204/EPTCS.359.12.