

Multitape Turing Machine Design

Extended Abstract

Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
morazanm@shu.edu

A hallmark of any Formal Languages and Automata Theory (FLAT) course is the study of Turing machines (**tms**). Almost invariably, students are able to understand that **tms** are powerful enough to decide languages that are not context-free and to compute arbitrary functions. Students also realize that these machines are difficult to design and, in a programming-based course, to implement due to the low-level of abstraction they provide. One of Alan Perlis' epigrams in programming directly speaks to this: *"Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy."* In a FLAT course, this is a problem for students because they are marked down for buggy machines leading to apathy for the material and it is a problem for instructors that are often unable to tease out student designs to offer useful feedback.

Some of these issues are addressed by adopting a design-based programming methodology in a FLAT course. To support this goal, the domain-specific language **FSM** (**F**unctional **S**tate **M**achines) has been developed [3]. **FSM** is embedded in **Racket** [1] and provides the programmer with the ability to design, define, test, and visualize the execution of state-based machines [4]. This language-based framework provides students and instructors a *lingua franca* to discuss **tm** design and implementation. In addition, it makes student training in programming more relevant in a FLAT course.

As useful as **FSM** is to spark CS-student interest and to help instructors understand a design, the design and implementation of **tms** remains a difficult error-prone exercise. To make the design and implementation process easier, many textbooks introduce Turing machine extensions [2,5,6]. Chief among these is extending **tms** with multiple tapes. Such machines are coined multitape Turing machines (**mttm**). Computationally, **mttms** are equivalent to **tms**. Nonetheless, it is important for students to understand, design, and implement such machines given that they make deciding some languages easier. Understanding and being familiar with such machines is also pivotal to more easily understand, for example, the equivalence between context-sensitive grammars and **tms**.

To aid students in understanding **mttms**, **FSM** has been extended to provide the necessary language infrastructure to implement them. A constructor is provided that has the following signature:

```
(listof state) alphabet state (listof state)
(listof mttm-rules) natnum [state]           → mttm
```

The first and second arguments, respectively, denote the machine states and the input alphabet. The third argument is the machine's starting state while the

1. Name the machine and specify alphabets
2. Write unit tests
3. Identify conditions that must be tracked as input is consumed, associate a state with each condition, and determine the start and final states.
4. Formulate the transition relation
5. Implement the machine
6. Test the machine using unit tests and random testing
7. Design, implement, and test an invariant predicate for each state
8. Prove $L = L(M)$

Fig. 1: The design recipe for state machines.

fourth argument if the list of final states. The fifth argument denotes the transition relation. The sixth argument is for the number of types. The final optional argument denotes the accept state among the final states. This optional argument is only provided if the `mttm` constructed decides a language. Each rule in the relation has the following structure: `((list state (listof symbol)) (list state (listof action)))`. The first pair contains the machine's current state and a list of the symbols read on each tape. The second pair contains the state the machine moves to and the list of actions taken on each tape. An action either writes an element to the tape, moves the head left, or moves the head right. To apply an `mttm`, programmers may use `sm-apply` and `sm-showtransitions`. Both take as input an `mttm`, the main tape's initial value, and the main tape's initial head position. The first returns the final machine configuration after execution. The second returns a trace of all the machine configurations during a computation. In addition, programmers may dynamically visualize a `mttm`'s execution using `sm-visualize`.

A design recipe for a state machine is presented to students to help guide machine development and implementation. The design recipe for state machines is displayed in Figure 1. Step 1 asks to pick a descriptive name for the machine and to define the needed alphabets (e.g., an `mttm`'s input alphabet). Step 2 asks for the development of a thorough set of unit tests. Step 3 asks for the conditions that must be tracked as the machine consumes a word. Each condition is associated with a state. For instance, the conditions associated with an `mttm` describe properties of the content of all tapes and, possibly, the position of each head. In this step, students must clearly annotate the start and final/accepting states. Step 4, based on the result of step 3, asks for the development of the transition relation. Each transition is developed assuming that the condition describing the source state holds and that the actions taken on each tape make the conditions of the destination state hold. Steps 5 and 6 ask for the machine's implementation and the running of tests. Step 7 asks for the development of state invariant predicates. A state invariant predicate for a `mttm` takes as input the tape values and the head positions and verifies that the conditions the state represents hold. For machines that decide a language, step 8 asks for the development of a proof demonstrating that the machine's language, $L(M)$, is the same as the language,

L , the machine is designed to decide. This is done by first proving by induction that the state invariants hold when an arbitrary word is processed. This proof is then used to prove that $L = L(M)$.

This talk discusses how students are introduced to `mtms` in a classroom setting. It highlights all the characteristics outlined above from interfacing with `mtms` to their design, implementation, and proof of partial correctness. The talk presents an extended development example and how execution visualization is used before attempting to prove machine correctness. The result is an integrated design-based programming approach that teaches students the same material found in classical automata theory textbooks.

References

1. Flatt, M., PLT: The Racket Reference, <https://docs.racket-lang.org/reference/index.html>, last accessed: November 2023
2. Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edn. (1997). <https://doi.org/10.1145/300307.1040360>
3. Morazán, M.T., Antunez, R.: Functional automata-formal languages for computer science students. In: Caldwell, J.L., Hölzenspies, P.K.F., Achten, P. (eds.) Proceedings 3rd International Workshop on Trends in Functional Programming in Education, TFPIE 2014, Soesterberg, The Netherlands, 25th May 2014. EPTCS, vol. 170, pp. 19–32 (2014). <https://doi.org/10.4204/EPTCS.170.2>, <https://doi.org/10.4204/EPTCS.170.2>
4. Morazán, M.T., Schappel, J.M., Mahashabde, S.: Visual designing and debugging of deterministic finite-state machines in FSM. Electronic Proceedings in Theoretical Computer Science **321**, 55–77 (aug 2020). <https://doi.org/10.4204/eptcs.321.4>
5. Rich, E.: Automata, Computability and Complexity: Theory and Applications. Pearson Prentice Hall (2019)
6. Sipser, M.: Introduction to the Theory of Computation. Cengage Learning, 3rd edn. (2013)