

Visualizing Composed Turing Machines in FSM

Draft Student Paper

Tijana Minić and Oliwia Kempinski and Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
{minictij|kempinol|morazanm}@shu.edu

Abstract. Students find their first course in Formal Languages and Automata Theory challenging. In addition to the development of formal arguments, most students struggle to understand how to compose Turing machines. That is, they struggle with the idea that a Turing machine may make use of auxiliary Turing machines. This article presents two visualization tools developed to help students understand Turing machine composition. The tool is integrated into, **FSM**, a domain-specific functional language for the Automata Theory classroom. One first tool is static and generates a transition diagram for composed Turing machines. The second tool is interactive and simulates the execution of composed Turing machines.

1 Introduction

Most, if not all, Formal Languages and Automata Theory (**FLAT**) courses expose students to Turing machines (**tms**). As is eloquently stated in one of Alan Perlis' epigrams in programming: *Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy*. This warning is especially relevant for students that are being exposed for the first time to Turing machines. Designing a **tm** on pencil and paper is akin to programming in assembly. That is, students work at a very low-level of abstraction and have no debugging nor unit testing facilities. In addition, students are rarely exposed to design principles that can help manage the complexity of creating **tms** (on paper). In an attempt to address some of these problems, students are also introduced to Turing machine composition (again, mostly on pencil and paper). A composed **tm** (**ctm**) uses auxiliary **tms** to solve a problem much like programmers use auxiliary functions. Although this may allow students to think more abstractly (i.e., a **tm** is a black box that correctly performs a task), the problems of interacting, testing, and debugging remain. Without a doubt, this goes against the grain of a programming-based Computer Science education and in many cases leads to apathy for the material.

To address these shortcomings **FSM** (**F**unctional **S**tate **M**achines)—a domain specific functional language for the Automata Theory classroom embedded in **Racket**—has been developed. In **FSM**, students may design, implement, and test state machines (deterministic/nondeterministic finite-state automata, pushdown automata, Turing machines, and multitape Turing machines), grammars (regular, context-free, and context-sensitive), and regular expressions. In this manner,

the students' programming training is made more relevant, which makes the material more palatable. In addition, embedded in **FSM** is a domain specific language to design, implement, and test **ctms**. **FSM** provides primitives to execute and to generate a trace of machine configurations reached during a computation. In this manner, students have tools to debug their **ctms** before submitting them for grading. We note that it is important for students to understand **ctms** because they form the basis for discussing the Church-Turing thesis and for the modern Von Neumann architecture.

Despite the facilities provided by **FSM**, students still find it difficult to design and implement **ctms**. Simply stated, it is too difficult to mentally keep track of the state a **ctm** ought to be in during execution. To address this shortcoming, we have developed two **ctm** visualization tools. The first is a static tool that provides the programmer with a graphic for a given **ctm**'s transition diagram. The second is a dynamic tool that allows the programmer to visualize the stepwise execution of a given **ctm**. In the dynamic visualization tool, a step is the execution of a **tm**. This extended abstract is organized as follows. Section 2 presents a brief introduction to Turing machines and **ctms** in **FSM**. Section 3 presents the interfaces and examples of both the static and the dynamic visualization tools. Section 4 outlines the implementation of our visualization tools. Finally, Section 7 presents concluding remarks and directions for future work.

2 Brief Introduction to **FSM**

2.1 Turing Machines

In **FSM**, a Turing machine is an instance of:

`(make-tm κ Σ R S F [Y])`

κ is a set of states, Σ is an input alphabet, **R** is a transition relation, $S \in \kappa$ is the starting state, $F \subseteq \kappa$ is the set of final states, and the optional argument $Y \in F$ is the *accepting* final state for a Turing machine that decides or semidecides a language. The members of **R** are tuples of two tuples: $((\kappa \ \Sigma) (\kappa \ A))$, where **A** is an action. Such a rule states that if the machine is in the first tuple's state and the given alphabet member is read then the machine moves to the state in the second tuple and performs the given action. An action is either **RIGHT**, to move the head right, **LEFT** to move the head left, or an element of Σ to mutate the read element to the given element. Finally, we note that a **tm** halts when it reaches a final state.

To illustrate the implementation of basic **tms**, consider the definitions in Figure 1. For our current purposes, an input word, **w**, is an instance of `(a b BLANK)*`. Let us start with the **tm** in Figure 1a to move the head to the right. The first line indicates that the programming language used is **FSM**. Next, there are two comment lines defining the precondition and the postcondition for the machine. The precondition states that the expected initial configuration is a tape that contains the left-end marker, **LM**, followed by, **w**, the input word. In addition, it states that

<pre> #lang fsm ;; PRE: tape=(LM w) && i=k>0 ;; POST: tape=(LM w) && i=k+1 (define R (make-tm '(S F) '(a b) (list (list (list 'S 'a) (list 'F RIGHT)) (list (list 'S 'b) (list 'F RIGHT)) (list (list 'S BLANK) (list 'F RIGHT))) 'S '(F))) </pre>	<pre> #lang fsm ;; PRE: tape = (LM w) && i=k>0 ;; && tape[i]=s ;; POST: tape = (LM w) && i=k ;; && tape[i]=BLANK (define WB (make-tm '(S H) (list 'a 'b LM) (list (list (list 'S 'a) (list 'H BLANK)) (list (list 'S 'b) (list 'H BLANK)) (list (list 'S BLANK) (list 'H BLANK))) 'S '(H))) </pre>
(a) Move right Turing machine.	(b) Write a blank Turing machine.

Fig. 1: Basic Turing machines.

the head position, i , is a natural number k greater than 0. The postcondition states that the tape is unchanged and the position of the head is incremented by 1. The next lines define a Turing machine named `R`. The transition relation, for all possible elements read, moves the machine from the starting state to the only final state, moves the head to the right, and halts. Finally, the tests use FSM's `sm-showtransitions` to validate the expected computation. This function consumes the machine to execute, the initial state of the tape, and the initial head position and returns a list of the configurations the machine reaches during computation. As the reader may verify, for all tests the computations only take one step from `S` to `F`, leave the tape unchanged, and increment the position of the head by 1.

Figure 1b displays the `tm` to write an blank to the tape. The precondition abstracts the value under the head as `s` and the postcondition states that the head position remains unchanged, but the value under the head is mutated to a blank. The transition relation, for all possible elements read, moves the machine from the starting state to the only final state, writes a blank to the tape, and halts. Finally, the unit tests, instead of validating the entire computation, validate the final configuration of the machine.

FSM includes visualization tools for Turing machines [1]. The transition diagram for a `tm` is obtained by applying `sm-graph`. The transition diagrams obtained for the machines defined in Figure 1 are displayed in Figure 2. In addition, there is a dynamic visualization tool for Turing machine execution. In the interest of brevity, the reader is referred to a previous publication, [1], for its description.

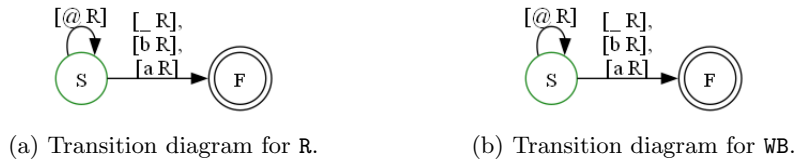


Fig. 2: Turing machine transition diagrams.

2.2 Composing Turing Machines in FSM

Turing machine composition is used to define more complex operations. In essence, think of Turing machines as (auxiliary) functions that may be composed to solve problems. This requires, however, a Turing machine that may be programmed. That is, a Turing machine that can consume and execute Turing machines much like an interpreter executes a program. In addition, such a Turing machine must also be able to perform conditional and unconditional branching, abstract over the read value, and, as a consequence of branching, sequence Turing machines. Such a machine is known as the universal Turing machine (UTM). The UTM is given as input a description of the machine to simulate as well as a description of the simulated input tape. The description of the machine to simulate may be thought of as a program that is interpreted. The UTM performs the actions in the description of the machine to simulate on the simulated input tape. To this end, it must track the next action to perform and the position of the head in the simulated input. Students may think of the next action to perform as a program counter. It indicates where to continue the execution of the simulated machine once the current action is simulated. The position of the simulated head indicates the next element to be read from the simulated input tape when the current action is executed.

The description of a composed Turing machine, `ctmd`, requires syntax like any other programming language. The syntax to describe a composed Turing machine is a list defined as follows:

1. empty list
2. `(cons m ctmd)`, `m` is either a `tm` or a `ctmd`
3. `(cons LABEL ctmd)`
4. `(cons (list GOTO LABEL) ctmd)`
5. `(cons`

`(BRANCH (listof (list symbol`

`(list GOTO LABEL))))`

`ctmd)`

- 6. `(cons ((VAR symbol) ctmd) ctmd)`
- 7. `(cons variable ctmd)`

The empty `ctmd` represents a machine that must only halt. If during the execution of a `ctm` the empty list is encountered then execution halts and the existing `tm` configuration is returned. The second variety states that a `ctmd` may be a list

```

#lang fsm

;; PRE: tape = (LM w) && i=k>0 && w in (a b)*
;; POST: tape = (LM w) && i=k+2 && w in (a b)*
(define RR (combine-tms (list R R) '(a b)))

(check-equal? (ctm-run RR (list LM 'b 'a 'a) 1)
              (list 'F 3 (list LM 'b 'a 'a)))
(check-equal? (ctm-run RR (list LM 'a 'a 'a) 2)
              (list 'F 4 (list LM 'a 'a 'a BLANK)))
(check-equal? (ctm-run RR (list LM 'a 'b 'b 'a) 3)
              (list 'F 5 (list LM 'a 'b 'b 'a BLANK)))
(check-equal? (ctm-run RR (list LM 'b) 1)
              (list 'F 3 (list LM 'b BLANK BLANK)))

```

Fig. 3: Composing R to move the head two spaces to the right.

that contains a machine, `m`, and a `ctmd`. The machine, `m`, may be a `tm` or a `ctmd`. During execution, first `m` is executed and then the nested `ctmd` is executed.

The third variety introduces a label to a `ctmd`. The label is a number used for unconditional branching. It represents the start of a `ctmd` that must be executed when control moves to this point in the `ctmd`. The fourth variety introduces a `GOTO` statement to a `ctmd`. This is how unconditional branching occurs. A `GOTO` statement contains a label that is jumped to when it is executed. That is, control is passed to the `ctmd` designated by the label.

The fifth variety introduces a conditional branching point. A `BRANCH` statement contains a list of branching options. Each option is a list that contains a symbol and an unconditional jump. If the read element matches the symbol in a branching option then control is passed to the `ctmd` specified by the label in the unconditional jump. One of the symbols must match the element being read. Otherwise, an error is thrown.

The sixth variety introduces an abstraction. A `VAR` statement contains a symbol representing a variable and an embedded `ctmd`. The variable is introduced to capture the value being read on the tape. The scope of the variable is the embedded `ctmd`. In the embedded `ctmd`, the seventh `ctmd` variety may occur. When variable is encountered, the input tape is mutated. The value of the variable is written at the head's position.

To illustrate how to define and run a composed Turing machine, consider the `ctm` to move the head two spaces to the right defined in Figure 3. The function `combine-tms` takes as input the description of a composed Turing machine and the input alphabet for the machine and returns a machine that expects as input the initial tape and the initial tape position. The composed machine may be applied to an initial tape and an initial head position and returns a Turing machine configuration. In this example, the composed Turing machine is a list that contains `R` twice. It is interpreted to execute the first `R` and when this

```

;; PRE:  tape = (LM BLANK w BLANK) and head on blank after w
;; POST: tape = (LM BLANK w BLANK w BLANK) and head on blank after second w
(define COPY (combine-tms
  (list FBL
    0
    R
    (cons BRANCH (list (list BLANK (list GOTO 2))
      (list 'a (list GOTO 1))
      (list 'b (list GOTO 1))))
    1
    (list (list VAR 'k)
      WB
      FBR
      FBR
      'k
      FBL
      FBL
      'k
      (list GOTO 0))
    2
    FBR
    L
    (cons BRANCH (list (list BLANK (list GOTO 3))
      (list 'a (list GOTO 4))
      (list 'b (list GOTO 4))))
    3
    RR
    (list GOTO 5)
    4
    R
    (list GOTO 5)
    5)
  '(a b)))

;; Tests for COPY
(check-equal? (rest (ctm-run COPY '(,LM ,BLANK ,BLANK ,BLANK) 3))
  '(5 (,LM ,BLANK ,BLANK ,BLANK ,BLANK ,BLANK)))
(check-equal? (rest (ctm-run COPY '(,LM ,BLANK a b b ,BLANK) 5))
  '(9 (,LM ,BLANK a b b ,BLANK a b b ,BLANK)))

```

Fig. 4: ctm that copies a given input tape.

execution halts it executes the second R starting in the configuration the first R left the machine in. After the second R is executed, RR halts given that there is nothing more to execute and returns the final configuration. The unit tests illustrate how to execute and test a composed Turing machine. The function `ctm-run` takes as input a composed Turing machine, an initial tape, and an

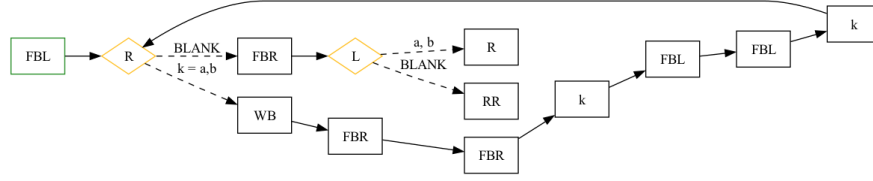


Fig. 5: Transition diagram for the `ctm` copy machine.

initial head position. It returns the configuration after the given composed Turing machine halts. For instance, in the second unit test, the head is reading the second `a`. After moving two spaces to the right, the tape is unchanged and the head position is 4 on the first blank after the third `a`.

3 Visualization Interface

3.1 Transition Diagram

The ability to generate a `ctm` transition diagram is provided through an extension to `sm-graph`. Each node in the transition diagram represents a `tm`. To stress the distinction with transition diagrams for non-composed Turing machines, nodes in a `ctm` graph are square-shaped. The `ctm`'s starting Turing machine is outlined in a green square. This coloring is consistent with the rendering of other FSM transition diagrams. To highlight conditional branching point, a node preceding a branch is rendered as a golden diamond with more than one labeled edge.

Rather than transitions, edges in `ctm` transition diagrams represent flow control from one `tm` to the next `tm`. Such edges are rendered as black arrows. In the case of a branch, however, there are multiple flow control options out of a diamond-shaped node. Such a node is connected to its possible successors using dashed edges with labels. The label on such an edge denotes any of the elements that must be read on the tape to take the branch. If there is a branch to a label that declares a variable then the label also denotes the values the variable may take.

Figure 4 displays a `ctm` to copy a word on the input tape. In the specification, the word to copy is denoted by `w`. `LM` denotes, `@`, the tape's left-end marker and `BLANK` denotes, `_`, a blank space on the tape. The machine starts by moving the head to the first blank to the left and then moving right to place the head on `w`'s next element, if any, to copy. After moving right, the machine branches based on the element read. If a blank is read then there are no more elements to copy and the machine branches to properly place the head in its final position. If anything else is read then `w`'s next element is copied to the second blank to the left and the machines loops to copy any remaining elements.

The transition diagram is displayed in Figure 5. In the graphic, `FBL`, the starting `tm`, is highlighted in green. The next `tm` to be executed is `R`. A directed edge towards `R` connects both nodes. Given that `R` precedes a branch, it is rendered within a golden diamond. There are two branching options denoted by

labeled dashed directed edges. The top branch is followed if a **BLANK** is read. The bottom branch is followed if the read element is an **a** or a **b**. In addition, the edge's label also indicates that the element read is defined as the binding of a local variable named **k**.

3.2 Dynamic Execution

In addition to generating a **ctm** transition diagram, the user can visualize machine execution using **ctm-viz**. This tool, as other **FSM** visualization tools, aims to keep the extraneous cognitive load of learning to how use it to a minimum. The user only needs to understand the use of 4 navigation arrows to advance the visualization forwards or backwards. The use of the arrows is specified as follows:

- Moves the visualization to the next step
- ← Moves the visualization to the previous step
- ↓ Moves the visualization to the end
- ↑ Moves the visualization to the start

Invoking the visualization tool is done as follows:

```
(ctm-viz ctm ctmd tape natnum)
```

The first argument is the **ctm** to execute (built using **combine-tms**). The second argument is the **ctmd** for the first argument. The third argument is the initial tape value. Finally, the fourth argument is the position of the head in the third argument. A visualization state consists of the **ctm** transition diagram and the current tape value. As the computation advances, the edge to the next machine to execute and the tape element read are highlighted in color.

4 Implementation

4.1 Static **ctm** Visualization

A **ctm** transition diagram is generated using two primary functions. The first traverses a given **ctmd** to extract all auxiliary machines used. These are used to generate the transition diagram's nodes. The second traverses the **ctmd** to generate the transition diagram's edges.

The function to generate the nodes traverses the given **ctmd**. As the traversal progresses, nodes are accumulated making sure to distinguish between nodes for **tms** that are used more than once (e.g., **R**, **FBR**, **FBL**, and **k** in Figure 5). Distinguishing between repeated **tms** is done by maintaining a counter and appending the value of the counter every time a **tm** is encountered in the **ctmd**. In addition, there is an accumulator for visited labels to prevent visiting a label more than once. Finally, a node graphic is generated using **Graphviz** and, therefore, during the **ctmd** traversal, a node is associated with a list of attributes needed to render it.

The function to generate nodes dispatches on the **ctmd**'s value as follows:

empty The function halts given that the `ctmd` is traversed.

First element is a label If the label is part of the accumulator then the function halts given that the label has already been processed. Otherwise, the label is added to the accumulator and the function recursively processes the rest of the `ctmd`.

First element is a GOTO or a BRANCH The first element is ignored and the function recursively processes the rest of the `ctmd`.

First element is a VAR Each element in the variable's scope is recursively processed as well as the rest of the `ctmd`.

Otherwise A node representation is created that contains a name, a color, a shape, and a label.

- a-node: Machine name appended with the counter value.
- a-color: Green for the start machine, gold for pre-branching machines, and black otherwise.
- a-shape: Diamond shaped for pre-branching machines and rectangle-shaped otherwise.
- a-label: Machine name as it appears in the `ctmd`.

The second function generates the transition diagram's edges by traversing the given `ctmd`. As the traversal progresses, the first and second elements are examined to determine the type of edge, if any, to generate. It is outlined as follows:

ctm of length 1 The function halts, because the last `tm` does not have an outgoing transition.

ctm of length 2 and the last element is a label The function halts, because the last `tm` does not have an outgoing transition.

The first two elements are symbols A solid edge without label is generated from the first `tm` to the second `tm`.

The first element is a label If the label is a member of the label accumulator then the function halts as the label has already been traversed. Otherwise, the label is added to the accumulator and the rest of the `ctmd` is recursively processed.

The first element is a VAR declaration Edges are generated for the part of the machine where the variable is in scope.

The first element is not a symbol The first element must be a `GOTO`, a label, or a branch. All are skipped and the rest of the `ctmd` is recursively processed.

First element is a symbol but second is not a symbol The first is the source auxiliary machine for a new edge. To determine the destination of the new edge, the second element is processed as follows:

- Second element is a label: The label is skipped and the rest of the `ctmd` is processed to find the destination of the new edge.
- Second element is a branch: Labeled dashed edges to the destinations of the branches are created.
- Second element is a `GOTO`: A solid edge to the destination of the unconditional jump is created.

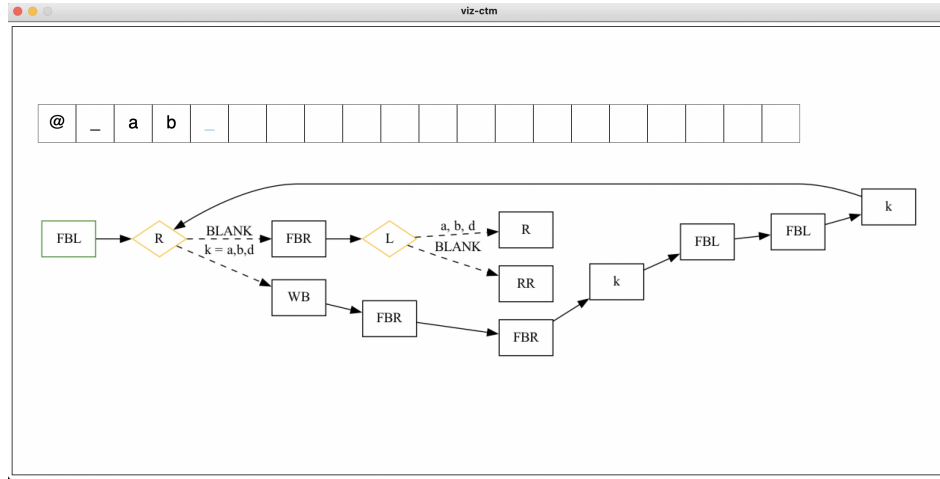


Fig. 6: Initial visualization state for the COPY ctm.

- Second element is a **VAR** list: An edge to the first machine in the variable's scope is created.

Once both nodes and edges are computed, a list containing them is given as input to **Graphviz**'s dot compiler to generate the graphic for the transition diagram. This graphic is returned as the value of the function.

4.2 Dynamic ctm Visualization Implementation

The trace of applying the given ctm to the given word and the list of transition diagram edges are used to create the dynamic visualization. A specialized transition-diagram-based graphic is computed for each auxiliary machine executed during a computation. This specialized graphic displays the state of the tape and highlights the edge to the next machine to execute. These images are stored in the following structure:

```
(struct vst (upimgs pimgs))
```

Here, **upimgs** denotes the list of graphics that must still be displayed and **pimgs** denotes the list of graphics that have already been displayed. When the user moves the visualization one step forward, a single visualization state image moves from **upimgs** to the end **pimgs** and it is displayed on the screen. If the user moves the visualization one step backwards, the last **pimgs** graphic is moved to **upimgs** and the next-to-last image from **pimgs** is displayed.

The creation of these graphics is done by traversing the trace of the computation to extract each of the computation's state and the lists of transition diagram nodes and edges to highlight the correct edge. The computation's state provides the current state, the current tape value and the current head position.

Thus, a graphic for the tape with the correct element highlighted may be generated. The correct edge to highlight is obtained by searching the list of edges for the one from the current state to the state in the next machine configuration.

4.3 Illustrative Example

Consider the COPY machine displayed in Figure 4 and its transition diagram displayed in Figure 5. When given (@ _ a b _) as the tape's initial value and 4 as the initial head position, the generated graphic is displayed in Figure 6. Observe that the blank in position 4 is highlighted (in blue) and no edges are highlighted (given that no machine has been executed).

The computation starts as follows:

$$\begin{aligned} (4 \text{ (@ _ a b _)}) &\vdash (\text{FBL } 1 \text{ (@ _ a b _)}) \\ &\vdash (\text{R } 2 \text{ (@ _ a b _)}) \end{aligned}$$

FBL moves the head to the first BLANK to the left. The machine's state after this step is displayed in Figure 7a. Observe that the head has moved to the blank in position 1 (which is highlighted) and the edge to the next machine, R, to execute is highlighted. After R executes, the head moves to position 2. Therefore, the first a is highlighted. The next auxiliary machine to execute depends on a conditional branch. Given that an a is under the head, the next machine to execute after branching is WB and the edge from R to WB is highlighted. The resulting graphic for the machine's state is displayed in Figure 7b.

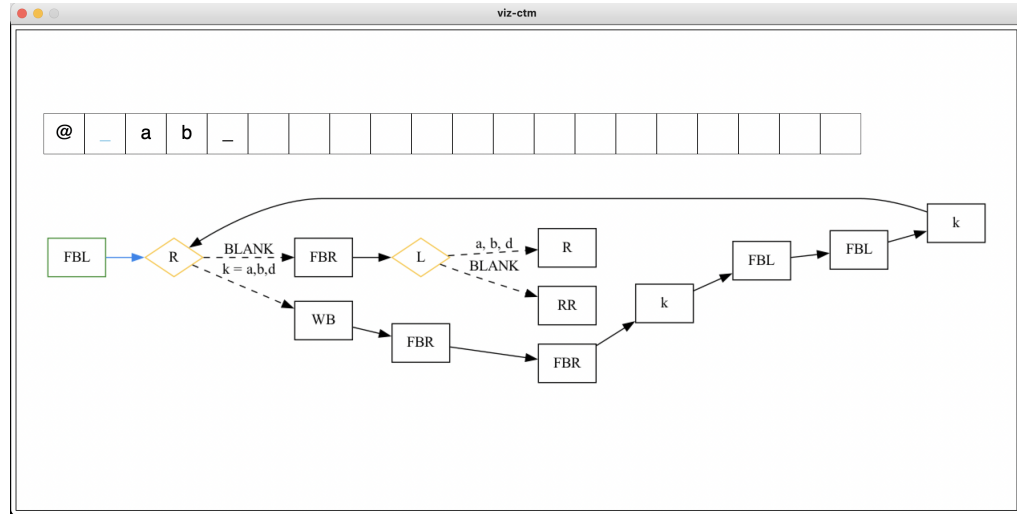
The computation continues as follows:

$$\begin{aligned} &\vdash (\text{WB } 2 \text{ (@ _ _ b _)}) \\ &\vdash (\text{FBR } 4 \text{ (@ _ _ b _)}) \\ &\vdash (\text{FBR } 5 \text{ (@ _ _ b _)}) \\ &\vdash (\text{k } 5 \text{ (@ _ _ b _ a)}) \\ &\vdash (\text{FBL } 4 \text{ (@ _ _ b _ a)}) \\ &\vdash (\text{FBL } 2 \text{ (@ _ _ b _ a)}) \\ &\vdash (\text{k } 2 \text{ (@ _ a b _ a)}) \\ &\vdash (\text{R } 3 \text{ (@ _ a b _ a)}) \\ &\vdash (\text{R } 4 \text{ (@ _ a b _ a b)}) \\ &\vdash (\text{FBR } 7 \text{ (@ _ a b _ a b)}) \\ &\vdash (\text{L } 6 \text{ (@ _ a b _ a b)}) \\ &\vdash (\text{R } 7 \text{ (@ _ a b _ a b)}) \end{aligned}$$

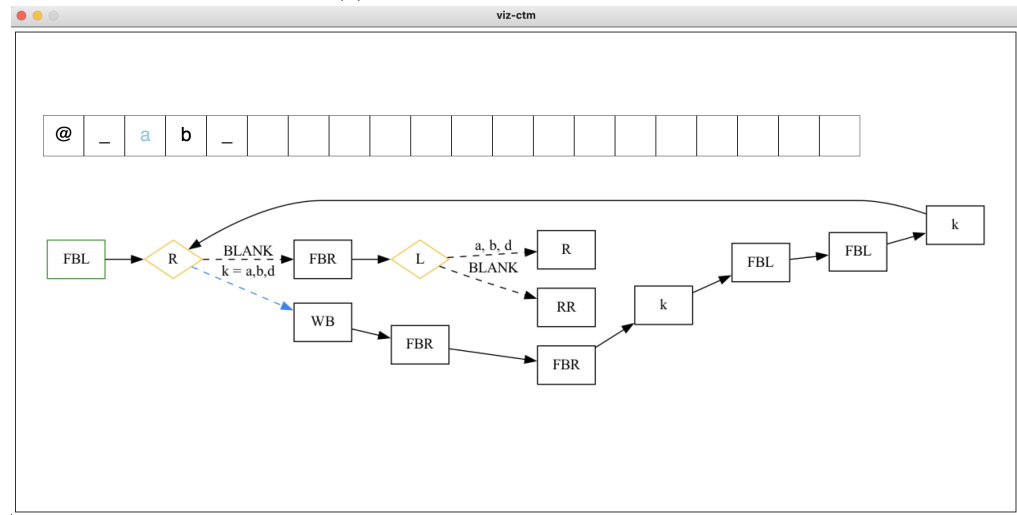
For each of these configurations a graphic is generated. In the final configuration, the head is at position 7 and there is no next auxiliary machine to execute. Thus, there is no edge to highlight. The generated graphic is displayed in Figure 8.

5 Focus Group Feedback

This section shall present empirical data collected from a focus group.



(a) First step of the visualization.



(b) Second step of the visualization.

Fig. 7

6 Related Work

This section shall compare and contrast with related work.

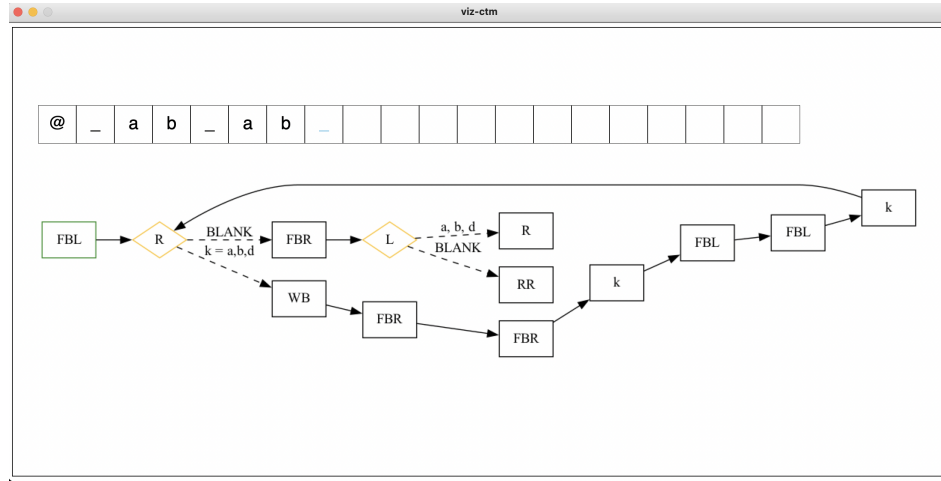


Fig. 8: Final visualization state.

7 Concluding Remarks

This article presents a novel graphical approach to help students understand composed Turing machines. The generation of transition diagram and a dynamic visualizer are now integrated into **FSM**—a domain-specific language for the automata theory classroom embedded in **Racket**. The transition diagram provides a mechanism for programmers to see the layout of their composed Turing machines. This layout serves as the basis to dynamically observe machine execution by displaying the machine’s state after the execution of every auxiliary machine.

Future work includes an in-depth study to collect feedback from students using the tools we developed in a classroom setting. In addition, the work presented in this article is being expanded to generate transition diagrams for multi-tape Turing machines, as well as adding visualization tools for word derivation using regular grammars, context-free grammars, and context-sensitive grammars.

References

1. 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>