The FSM Interface with Graphviz

Joshua M. Schappel and Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
jmschappel12@gmail.com|morazanm@shu.edu
Student Research Article

Abstract. Rendering graphs in an appealing manner is hard to do. For this reason, many programming languages interface with a library to render such graphics. There is very little in the literature, however, on how such an interface is implemented and on how the interface is integrated into the software architecture of a language. This article presents the interface to a graph-rendering library, Graphviz, developed for a domain-specific language, FSM, to program state-based machines. Design and implementation choices are discussed in the context of the FSM software architecture. The goal is to share the design and implementation choices made so that others may learn from them and implement such an interface for the programming language of their choice.

1 Introduction

FSM[1] (**F**unctional **S**tate **M**achines) is a domain-specific language (DSL) implemented in Racket for the automata theory and formal languages classroom [14]. In addition to easily defining state-based machines, grammars, and regular expressions, programmers may implement algorithms stemming from constructive proofs. Furthermore, FSM provides automated facilities to render a machine's transition diagram and to visualize machine execution. Both of these facilities require drawing graphs. Drawing graphs, however, is a complex and costly task that requires automation [11]. Without automation, FSM programmers are distracted from their primary task which is to implement a machine or a construction algorithm.

Given that drawing graphs is complex, the main impetus for current research on computer-aided graph drawing is to facilitate the visual analysis of various kinds of complex networked or connected systems [12]. Several graph drawing libraries have been built and successfully deployed. Among the most successful is Graphviz. Graphviz is an open-source software project used to help visualize information as abstract graphs and networks [9]. An attractive Graphviz feature is that it arranges nodes in an appealing manner. This liberates users from the burden of choosing an appealing graph layout. Another attractive characteristic is that it is programming-language independent. This achieved by providing a

[1] The reader may explore documentation and download from: https://morazanm.github.io/fsm/ .

DSL, called the DOT language, to create graph images. For a programmer using, for example, a functional language this is a mixed blessing. On the one side, the graph layout and rendering problem is solved for them. On the other, they must learn and use the DOT syntax.

Many higher-level programming languages hide the details of the DOT syntax by implementing an interface library to automatically generate DOT code and graphics. Programmers only need to specify the desired graph characteristics. This liberates programmers from using DOT syntax. Very little has been published on how a language implementation achieves this. A language implementor may, of course, dive into open source code to mine nuggets of implementation wisdom. Such a process, however, is time-consuming and error-prone, because documentation is lacking or incomplete leaving design ideas implemented and design choices made unclear.

FSM uses Graphviz to render the transition diagrams of state-based machines and to implement the visualization tool. An important lesson readers can walk away with is that the implementation of such a library does not need to be a difficult-to-understand complex system. In fact, thanks to the abstractions provided by functional programming, the individual functions required are straightforward to implement. This article presents FSM's implementation of its Graphviz library interface (for short, Graphviz library). How the Graphviz library is integrated into the FSM software architecture is presented. The design choices made are discussed and the concrete implementation of core Graphviz library functions is presented. In this manner, any reader wishing to build a Graphviz interface for their favorite functional programming language may build on our experience. The article is organized as follows. Section 2 discusses related work. Section 3 describes the DOT syntax targeted. Section 4 presents the FSM software architecture. Section 5 discusses the machine representations used in FSM's implementation. Section 6 presents the design and implementation of the Graphviz library. Section 7 presents the generation of DOT code. Section 8 illustrates how it is all put together to provide functionality to FSM programmers. Finally, Section 9 presents concluding remarks and directions for future work.

2 Related Work

The Racket Generic Graph Library includes a feature to render a graph as a DOT program. The primary goal of this library is not graph rendering, but users may specify features to render a graph. It takes an imperative approach with programmers adding nodes and edges piecemeal. In contrast, the work described in this article liberates FSM programmers from having to specify features about how to render a state machine's transition diagram. Instead, FSM programmers are presented with a standard interface for rendering transition diagrams and for visualizing machine execution. FSM programmers do not need to know nor be aware of the DOT syntax. Equally important, FSM developers are presented with a versatile interface to define new transition diagrams as FSM is expanded and are not required to be familiar with the DOT language.

Racket provides an implementation for Markov chains [10] that allows programmers to visualize them as transition diagrams. This library produces DOT programs for directed graphs in black and white with circle nodes and single-label edges for a given chain. Likewise, hidden from programmers, FSM produces DOT programs for directed graphs. In contrast, the transition diagrams produced by FSM utilize different colors and shapes to distinguish the role of a node and have edges with multiple labels representing multiple transition rules between a pair of states. In addition, FSM developers may customize a graph for new types of transition diagrams. That is, they are not restricted to circle nodes in black and white.

OCaml provides an interface for Graphviz as part of their ocamlgrpah library [5]. Types that allow programmers to define nodes and edges to specify graph attributes, such as colors and arrow styles, are provided. Nodes and edges are added piecemeal. Programmers can render their created graphs as graphics by invoking the DOT compiler. As in OCaml, FSM uses a system call to invoke the DOT compiler. In contrast, FSM programmers do not have to invoke the DOT compiler nor do they have to explicitly open a graphic file to visualize their state machines. This convenience removes the burden of explicitly building graphs and graph attributes.

Outside the functional programming world Graphviz is widely used. In Python, for example, Graphviz is used as part of the graph-tool [2] and the graphviz [3,15] libraries. Programmers may add nodes and edges piecemeal in an imperative fashion. PlantUML uses Graphviz to render UML diagrams [6]. The UML diagrams are generated from textual descriptions. There are many other software projects that use Graphviz (too many to list here). In contrast to all, FSM programmers are not required to specify graph attributes and FSM developers do not need to be aware of the DOT syntax.

Outside the programming world, for example, DrugMap uses Graphviz to visualize drug repositioning studies [7]. Its visualization tool converts related database items into a DOT program to create a directed tree rendering. Users are not required to provide graph characteristics for the tree. Similarly, FSM programmers are not required to provide graph characteristics. These are automatically derived from the machines they define. In contrast, the transition diagrams produced by FSM are not tree-based graphs.

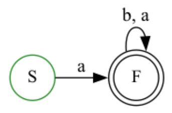
3 The Dot Language Targeted

The DOT grammar allows programmers to specify detailed graph attributes such as graph type, node characteristics, edge characteristics, and subgraph characteristics [1]. The core subset of the DOT language grammar used by FSM is:

```
    graph ::= (graph | digraph) [ID] stmt-list
stmt-list ::= [stmt [;] stmt-list]
     stmt ::= node-stmt | edge-stmt | attr-stmt
node_stmt ::= node_id [attr_list]
edge_stmt ::= node_id edgeop node_id [attr_list]
```

```
1. digraph G {rankdir="LR";
2.  F [color="black",shape="doublecircle",label="F"];
3.  S [color="forestgreen",shape="circle",label="S"];
4.  F -> F [fontsize=15, label="b,a"];
5.  S -> F [fontsize=15, label="a"];}
```

(a) DOT program



(b) Graph produced

Fig. 1: Dot code and graph produced

Keywords are in bold and square brackets indicate optional items. A graph defines its type (undirected or directed), an optional identifier, and a statement list defining characteristics. Each statement in a statement list describes a node, an edge, or a set of attributes. A node statement contains a node identifier and a list of attributes. An edge statement contains a left node identifier, an edge operator, a right node identifier, and a list of attributes. Figure 1a displays a sample DOT program. Line 1 indicates the construction of a directed graph named G with an attribute indicating that the graph be laid out with edges pointing from left to right. Lines 2–3 are node statements with attributes. Lines 4–5 are edge statements with attributes. Evaluating the program results in the graph displayed in Figure 1b

FSM always produces a digraph because transition diagrams indicate an automaton's movement between states. Nodes represent states and their attributes include color, shape, and label. For example, in Figure 1b, S is the starting state (in a green outlined circle) and F is the only final state (in a doubled circle). Edges represent transitions between nodes and their attributes include color, label, and font size. For example, in Figure 1b, there is a single transition from S to F that consumes an a and there are two transitions from F to F consuming, respectively, b and a.

4 FSM Architecture

FSM's implementation has four major components: FSM Core, FSM GUI, Graphviz library, and the User Interface. FSM Core implements constructors, observers, and random testing for state machines, grammars, and regular expressions. It provides the bulk of the interface available to write FSM programs.

The FSM GUI implements the Visualization Tool for machine execution. Given that the Visualization Tool must track the state of the machine throughout a computation, it stores more information about the machine than what is provided to a machine's constructor. The extra information allows users to step, forward and backwards, through a computation's transitions. In addition, the Visualization Tool allows programmers to provide for each state, S, an invariant predicate that ought hold when the machine enters S. During visualization, users can see if their invariant predicates hold. This allows machines to be vali-

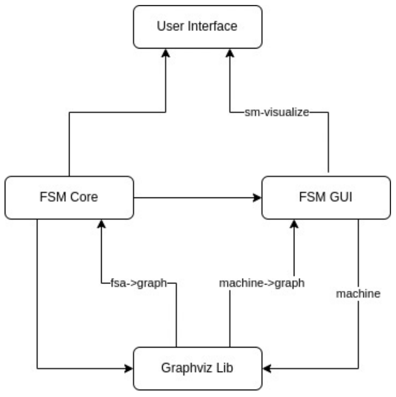


Fig. 2: `FSM` software architecture

dated before writing formal proofs. A single function, `sm-visualize`, is provided to the `FSM` User Interface.

The `Graphviz` library provides the functionality to render a state machine's transition diagram. This is done by generating and compiling a `DOT` program for a given machine instance. Finally, the User Interface is a set of functions provided from `FSM` Core and `FSM` GUI that form the interface offered to programmers.

Figure 2 displays the `FSM` architecture. `FSM` Core provides its functionality (constructors, observers, testers) to the `FSM` GUI, `Graphviz` library, and the User Interface. The `Graphviz` library provides graphic-rendering functions to `FSM` Core and `FSM` GUI. `FSM` GUI provides `sm-visualize` to the User Interface. `FSM` Core and `FSM` GUI provide their internal machine representation to the `Graphviz` library that uses it to generate the proper graphic-rendering function.

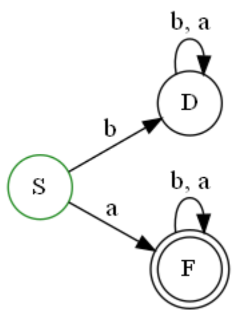
To bring the whole process together, consider the `FSM` definition for a deterministic finite-state automaton displayed in Figure 3. Applying `sm-graph` to `a-aUb*` results in the graph displayed in Figure 4a. The image is obtained using `fsa->bitmap` provided by the `Graphviz` Library and only displays states and edges. Running the visualization tool with invariants is done as follows:

```
(sm-visualize a-aUb*
              (list 'S S-INV) (list 'F F-INV) (list 'D D-INV))
```

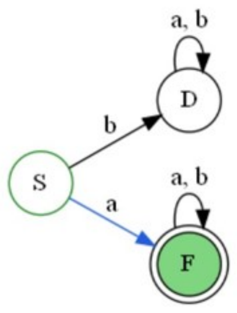
A snapshot of machine execution is displayed in Figure 4b. The machine has moved from `S` to `F` (edge highlighted in blue). In addition, `F-INV` holds (state highlighted in green). As the reader can see machine visualization includes more information than simply rendering the transition diagram using `sm-graph`. This is why a different rendering function is required.

```
#lang fsm
(define a-aUb*
  (make-dfa '(S F D) '(a b)      ;; the states & input alphabet
            'S  '(F)             ;; the starting and final states
            '((S a F) (F a F) (F b F)) ;; the transition function
            'nodead))                  ;; do not add a dead state
(define (S-INV ci) (empty? ci))
(define (F-INV ci)
  (and (not (empty? ci)) (eq? (first ci) 'a)
       (andmap (λ (s) (or (eq? s 'a) (eq? s 'b))) (rest ci))))
(define (D-INV ci) (and (not (empty? ci)) (eq? (first ci) 'b)))
```

Fig. 3: A deterministic finite automaton in FSM for L = a(a ∪ b)*.



(a) Transition diagram



(b) Visualization transition diagram

Fig. 4: The renderings of `a-aUb*`.

5 Machine Representations

5.1 FSM Core

In FSM Core, machines are represented as functions. Applying a machine, for example, to a word returns whether or not the word is accepted or rejected. Observers return the components (e.g., the states or the transition rules) of the machine. In this manner, the machine may be manipulated or used to construct new machines with which it shares components.

The components common to all machine types are defined as follows:

```
K = (listof state)  Σ = (listof symbol)  s = start state ∈ K
F = (listof state)  R = (listof rules)
```

K is the set of states, Σ is the input alphabet, s is the starting state, F is the set of final states, and R is the transition relation. The constructor signatures for the different machine types are:

```
dfa: (make-dfa K Σ s F R)   ndfa: (make-ndfa K Σ s F R)
```

```
pda: (make-ndpda K Σ Γ s F R), where Γ = stack alphabet
tm: (make-tm K Σ R s F), (make-tm K Σ R s F Y), where Y∈K
```

The constructors are, respectively, for deterministic finite-state automata, nondeterministic finite-state automata, pushdown automata, Turing machine, and Turing machine language recognizer. A Turing machine language recognizer decides a language and requires, `Y`, a single final accepting state. A Turing machine performs a computation instead of deciding a language.

5.2 `FSM` GUI

All machine types have a state list, a start state, a list of final states, a rule list, an input alphabet, and type tag in common. This is akin to the elements provided to a constructor in `FSM` Core. The `Visualization Tool`, however, does not represent machines as functions. Instead, it stores all this information in a `machine` structure:

```
(struct machine (state-list start-state final-state-list
                 rule-list  sigma-list  alpha-list  type))
```

Deterministic and nondeterministic automatons do not require any more information. That is, they are represented with an instance of the above structure.

In addition to the fields in a `machine` structure, a pushdown automata has a stack alphabet. Its structure representation is defined as follows:

```
(struct pda-machine machine (stack-alpha-list))
```

It inherits the fields in `machine` and adds a field for the stack alphabet.

Similarly, Turing machines have additional fields. A Turing machine, regardless of subtype, requires a tape position. This is needed to highlight the position of the head on the tape. In addition to a tape position, a language recognizer requires an accepting final state Their structures are defined as follows:

```
(struct tm-machine machine (tape-posn))
(struct lang-rec-machine tm-machine (accept-state))
```

5.3 Adapter

Function-based and structure-based machine representations are incompatible on the surface. Nonetheless, the `Graphviz` library must work with both representations. To reduce the burden of two different machine representations on `FSM` developers, an adapter pattern strategy is used [8,13]. The purpose of an adapter is to provide the interface that an `FSM` component expects while using the services of a component with a different interface. In this case, it converts the interface of each machine representation into a single interface. In essence, we may think of an adapter as a wrapper that hides the details of different representations.

The adapter converts a given instance of a machine representation to a structure that contains all the data provided by either the FSM Core or the FSM GUI representation. The structure contains a field for each component common to both representations (e.g., the set of states) as well as additional fields contained in the FSM GUI representation (e.g., the current state). In addition, the structure contains the color-palette for color-blind options. When the adapter constructor is called with an FSM Core machine representation the default value for FSM GUI fields is false.

Functions may now be defined to generate a bitmap file from a given machine representation. A given machine representation is adapted, transformed into a graphic using Graphviz, and converted to a bitmap. In this manner, for example, any FSM developer working on the Graphviz library needs to only reason about a single representation. The conversion functions are displayed in Figure 5. Observe that a single auxiliary function, `graph->bitmap`, is called to produce the graphic.

Given that the main functions above are provided to other modules as displayed in Figure 2, a contract is used for safety. The contract is:

```
(provide
 (contract-out
  [fsa->bitmap (-> any/c colorblind-opt? image?)]
  [machine->bitmap (-> machine?
                       colorblind-opt?
                       (or/c dfa/ndfa-rule? pda/tm-rule? boolean?)
                       (or/c symbol? boolean?)
                       inv-state?
                       image?)]))
```

The contract above allows FSM Core and FSM GUI developers to trust that the functions provided by the Graphviz library meet the expected specification.

6 Graphviz Core Library

Nodes, edges, and graphs are represented as structures. Nodes contain a name and attributes. Edges contain two nodes and attributes. Graphs contain a name, nodes, edges, formatters to convert an attribute to a string (for a DOT program), and a set of attributes. Attributes are stored in a hash table using attribute symbols as keys. For instance, a black double circled node named F is represented as follows:

```
    (hash 'color "black" 'shape "doublecircle" 'label "F")
```

This hash table is used to produce the following DOT code for a node:

```
    F [color="black", shape="doublecircle", label="F"];
```

A formatter is a structure containing three hash tables: one for the graph, one for graph's node, and one for the graph's edge. The hash tables associate an

```
;; core-machine symbol → bitmap
(define (fsa->bitmap fsa color-blind-mode)
  (graph->bitmap (fsa->graph fsa color-blind-mode)
                 (current-directory)
                 "vizTool"))
 ;; core-machine symbol → image
 ;; Purpose: Generate graphic for given Core machine
 (define (fsa->graph fsa color-blind-mode)
   (define adapter
     (fsa-adapter
      (sm-states fsa) (sm-start fsa) (sm-finals fsa)
      (sm-rules fsa)  (sm-type fsa)
      (if (is-tm-lang-rec? (sm-type fsa)) (sm-accept fsa) #f)
      #f #f 'none (make-color-palette color-blind-mode)))
   (fsa-adapter->graph adapter))

;; gui-machine symbol rule state state → bitmap
;; Purpose: Generate image for given fsm GUI machine
(define (machine->bitmap machine  color-blind-mode
                         cur-rule cur-state inv-state)
    (graph->bitmap (machine->graph machine   color-blind-mode
                                  cur-rule  cur-state inv-state)
                   (current-directory)
                   "vizTool"))
;; gui-machine symbol rule state state → image
(define (machine->graph machine   color-blind-mode
                        cur-rule  cur-state inv-state)
  (define adapter
    (fsa-adapter
     (map (lambda (s) (fsm-state-name s))
          (machine-state-list machine))
     (machine-start-state machine)
     (machine-final-state-list machine)
     (machine-rule-list machine)
     (machine-type machine)
     (if (is-tm-lang-rec? (machine-type machine))
         (lang-rec-machine-accept-state machine)
         #f)
     cur-state
     cur-rule
     inv-state
     (make-color-palette color-blind-mode)))
  (fsa-adapter->graph adapter))
```

Fig. 5: Bitmap-generating functions for the transition diagram of an automaton.

attribute with a formatting function used to generate DOT code. If a formatter is provided for an attribute then it is used anytime DOT code is generated for that

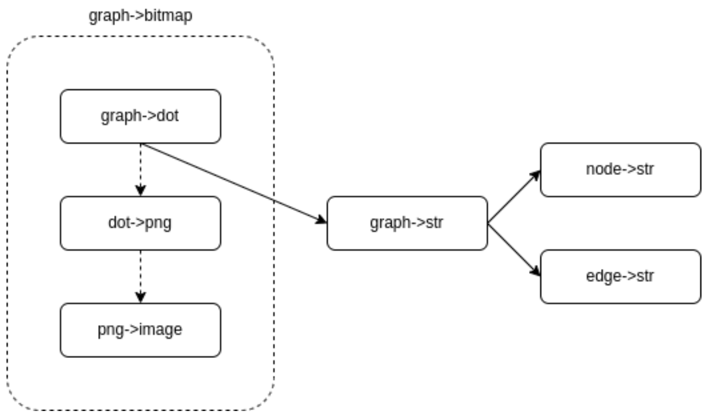


Fig. 6: `FSM`'s `Graphviz` Control Flow

attribute. For instance, if the value for a node label is a Boolean then a formatting function converts a Boolean to a string. If this were the only formatting needed then the required structure is:

```
(formatters
 (hash) (hash 'label (lambda (b) (if b "true" "false"))) (hash)
```

When an attribute formatter is not provided then the attribute must be a string.

6.1 Control Flow

There are three operations that `FSM`'s `Graphviz` library performs: convert an `fsa-adapter` to a `DOT` language program, generate a portable network graphic (i.e., a `.png` file), and convert a portable network graphic to a `Racket` bitmap. The generation of the portable network graphic is done using the `Graphviz` compiler. The generation of a bitmap is done to display the graphic in either the `REPL` or in `FSM`'s visualization tool. Figure 6 displays the software architecture of the `Graphviz` library. The function `graph->bitmap` generates a bitmap from a given adapter. It does so in three steps (denoted by dashed arrows). In the first step, it generates a string representing a `DOT` program using an auxiliary function, `graph->str`, to generate the `DOT` program for a graph. This function uses two auxiliary functions to generate the `DOT` code for nodes and for edges. In the second step, it generates the graphic by calling the `DOT` compiler. In the third step, a bitmap is generated from the graphic.

The generation of a bitmap file from a `graph` structure is done using function composition as follows:

```
(define (graph->bitmap grph dir fname)
 ((compose png->bitmap dot->png graph->dot) grph dir fname))
```

The second and third parameters, respectively, specify the directory and the name for the bitmap file.

The generation of the DOT program from a graph saves a .dot file in the given directory with the given name and returns the path to the generated file. The auxiliary function, graph->str, is used to generate the DOT code. If the filename already exists in the specified directory it is overwritten. The function is implemented as follows:

```
;; graph path string → path
(define (graph->dot graph save-dir filename)
  (define dot-path (build-path save-dir
                               (format "~a.dot" filename)))
  (call-with-output-file dot-path #:exists 'replace
    (lambda (out) (displayln (graph->str graph) out)))
  dot-path)
```

To generate the portable network graphic, the extension of the .dot file is changed to .png and a system call is made to invoke the Graphviz compiler. If the operation is successful the path to the newly generated file is returned. Otherwise, an error is thrown (e.g., when the Graphviz compiler is not found). The function is implemented as follows:

```
;; path → path
(define (dot->png dot-path)
  (define png-path (path-replace-extension dot-path ".png"))
  (if (system (format "dot -Tpng ~s -o ~s"
                      (path->string dot-path)
                      (path->string png-path)))
      png-path
      (error "Error when creating png file")))
```

Finally, png->bitmap is simply an alias for Racket's image library function bitmap/file [4]. It is defined as follows:

```
;; png->bitmap: path → string
(define png->bitmap bitmap/file)
```

6.2 From graph to Dot Program

The function graph->dot converts a graph, its nodes, its edges, and their attributes to a DOT program string. Following the DOT grammar, the first component is the program header. It includes the type digraph, the name of the graph, and the graph's attributes. For example, the result of this step is line 1 in Figure 1a. This is generated using the graph formatters, if any, in the given graph. An auxiliary function, hash->str, is used to generate the needed string using the formatting functions. The second component of the returned program is for the nodes and their attributes. This is generated by folding a function over the graph's list of nodes. This function creates a DOT line of code for each node

```
;; graph → string
(define (graph->str g)
  (define header (format "digraph ~s \n" (graph-name g)))
  (define fmtrs (graph-fmtrs g))
  (string-append
   header
   (format "~a;\n" (hash->str
                    (graph-atb g)
                    (formatters-graph fmtrs)
                    ";\n"))
   (foldl (lambda (n a)
            (string-append
             a
             (node->str n (formatters-node fmtrs))))
          ""
          (graph-node-list g))
   (foldl (lambda (e a)
            (string-append
             a
             (edge->str e (formatters-edge fmtrs))))
          ""
          (graph-edge-list g))
   ""))
```

Fig. 7: The function to generate a DOT program for a graph.

containing the node's name and its attributes. The string for the attributes is generated using an auxiliary function, node->str, and using the node formatting functions. The result of this step, for instance, are lines 2–3 in Figure 1a. The final component of the returned program is for the edges and their attributes. It is computed in a similar fashion as the nodes and their attributes except that graph's edges and the edge formatting functions are used. Lines 4–5 in Figure 1a illustrate the result of this step. The function to convert a graph to a DOT program is implemented as displayed in Figure 7.

The function hash->str traverses the entries in a hash table. Each attribute is formatted using its formatting function if it exists and, otherwise, its value is used. The results for the attributes are collected into a single string. The function is implemented as follows:

```
;; hashtable hashtable [string] → string
(define (hash->str hash fmtr (spacer ", "))
  (define (key-val->string key value)
    (define fmtr-fun (hash-ref fmtr key #f))
    (if fmtr-fun
        (format "~s=~s" key (fmtr-fun value))
        (format "~s=~s" key value)))
  (string-join (hash-map hash key-val->string) spacer))
```

The optional argument represents the element separator. It is customizable because for graph attributes it must be a semicolon but for nodes and edge attributes a comma is used.

The `node->str` and `edge->str` functions use `hash->str` to format the attributes for nodes and edges. They are implemented as follows:

```
;; node hashtable → string
(define (node->str node fmtr)
  (format "~s [~a];\"
          (node-name node)
          (hash->str (node-atb node) fmtr)))

;; edge hashtable → string
;; Purpose: Generate dot code for the given edge
(define (edge->str edge fmtr)
  (format "~s -> ~s [~a];\"
          (edge-start-node edge)
               (edge-end-node edge)
     (hash->str (edge-atb edge) fmtr)))
```

7 From an `fsa-adapter` to a `DOT` Program

Section 5.3 presents the adapter structure to unite the different machine representations. The function `fsa-adapter->graph` generates a `DOT` program from a `fsa-adapter` instance. The function creates an initial `graph` with a formatter for the transition rules, converts states to `nodes`, and converts transition rules to `edges`. The function is implemented as follows:

```
;; fsa-adapter → string
(define (fsa-adapter->graph adapter)
  (fsa-rules->edges
   adapter
   (fsa-states->nodes
    adapter
    (create-graph
     'G
     #:fmtrs (formatters
              (hash)
              (hash)
              (hash 'label rule-label->str))))))
```

The initial `graph` only has a formatter, `rule-label->str`, for an edge label. This formatter, of course, must be able to process the rules for any machine type that is represented by the given `fsa-adapter`. The function `fsa-states->nodes` adds the `nodes` to the initial graph. Finally, `fsa-rules->edges` adds the edges to the graph that contains the nodes.

7.1 Generating Labels

The `rule-label->str` function converts a transition rule to a `Graphviz` edge label. Given that there may be multiple transitions that take a machine from a state `A` to a state `B`, this function may produce an edge with multiple labels. Given that the length of all the transitions from one state to another is arbitrary, this function limits the length of any one label. If this length is surpassed then labels are stacked. Based on this design idea, the function is implemented as follows:

```
;; (listof rules) → string
;; Purpose: Generate dot code for labels
(define (rule-label->str rules)
  ;; (listof string) → (listof string)
  ;; Purpose: Generate a list of strings to stack
  (define (format-lines rule-strs)
    ;; (listof string) → (cons (listof string) (listof string))
    ;; Purpose: Generate a pair: new label and unprocessed rules
   (define (format-line l acc count)
     (match l
       ['() (cons acc '())]
       [`(,x ,xs ...)
        (if (and (not (empty? acc))
                 (> (+ 2 count (string-length x)) RULE-LIMIT))
            (cons acc l)
            (format-line
             xs
             (append acc (list x))
             (+ count (string-length x))))]))
   (match-define (cons line xs) (format-line rule-strs '() 0))
   (if (empty? lines)
       '()
       (cons (string-join line ", ") (format-lines xs))))
 (string-join
  (format-lines (map fsa-rule->label rules)) ",\n"))
```

The local function `format-lines` consumes a list of strings and returns a new list of strings in which each string is of a length that does not exceed `RULE-LIMIT`. The returned list of strings are stacked as the label for an edge by adding a return string to each. In this manner, the transition strings are stacked as the label for an edge. The auxiliary function `format-line` is used to generate the strings to stack. It takes as input the transition strings and returns a pair. The first element of the returned pair is the next label string to stack and the second element is the remaining unprocessed transition strings. In essence, this function collects all the transition strings, from the front of the given list, that fit in one line of a label string, adds a comma at the end, and recursively processes the remaining strings.

The function `fsa-rule->label` generates `DOT` code for a label based on the given transition rule. It dispatches on the rule type. For finite-state automaton the generated label contains the element read from the tape. For a pushdown automata, the generated label contains the read element, the values popped from the stack, and the value pushed onto the stack. For both varieties of Turing machines, the generated label contains the element read and the action taken by the machine. The auxiliary function `stringify->value` converts numbers and symbols to their string equivalent. The function is implemented as follows:

```
;; rule → string
(define (fsa-rule->label aList)
  (define (list->str los accum)
    (match los
      ['() (string-append (string-trim accum) ")")]
      [`(,x ,xs ...)
       (list->str xs (string-append accum
                                    (stringify-value x)
                                    " "))]))
  (match aList
    ;; dfa/ndfa
    [(list _ input _) (stringify-value input)]
    ;; pda
    [(list (list _ read pop) (list _ push))
     (format "[~a ~a ~a]"
             (stringify-value read)
             (if (list? pop)
                 (list->str pop "(")
                 (stringify-value pop))
             (if (list? push)
                 (list->str push "(")
                 (stringify-value push)))]
    ;; tm and tm lang rec
    [(list (list _ b) (list _ d))
     (format
       "[~a ~a]" (stringify-value b) (stringify-value d))]))
```

7.2 Adding Nodes

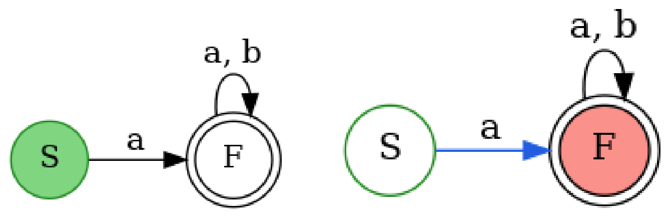
The function `fsa-states->nodes` adds an `fsa-adapter`'s states to the given `graph` as follows:

```
;; fsa-adapter graph → graph
(define (fsa-states->nodes fsa graph)
  (define (fsa-state->node state graph)
    (add-node graph state #:atb (build-node-hash state fsa)))
 (foldl fsa-state->node graph (FSA-adapter-states fsa)))
```

```
;; symbol fsa-adapter → hashtable
(define (build-node-hash state fsa)
 (define inv-state (fsa-adapter-inv-state fsa))
 (define is-cur-state? (equal? state (fsa-adapter-cur-state fsa)))
 (define palette (fsa-adapter-palette fsa))
 (define state-type
   (cond
     [(and (is-tm-lang-rec? (fsa-adapter-type fsa))
           (equal? state (fsa-adapter-accept fsa)))
      'accept]
     [(and (equal? state (fsa-adapter-start fsa))
           (member state (fsa-adapter-finals fsa)))
      'startfinal]
     [(equal? state (fsa-adapter-start fsa)) 'start]
     [(member state (fsa-adapter-finals fsa)) 'final]
     [else 'default]))
 (define color
   (match state-type
     [(or 'start 'startfinal 'startaccept)
      (color-palette-start palette)]
     [_ "black"]))
 (define shape
   (match state-type
     [(or 'startaccept 'accept) "doubleoctagon"]
     [(or 'final 'startfinal) "doublecircle"]
     [_ "circle"]))
 (define attributes `((color . ,color) (shape . ,shape)))
 (make-immutable-hash
  (if (and is-cur-state? (not (equal? 'none inv-state)))
      (append
       attributes
       `((style . "filled")
         (fillcolor
          .
          ,(match inv-state
             ['pass (color-palette-inv-true palette)]
             ['fail (color-palette-inv-false palette)]))))
      attributes)))
```

Fig. 8: Function to create a node's attribute hash table.

The function `build-node-hash` builds the attribute hash table for the node. A node's attribute hash table is constructed from a `fsa-adapter` instance. There are two attributes that must be defined for all states: color and shape. If the state is a starting/final/accept state then its color is obtained from the color palette. Otherwise, it is black. If the state is a Turing machine language recognizer accept state then its shape is a double octagon. If it is a final state then its shape is a



(a) Invariant holds. (b) Invariant does not hold.

Fig. 9: Machine with invariants rendering.

double circle. Otherwise, its shape is a circle. Finally, it must be determined if the state is the current state and if there is an invariant predicate for the state. If so, the fill color is added to the attributes, which is obtained from the color palette based on whether the invariant holds. Otherwise, no more attributes are added. The implementation is displayed in Figure 8.

Figure 9 illustrates the goals achieved when invariants are provided. Observe that the start state, `S`, is a green outline circle. The final state, `F`, is a double circle. In Figure 9a, the machine is in `S` and the invariant holds (denoted by the green filling). In Figure 9b, the machine is in `F` and the invariant does not hold (denoted by the red filling).

7.3 Adding Edges

The creation of an edge's attribute hash table is created in a similar manner from the values in the given `fsa-adapter`. In the interest of brevity, its implementation is not displayed. We note that if an edge between two nodes already exists then the transition rule is appended to the edge (i.e., a new edge is not created). The goals achieved are also displayed in Figure 9. In Figure 9a, no arrow is highlighted because the machine has not yet made a transition. Once a transition is made, the arrow representing the transition is highlighted in blue as displayed in Figure 9b.

8 Putting it all Together

The final step of the implementation is to employ the two functions provided by the `Graphviz` library. As depicted in Figure 2, `fsa->bitmap` is provided to `FSM` Core and `machine->bitmap` is provided to `FSM` GUI. `FSM` Core uses `fsa->bitmap` to implement the primitive `sm-graph` provided to programmers. This primitive consumes an `FSM` Core machine and returns the machine's transition diagram as a bitmap image. It is implemented as follows:

```
;; fsa [number] → image
(define (sm-graph fsa #:color [color-blind-mode 0])
   (fsa->bitmap fsa color-blind-mode))
```

The function uses `fsa->bitmap` from Figure 5 to produce the graphic to display.

`FSM` GUI internally uses `machine->bitmap` to produce a graphic for the machine's current state in the visualization tool. The image-producing function is implemented as follows:

```
;; machine bool symbol symbol number → image
(define (create-dot-png machine  hasRun?  cur-state
                         cur-rule cd-opt)
  (define inv-type
    (if hasRun?
        (determin-inv machine cur-state #:graphViz true)
        'none))
  (scaled-graph (machine->bitmap
                 machine
                 (get-cb-opt)
                 cur-rule
                 cur-state
                 inv-type)
               MACHINE-TYPE))
```

In essence, it scales the bitmap image to fit in the visualization tool's GUI. It provides `machine->bitmap` all the necessary arguments to correctly display the current state of the machine and the value of the invariant for the current state. A final illustrative graphic produced is displayed in Figure 10. The graphic displays the state of a Turing machine language recognizer for $\mathtt{a}^{\mathtt{n}}\mathtt{b}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}}$. Such a visualization unequivocally drives home the point that the generation of such images needs to be automated as done in `FSM`. It is unlikely to be insightful or productive to have programmers (e.g., students in formal languages and automata theory course) drawing such transition diagrams by hand.

9 Concluding Remarks and Future Work

This article presents `FSM`'s interface with the `Graphviz` library. The interface liberates `FSM` programmers from having to explicitly build transition diagrams for their state-based machines. In addition, the interface liberates `FSM` developers from having to reason about each machine representation used in `FSM`'s software architecture. This latter benefit is achieved by using an adapter to present a uniform view for machines. The library described uses `Graphviz`'s `DOT` compiler to graphically render a machine. It converts an adapter to a `DOT` program, invokes the `DOT` compiler to create a portable network graphic, and converts the portable network graphic to a bitmap that may be displayed in the `REPL` or in `FSM`'s visualization tool. The generation of `DOT` code is done using data in an adapter

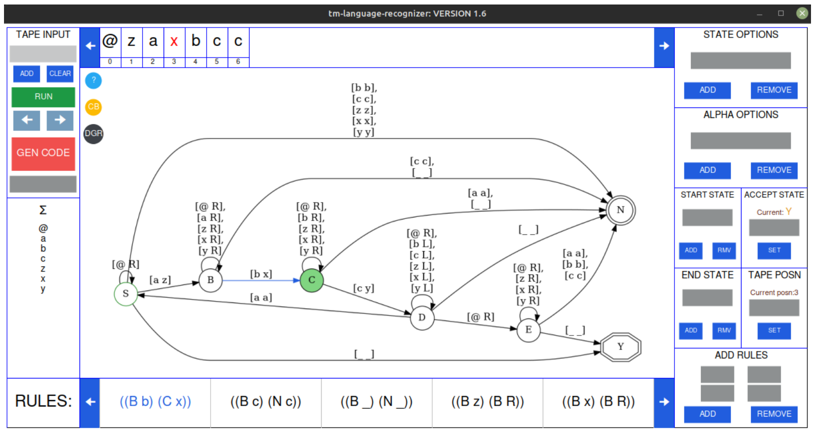


Fig. 10: Turing machine language recognizer for a^{n}b^{n}c^n visualization.

and associating it with formatting functions that render attributes as strings. This done for states and transition rules in a machine to create a transition diagram. The bulk of the implementation of the interface is presented to assist others to build such an interface for their desired domain. As the reader can now appreciate, the abstractions provided by functional programming facilitate managing the complexity of designing and of implementing such a system.

`FSM` is in constant growth and being expanded with new features and machines. Currently, we are working on implementing multitape Turing machines. This will require an extension to the visualization tool to display multiple tapes. Longer term, we wish to implement a generic visualization system for word derivation using a regular, context-free, or context-sensitive grammar.

References

1. Dot language. `https://graphviz.org/doc/info/lang.html` (November 2022), last accessed: November 2022
2. graph-tool. `https://graph-tool.skewed.de/` (November 2022), last accessed: November 2022
3. graphviz. `https://graphviz.readthedocs.io/en/stable/manual.html` (November 2022), last accessed: November 2022
4. How to design programs teachpacks: images.rkt. `https://docs.racket-lang.org/teachpack/2htdpimage.html` (November 2022), last accessed: November 2022
5. Module graph: Graphviz. `https://ocaml.org/u/2b073b70e60bfed059bd54da52ad3f28/ocamlgraph/2.0.0/doc/Graph/Graphviz/index.html` (November 2022), last accessed: November 2022
6. Plantuml. `https://github.com/plantuml/plantuml` (November 2022), last accessed: November 2022
7. Fu, C., Jin, G., Gao, J., Zhu, R., Ballesteros-villagrana, E., Wong, S.T.C.: DrugMap Central: an on-line query and visualization tool to facilitate

drug repositioning studies. Bioinformatics **29**(14), 1834–1836 (05 2013). https://doi.org/10.1093/bioinformatics/btt279

8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley, first edn. (1995)
9. Gansner, E.R., North, S.C.: An Open Graph Visualization System and Its Applications to Software Engineering. Softw. Pract. Exper. **30**(11), 1203–1233 (September 2000). https://doi.org/10.1002/1097-024X(200009)30:11¡1203::AID-SPE338¿3.0.CO;2-N
10. Johnston, S.: Behavioral models: Module behavior/markov-chain. `https://docs.racket-lang.org/behavior/Module_behavior_markov-chain.html` (November 2022), last accessed: November 2022
11. Kaufmann, M., Wagner, D. (eds.): Drawing Graphs: Methods and Models. No. 2025 in Lecture Notes in Computer Science, Springer (2001)
12. Kruja, E., Marks, J., Blair, A., Waters, R.: A short note on the history of graph drawing. In: Mutzel, P., Junger, M., Leipert, S. (eds.) International Symposium on Graph Drawing (GD). pp. 272–286. Lecture Notes in Computer Science, Springer (September 2001), `https://www.merl.com/publications/TR2001-49`
13. Metsker, S.J., Wake, W.C.: Design Patterns in Java. The Software Pattern Series, Addison-Wesley, first edn. (2006)
14. 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`
15. Shrestha, H.B.: Graph Visualisation Basics with Python, Part III: Directed Graphs with graphviz. Towards Data Science (June 2022), `https://medium.com/towards-data-science/graph-visualisation-basics-with-python-part-iii-directed-graphs-with-graphviz-50116fb0d670`, last accessed: November 2022