`fsmActions` – a Haskell library for finite state machines & FSM actions

Andy Gimblett
`haskell@gimbo.org.uk`

version 0.4.0 — October 21, 2009

Please note that §3 of this document is falling somewhat behind reality; however, the library's Haddock documentation is up to date and fairly complete – and the rest of this document is accurate and relevant.

1 Introduction

This is a library for representing and manipulating *finite state machines* (FSMs) in Haskell, with an emphasis on computing the effects of sequences of transitions across entire machines (which we call *actions*), and in particular investigating *action equivalences* between such sequences.

The motivation for writing this library is investigating models of user interfaces; in this context, states are implicit, transitions correspond to UI events (e.g. button presses), and sequences of transitions correspond to sequences of user actions. We're interested in comparing actions, which are the effects of sequences of transitions across the whole device (for example, noticing when some action is in fact an *undo*); for that we need a representation geared towards such comparisons – hence this library, and its idiosyncratic view of what's important about FSMs, which differs somewhat from alternative definitions the reader may be familiar with. E.g. we are unconcerned with start/accepting states (as found in finite automata), and – for now at least – output functions (as found in finite state transducers). We are interested only in finite machines with total transition functions, else we might call our machines labelled transition systems. Future versions of the library might add some or all of these features, particularly if someone else has an itch to scratch. More generally, perhaps someone else will find this useful, hence its release as a library.

The rest of this document is structured as follows. Section 2 introduces and defines the mathmatical abstractions on which the library is based. Section 3 outlines the current implementation in Haskell, though the full details are left to the Haddock documentation. Section 4 suggests future work. Section 5 describes the history of the library.

2 Definitions

2.1 Finite state machines

For the purpose of this library, a **finite state machine** (FSM) is a tuple $(S, \Sigma, \curvearrowright)$ where S is a finite set (of states), Σ is a finite set (of symbols) called the system's **alphabet**, and $\curvearrowright : S \times \Sigma \to \mathcal{P}(S)$ is a total function, called the machine's **transition function** (we have chosen to define $\curvearrowright$ as a total function to the powerset of states, as this unifies the treatment of deterministic and nondeterministic machines).

Given $p \in S$ and $a \in \Sigma$, we call the set of states in $\curvearrowright(p, a)$ the **transition set** for a at p, written $p_{\underset{a}{\curvearrowright}}$.

We introduce an infix shorthand for single **transitions**:

$$\forall\, p, q \in S \;\bullet\; \forall\, a \in \Sigma \;\bullet\; p \overset{a}{\curvearrowright} q \iff q \in p_{\underset{a}{\curvearrowright}}$$

Here, p and q are called the **source** and **destination** states of the transition, respectively, and a is called its **label**.

Example

Consider the FSM illustrated in figure 1(a), i.e. $(S, \Sigma, \curvearrowright)$ with:

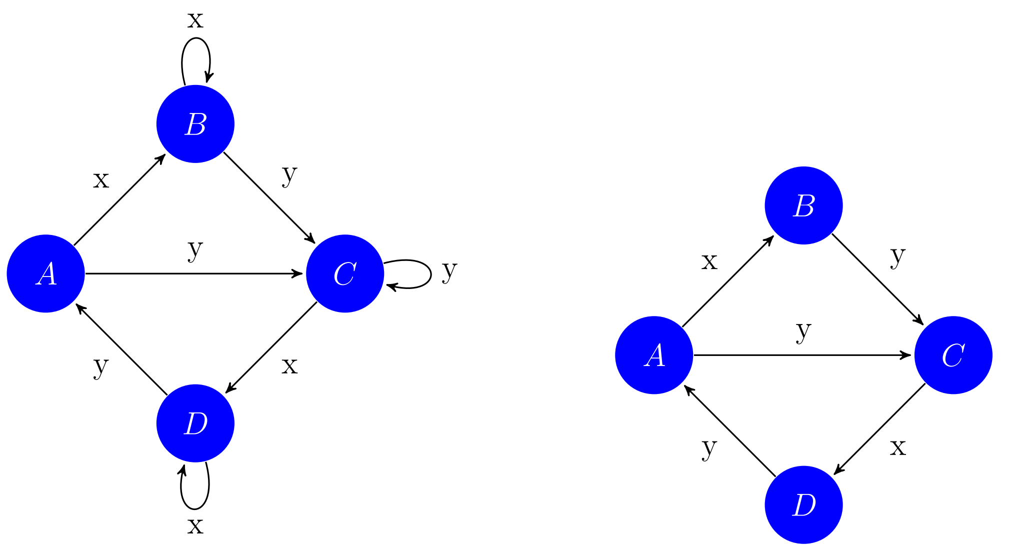
$$\begin{aligned}
S &= \{A, B, C, D\} \\
\Sigma &= \{x, y\} \\
\curvearrowright &= \{A \overset{x}{\curvearrowright} B, A \overset{y}{\curvearrowright} C, B \overset{x}{\curvearrowright} B, B \overset{y}{\curvearrowright} C, C \overset{x}{\curvearrowright} D, C \overset{y}{\curvearrowright} C, D \overset{x}{\curvearrowright} D, D \overset{y}{\curvearrowright} A\}
\end{aligned}$$

Note abuse of notation in enumerating $\curvearrowright$ here; properly it would written as

$$\curvearrowright \;=\; \{(A, x) \mapsto \{B\}, (A, y) \mapsto \{C\}, (B, x) \mapsto \{B\}, (B, y) \mapsto \{C\}, \cdots\}$$

So, e.g. $A_{\underset{x}{\curvearrowright}} = \{B\}$, $A_{\underset{y}{\curvearrowright}} = \{C\}$, etc.

This FSM includes a number of **self-loops**, e.g. $B \overset{x}{\curvearrowright} B$. Because $\curvearrowright$ is total, self-loops may in general be omitted from FSM representations without difficulty: provided S and Σ are known, the self-loops may be easily inferred. For example, figure 1(b) illustrates the



(a) A finite state machine (b) With self-loops removed

Figure 1: Two graphical representations of the same FSM

same FSM without self-loops, and we could write it out as:

$$\begin{aligned}
S &= \{A, B, C, D\} \\
\Sigma &= \{x, y\} \\
\curvearrowright &= \{A \overset{x}{\curvearrowright} B, A \overset{y}{\curvearrowright} C, B \overset{y}{\curvearrowright} C, C \overset{x}{\curvearrowright} D, D \overset{y}{\curvearrowright} A, \}
\end{aligned}$$

2.2 Nondeterminism

An FSM $(S, \Sigma, \curvearrowright)$ is **nondeterministic** (an NFSM) if the transition set for some word at some state has more than one member. Equivalently, if:

$$\exists\, p, q, r \in S \wedge \exists\, a \in \Sigma \bullet p \overset{a}{\curvearrowright} q \wedge p \overset{a}{\curvearrowright} r \wedge q \neq r$$

Otherwise the machine is **deterministic** (a DFSM).

Examples

Both machines in figure 1 are deterministic. Figure 2 illustrates a nondeterministic FSM where $S = \{E, F, G, H\}$ and $\Sigma = \{j, k\}$ (with implicit self-loops omitted, e.g. $E \overset{j}{\curvearrowright} E$). Here, for example, we have $E_{\underset{k}{\curvearrowright}} = \{F, G\}$, i.e. both $E \overset{k}{\curvearrowright} F$ and $E \overset{k}{\curvearrowright} G$.

In §2.1 we stated that self-loops may in general be omitted from FSM representations, and inferred as required. With NFSMs, some care must be taken. Consider the example in

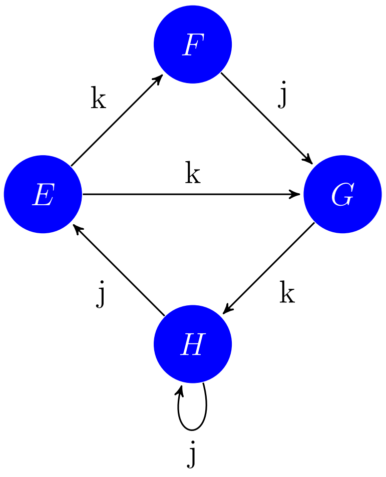


Figure 2: A nondeterministic FSM

figure 2; as well as four implicit self-loops, there is also an *explicit* self-loop $H \overset{j}{\curvearrowright} H$. This *cannot* be removed, because it is a source of nondeterminism, along with $H \overset{j}{\curvearrowright} E$.

2.3 Strings and destination sets

We adopt the standard definition of **strings** over an alphabet. Specifically, if Σ is an alphabet then Σ^*, the set of strings over Σ is the smallest set such that:

1. Σ^* contains the empty string: $\lambda \in \Sigma^*$
2. $w \in \Sigma^* \wedge a \in \Sigma \implies wa \in \Sigma^*$

Then the transition function $\curvearrowright : (S \times \Sigma) \to \mathcal{P}(S)$ may be lifted to the **string transition function** $\twoheadrightarrow : (S \times \Sigma^*) \to \mathcal{P}(S)$ as follows:

1. $\twoheadrightarrow (p, \lambda) = \{p\}$
2. $\twoheadrightarrow (p, wa) = \bigcup \{q_{\overset{\curvearrowright}{a}} \mid q \in p_{\overrightarrow{w}}\}$

Given $p \in S$ and $w \in \Sigma^*$, we call the set of states in $\twoheadrightarrow (p, w)$ the **destination set** for w at p, written $p_{\overrightarrow{w}}$.

As with single symbols, we introduce an infix shorthand for single **string transitions**:

$$\forall\, p, q \in S \;\bullet\; \forall\, w \in \Sigma^* \;\bullet\; p \overset{w}{\twoheadrightarrow} q \iff q \in p_{\overrightarrow{w}}$$

Given $p \in S$, two strings are said to be **destination equivalent** at p, written $\sim_p$, if their destination sets at p are identical:

$$\forall\, p \in S \;\bullet\; \forall\, w, x \in \Sigma^* \;\bullet\; w \sim_p x \iff p_{\overrightarrow{w}} = p_{\overrightarrow{x}}$$

Examples

Consider the DFSM in figure 1; here we have, for example $A \overset{xyxxx}{\twoheadrightarrow} D$ and $C \overset{xyxxx}{\twoheadrightarrow} B$.

Consider the NFSM in figure 2; here we have (for example) $E_{\overrightarrow{kkjjk}} = \{H, F\}$, so that both $E \overset{kkjjk}{\twoheadrightarrow} H$ and $E \overset{kkjjk}{\twoheadrightarrow} F$.

2.4 Actions and action equivalence

Given an FSM $(S, \Sigma, \curvearrowright)$, for every $w \in \Sigma^*$, w's **action**, written $\overrightarrow{w} \subseteq S \times \mathcal{P}(S)$ captures its effect across every state in S:

$$\overrightarrow{w} = \{(p, p_{\overrightarrow{w}}) \mid p \in S\}$$

Thus, an action captures the effect of a string of transitions, across all states in the machine.

An action is **deterministic** if, for every one of its pairs, the destination set found in the pair's second element has only a single member; otherwise it is **nondeterministic**. We state without proof that an FSM is deterministic if and only if all of its actions are deterministic.

Two strings are said to be **action equivalent**, written $\sim$, if their actions are identical:

$$\forall w, x \in \Sigma^* \bullet w \sim x \iff \overrightarrow{w} = \overrightarrow{x}$$

2.4.1 Examples

Consider again the FSM in figure 1. Here are some actions over this machine:

$$\begin{array}{rcl}
\overrightarrow{x} & = & \{(A,\{B\}),(B,\{B\}),(C,\{D\}),(D,\{D\})\} \\
\overrightarrow{y} & = & \{(A,\{C\}),(B,\{C\}),(C,\{C\}),(D,\{A\})\} \\
\overrightarrow{xx} & = & \{(A,\{B\}),(B,\{B\}),(C,\{D\}),(D,\{D\})\} \\
\overrightarrow{xy} & = & \{(A,\{C\}),(B,\{C\}),(C,\{A\}),(D,\{A\})\} \\
\overrightarrow{xxx} & = & \{(A,\{B\}),(B,\{B\}),(C,\{D\}),(D,\{D\})\} \\
\overrightarrow{xyy} & = & \{(A,\{C\}),(B,\{C\}),(C,\{C\}),(D,\{C\})\}
\end{array}$$

Notice that $xx \sim xxx$, for example, and that all of these actions are determinstic – as we would expect as the FSM is determinstic too.

Conversely, consider the nondeterministic FSM in figure 2. Here we have some nondeterminstic actions, e.g.

$$\begin{aligned}
\overrightarrow{j} &= \{(E,\{E\}),(F,\{G\}),(G,\{G\}),(H,\{E,H\})\} \\
\overrightarrow{jk} &= \{(E,\{F,G\}),(F,\{H\}),(G,\{H\}),(H,\{F,G,H\})\} \\
\overrightarrow{jkj} &= \{(E,\{G\}),(F,\{E,H\}),(G,\{E,H\}),(H,\{E,G,H\})\}
\end{aligned}$$

3 Implementation

Please note that this section is falling somewhat behind reality; however, the library's Haddock documentation is up to date and fairly complete.

3.1 `Data.FsmActions`

3.1.1 States, destination sets, actions, and words

The current Haskell implementation is focused strongly on actions, and makes some simplifying assumptions about FSMs. In particular, the members of the set S of states are integers, counting upwards from 0 (so e.g. in an FSM with 5 states, $S = \{0, 1, 2, 3, 4\}$. In the current implementation, S is not represented explicitly, but rather implicitly, as a property of the actions in the FSM (see below).

```
type State = Int
```

A destination set then consists of a set of integers, where all members of the set must be smaller than the number of states in the FSM. In the current implementation, we represent such sets as plain Haskell integer lists:

```
newtype DestinationSet = DestinationSet {
        destinations :: [State]
    } deriving (Eq, Ord, Show)
```

Then an action is just a list of destination sets; the index of each destination set within the list is the set's corresponding source state; for example, the first entry in the list contains the destination set for state 0. Contrast this with the definition in §2.4, where source states are explicitly paired with their destination sets.

```
newtype Action = Action {
        destinationSets :: [DestinationSet]
    } deriving (Eq, Ord, Show)
```

Actions may be constructed either directly using the `Action` constructor, or via convenience functions obviating the need to decorate destination lists with the `DestinationSet` constructor.

```
mkAction :: [[State]] -> Action
mkDAction :: [State] -> Action
```

Finally, we introduce a type, `Word`, for sequences of symbols.

```
newtype Word sy = Word [sy]
```

3.1.2 Finite state machines

A finite state machine is then a mapping from alphabet symbols to actions; each action tells us the effect, over the whole machine, of a particular singleton symbol. Note that this is parametric over the symbol type — typically we expect to use `Char`s or `String`s.

```
newtype FSM sy = FSM {
        unFSM :: M.Map sy Action
    } deriving (Eq, Ord, Show)
```

It's then easy to compute the machine's set of states (from the length of the first action found in the map, assuming they're all the same length – see §3.2) and its alphabet (just the keys of the map); but note that both of these results are lists, not sets.

```
states :: FSM sy -> [State]
alphabet :: FSM sy -> [sy]
```

For convenience, we expose the `lookup` function of an `FSM`'s underlying `Map`.

```
fsmAction :: Ord sy => sy -> FSM sy -> Maybe Action
```

3.1.3 Normalisation

An FSM which may be *normalised*, which ensures that all of its actions' destination sets are non-empty (empty ones become self-loops), are sorted and free from duplicates. Like well-formedness checks, this can be useful for cleaning data after I/O or conversion from other formats.

```
normalise :: FSM sy -> FSM sy
```

3.1.4 Computing actions/equivalences for strings

Given the actions for two strings w and x, `append` computes the action for the string wx formed by concatenating the strings. We map over every source state in the first action, and for every state in each destination set, we (`appendAtState`) look up the set of destination states reached from that state via the second action (`actionLookup`), collecting the results for each source state by flattening the list produced, sorting it, and removing duplicates.

```
append :: Action −> Action −> Action
appendAtState :: DestinationSet −> Action −> DestinationSet
actionLookup :: Action −> State −> DestinationSet
```

While this works nicely as an easily comprehensible canonical implementation of `append` — and is handy for testing purposes — we suspect a more efficient algorithm, probably based on a better representation than Haskell lists, will be found for future versions of the library.

It's then quite straightforward to compute the action of a given word (by folding `append` over the actions for the word's constituent symbols), and to test action equivalences on words. Note that since a word may include symbols not in the machine's alphabet, `action`'s return type (and much of the computation) necessarily has type `Maybe Action`.

```
action :: Ord sy => FSM sy −> Word sy −> Maybe Action
actionEquiv :: Ord sy => FSM sy −> Word sy −> Word sy −> Bool
```

3.1.5 Computing destination sets/equivalences for strings

Given a method to compute actions, computation of destination sets and destination equivalences is then straightforward (with the same comment regarding the use of `Maybe`).

```
destinationSet :: Ord sy => FSM sy −> State −> Word sy −> Maybe
    DestinationSet
destinationEquiv :: Ord sy => FSM sy −> State −> Word sy −> Word sy −> Bool
```

3.1.6 The identity action

An FSM's **identity action** is the action which maps all states back onto just themselves. Because this depends on the size of the FSM, there is no general identity action for all FSMs — which is the primary reason `FSM` is not an instance of `Data.Monoid`.

```
fsmIdentity :: FSM sy −> Action
identity :: Int −> Action
```

3.1.7 Determinism checks

We can test if an action or FSM is determinstic.

```
isDAction :: Action -> Bool
isDFSM :: FSM sy -> Bool
```

3.2 `Data.FsmActions.WellFormed`

In our implementation, an FSM is well-formed provided:

- Every action is the same length (the number of states in the machine).
- None of those destination sets contain out-of-range destination state values (i.e. negative or too high).
- It is (at least) *weakly connected* (see §3.5); otherwise, it is *disconnected*. (Note that an FSM with no nodes is considered to be disconnected.)

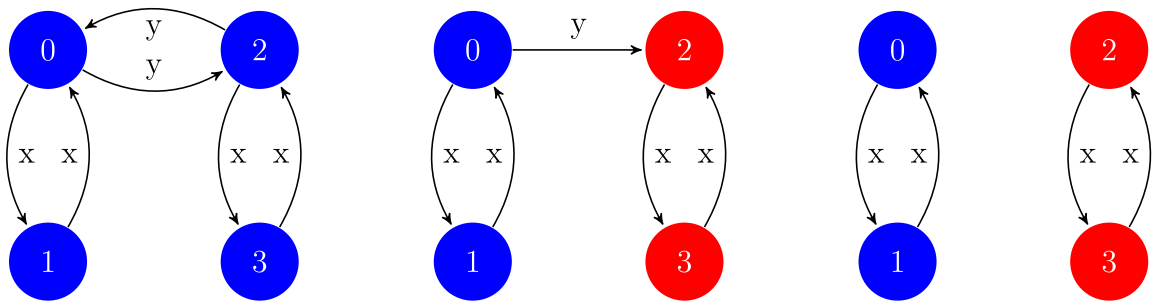
An FSM should be checked for well-formedness after construction (which typically involves some I/O) using the `isWellFormed` function, which yields a `WellFormed` value encoding any problems found.

```
data WellFormed sy = BadLengths [(sy, Int)]
                   | BadActions [(sy, Action)]
                   | Disconnected [[State]]
                   | WellFormed
                   deriving (Eq, Show)
isWellFormed :: Ord sy => FSM sy -> WellFormed sy
```

If all actions do not have the same length, the `BadLengths` constructor contains a full list of (symbol, action length) pairs so discrepancies may be identified. Similarly, if some actions contain out-of-range destinations, the `BadActions` constructor lists the offending actions and their corresponding symbols. If the FSM is not at least weakly-connected, the `Disconnected` constructor lists its weakly-connected components. Finally, if all is well, the `WellFormed` constructor is used.

3.3 `Data.FsmActions.FsmMatrix`

Matrix-based serialisation/deserialisation of FSMs. Here an FSM is represented as a single matrix, with one (newline-separated) row per state, and one (whitespace-separated) column per symbol. The first row contains symbol names. Subsequent rows contain destination sets as comma-separated lists of states. State/row numbering starts at 0.



(a) `strong` (b) `weak` (c) `disc`

Figure 3: Strongly connected, weakly connected, and disconnected FSMs

Note that this representation currently suffers from the restriction that symbol names may not contain whitespace. TODO: check, and then consider fixing this (or at least providing a workaround, e.g. allowing double-quoted symbol names).

TODO: Say more here, and give examples.

3.4 `Data.FsmActions.ActionMatrix`

Actionmatrix-based serialisation/deserialisation of FSMs. Here an FSM is represented as a collection of binary adjacency matrices, one per action.

TODO: Say more here, and give examples.

3.5 `Data.FsmActions.Graph`

`fsmToFGL` converts an `FSM` into an `fgl`[1] graph with labelled edges and unlabelled nodes. The `SelfLoops` data type encodes whether to keep all self-loops, or trim all those which are not sources of nondeterminism — see §2.2.

```
data SelfLoops = Keep | Trim deriving Eq
fsmToFGL :: FSM sy -> SelfLoops -> Gr () sy
```

`strongCCs` and `weakCCs` compute an `FSM`'s strongly-connected and weakly-connected components respectively, where the WCCs are the SCCs of an *undirected* graph of the machine.

```
strongCCs :: Eq sy => FSM sy -> [[State]]
weakCCs :: Eq sy => FSM sy -> [[State]]
```

[1]Functional Graph Library: `http://hackage.haskell.org/package/fgl`

Consider figure 3. `strong` is strongly-connected; `weak` is weakly connected but not strongly connected; `disc` is not even weakly-connected (it is *disconnected*):

$$\begin{array}{rcccccc} \text{strongCCs(strong)} & = & \text{weakCCs(strong)} & = & \text{weakCCs(weak)} & = & \{\{0,1,2,3\}\} \\ \text{strongCCs(weak)} & = & \text{strongCCs(disc)} & = & \text{weakCCs(disc)} & = & \{\{0,1\},\{2,3\}\} \end{array}$$

For graph output, `fsmToDot` converts an `FSM` to a `DotGraph`[2] with labelled edges and integer-labelled nodes, and all self-loops removed except sources of nondeterminism (see §3.2). It's fairly trivial; more sophisticated graph-drawing could be achieved by rolling your own based on this code. Similarly, `fsmToGML` converts an `FSM` to a GML[3] format graph.

```
fsmToDot :: (Ord sy, CleanShow sy) => FSM sy -> DotGraph
fsmToGML :: CleanShow sy => FSM sy -> Doc
```

Note that both of these functions require the symbol type to be an instance of the `CleanShow` typeclass. Defined in this same module, this subclasses the standard `Show` typeclass, providing a `cleanShow` function which acts like `show` except that it special-cases `String` and `Char` so they are not quoted. For example, while `show "s"` is `"s"`, `cleanShow "s"` is just `s`, which is more like what you want for graph labels. Anyway, labels of any type which is an instance of `Show` should 'just work' with the default implementation.

4 Future work

4.1 More examples

Add more examples of the various representations.

4.2 Quickcheck tests

We currently have some HUnit-based unit tests, but there are some properties which could usefully be QuickChecked, for example:

$$\forall w, x, y \in \Sigma^* \bullet w = xy \implies \overrightarrow{w} = append(\overrightarrow{x}, \overrightarrow{y})$$

4.3 Faster representations

Plain Haskell lists are surely not the fastest representation we could use; in particular, appending actions is based on many lookups of destination sets within actions, via `(!!)`,

[2]From the `graphviz` library: `http://hackage.haskell.org/package/graphviz`

[3]Graph Modelling Language: `http://en.wikipedia.org/wiki/Graph_Modelling_Language`

which is $O(n)$. A tree-based representation should give us $O(\log n)$. An unboxed representation (since, ultimately, we're just looking up integers) would be even better — but as destination sets vary in size, it's not currently clear how to implement that. Memoization may also have a role to play.

4.4 Interfaces to other packages

`halex`?

5 History

5.1 version 0.1 – June 30, 2009

First version of the library, Cabal-ised and released to Hackage. Naive implementation based on plain Haskell lists. Support for `ActionMatrix` and `FsmMatrix` serialisation formats. HUnit for unit tests.

5.2 version 0.2.0 – July 18, 2009

Cleaned up `WellFormed`, and broke it out into its own module. Added `ActionMatrix` output and some test cases. Added `fgl` interface, including computation of strongly/weakly-connected components. Added `graphviz` interface. Conform to Haskell package versioning policy[4].

5.3 version 0.3.0 – July 20, 2009

Added GML (Graph Modelling Language) output. Unified FGL, GraphViz and GML modules into `Data.FsmActions.Graph`. Fixed for latest `graphviz` library.

5.4 version 0.4.0 – October 21, 2009

Added FGL (Functional Graph Library) input. Added `FsmEdges` input/output (graph format produced by Mathematica). Added 'ActionSpecs' to `FsmActions` I/O formats, so that individual `ActionMatrix` files don't have to be explicitly identified. Added intelligent/unified load/save (in `Data.FsmActions.IO`). Fixed for latest `graphviz` library. Improved error handling. Section §3 of this documentation is now falling behind, however.

[4]`http://www.haskell.org/haskellwiki/Package_versioning_policy`