rFSM Cheatsheet
(beta5)

Specifying rFSM Models

Synopsis

```
return rfsm.state {
   ...
}
```

Decription

A rFSM model is contained in a file that returns the toplevel state.

Toplevel rFSM Configuration

The following fields may defined in the toplevel state:

- `getevents()`: function that will be executed during step/run and must return a table of new events (index part only).
- `dbg`, `info`, `err`: log functions. Must accept multiple parameters to log.

States

States are the principle building block of rFSM Statecharts.

Synopsis

```
rfsm.state {
   entry=function() do_this() end,
   exit=function() do_that() end,

   substate1=rfsm.state{...}, ...
   transition{...} ...
}
```

Fields

- `entry(fsm, state, 'entry'`: entry function
- `exit(fsm, state, 'exit'`: entry function
- `doo(fsm, state, 'doo'`: doo function/coroutine (only permited for leaf states)

Description

States model distinguishable conditions and encapsulate behavior. If a state has substates it is called a **composite state**. Otherwise it is called a **leaf state**. If a transition ends on a composite state, this state must define an initial transition. **Common error:** forgetting the commas.

Transitions

Transition connect states.

Synopsis

```
rfsm.transition{ src=''stateX'', tgt=''stateY'',
                 events={"e_1", "e_2"} }
rfsm.trans{ ... } -- short form
```

Fields

- `src=<state-ref>`: source state reference
- `tgt=<state-ref>`: target state reference
- `events={event1, event2, ...}` : list of events of which each may trigger the transition (`or`).
- `guard`: side-effect free function that returns `true` or `false`. If false will inhibit transition.
- `effect`: function that is execute when transition is taken.
- `pn=<number>`: Priority number. Transitions with larger numbers take priority in case of conflicting transitions.

Description

Transitions define how the FSM changes state when events are received. `<state-ref>` can be *absolute* "root.s1.s2", *local* "s1" or *relative* (".subst1.subst2"). Transitions may cross state boundaries or be layered on top of deeper nested states by parent states using the relative notation.

Connectors

Synopsis

```
rfsm.connector{}
rfsm.conn{} -- short form
```

Description

Connectors can be used to build composite transitions by chaining multiple connectors, or for defining different entry and exit points of composite states. *Note:* connectors only permit to build more sophisticated transitions, the connector is *never* active!

Loading, Instatiating and Advancing

Loading and Instatiating

Synopsis

- `model=rfsm.load(filename)` Load a rFSM Model from a file.
- `rfsm.init(model)` Instatiate and validate a rFSM model

Advancing rFSM instances

Synopsis

- `rfsm.step(fsm)`: `step` a rFSM instance. This will execute at most one transition or one doo cycle.
- `rfsm.step(fsm, N)`: `step` as above, but will perform at most `N` transitions or doo cycles.
- `rfsm.run(fsm)` will run step until there are no new events and no active `doo` function.
- `rfsm.send_events(fsm, event1, event2, ...)`: send events to the internal fsm queue.

Description

The basic step consists of the following: **1.** retrieve new events using `getevents` hook. **2.** Find enabled transitions starting from `root` state to active leaf. **3.** If enabled transition found → execute. **4.** elseif active `doo` → run it. **5.** discard current events.
After each `step` exactly **one** leaf-state will be active.

Miscellaneous

Tracing

Enable state entry and exit debug messages:

```
require "rfsmpp"
fsm.dbg=rfsmpp.gen_dbgcolor("fsmX",
                            { STATE_ENTER=true,
                              STATE_EXIT=true }, false)
```

Most important debug IDs are:

```
STATE_ENTER, STATE_EXIT, EFFECT, DOO, EXEC_PATH,
ERROR, HIBERNATING, RAISED, TIMEEVENT
```

Time Events

```
require "rfsm_timeevent
rfsm_timeevent.set_gettime_hook(gettime_func)
return rfsm.state {
   trans{ src="sA", tgt="sB", events={ "e_after(3)"} },
   ...
}
```

Description

The `rfsm_timeevent` module enables relative time events. After loading, a suitable `gettime` function must be set. This function returns two values: current seconds and current nanoseconds. Available functions: `rtp.clock_gettime` (from the rtp module, see section *Links*), `rtt.getTime` from RTT-Lua, or for second resolution events the Lua built-in:

```
function gettime() return os.time(), 0 end
```

Note: These timeevents only work for periodically triggered components.

Sequential AND state

Synopsis

```
seqAND = rfsm.seqand {
   subfsm1=rfsm.init(rfsm.load("subfsm1.lua")),
   subfsm2=...
}
```

Fields

- `order`: table of substate names that indicate the desired execution order. Not mentioned states are executed last.
- `andseqdbg`: if true print debug info
- `step`: number of steps to advance each time.
- `run`: if true, don't step but `run`.
- `idle_doo`: doo flag to be returned by seqand yield.

Description

Specialized state that `step`'s or `run`'s the initialized substates in a serialized manner inside the `doo` function of the seqand state.

Complete Example

```
local state, trans, conn = rfsm.state, rfsm.conn, rfsm.trans
return rfsm.state {
   dbg=rfsmpp.gen_dbgcolor("sample-fsm")
   entry=function() enable_robot() end,
   entry=function() disable_robot() end,

   stopped = state{},

   moving = state {
      entry=function() start_motor() end,
      exit=function() stop_motor() end,

      doo = function()
        while true do
           compute_next_pos()
           rfsm.yield(true)
        end,
   }
   trans{ src="initial", tgt="stopped" },
   trans{ src="stopped", tgt="moving", events={"e_start"} },
   trans{ src="moving", tgt="stopped", events={"e_stop"} },
}
```

Links

```
www.orocos.org/rfsm
http://www.orocos.org/wiki/orocos/toolchain/LuaCookbook
https://github.com/kmarkus/rtp
```