Introduction To rFSM

Gianni Borghesan
KU Leuven
October 6, 2021

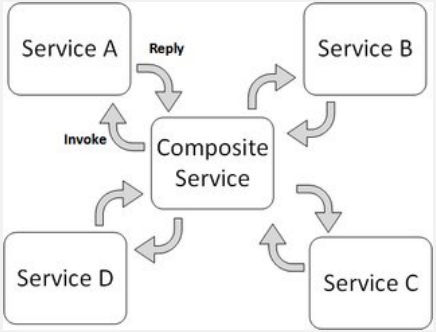
Outline

Coordination of subsystems

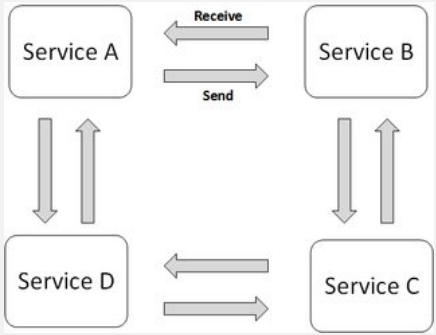
- Most of you see a single flow of computation in complex applications.
- In reality, computation and data flow are asynchronous.
- Computation needs to coordinate among themselves or being coordinated.
- Most of the sub-systems have more than one logic state;
  the system must traverse a set of state.

Types of coordination

Orchestration



Choreography



Images from https://stackoverflow.com/questions/4127241/orchestration-vs-choreography

Implementation of the coordination

- The easiest way is to use a state automata;
- For the applications at hand we can use a finite state automata (with some tricks)
- Other possibilities
  - behaviour tree – but they target other goals.
  - Petri Nets – for concurrent applications.

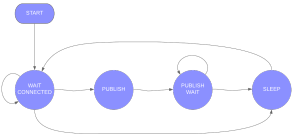


Image from https://docs.particle.io/

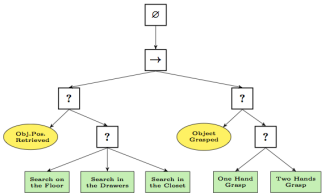


Image from https://en.wikipedia.org/

Orchestration or Choreography

- (Orchestration) Provide a single supervisor that coordinates all the transitions; scale up hierarchically, much easier to understand
- (Choreography) Peer-to-Peer, more scalable, less intuitive.

Example

Four systems (Service A to D)

- with states `IDLE`, `RUN`,
- starting all in `IDLE`,
- with conditions that B requires A in `RUN` to go to `RUN`.

Questions:
how would look like the states and transitions of the two coordinations?

Orchestration or Choreography

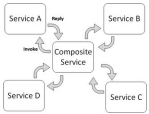
Orchestration

In the supervisor:

S1 Ask A to `RUN`;
when acknowledged

S2 Ask B to `RUN`;
when acknowledged,

...

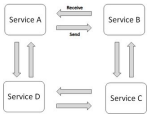


Choreography

In each system:

`IDLE` when my predecessor acknowledge a `RUN`, I go in ran, and then

`RUN` I acknowledge to my successor that I am in `RUN`.



Orchestration or Choreography

Orchestration

There is a specific system that takes care of to invoke the changes of state in the coordinated systems;
all systems that are coordinated must provide him with the state changes.

Choreography

Each system has a small supervisor, or the state automata is embedded inside the component. Small coordination efforts.

In my opinion ...

Orchestration – a single supervisor (or a hierarchy) – is good for most **centralised**, **small**, applications.

Finite state machine implementations

- **rFSM**: lua-based module, used with Orocoos.
  https://github.com/kmarkus/rFSM
- **Smach**: python-based module, used with ROS.
  http://wiki.ros.org/smach

Which feature I use?

- Mainly using Moore Machines - the "output" depends only by the current state.
- with some extensions that makes them more similar to **state charts**
  - Hierarchical states,
  - Additional states (e.g. a counter) that breaks the concept of finite states
  - (rarely) parallel states - like having more state machines in a single state.

Outline

rFSM

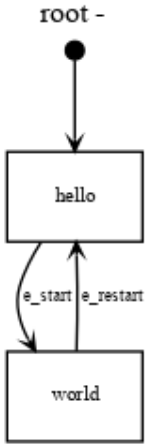
FSMs are characterised by

- states; characterised by
  - `entry`
  - `doo` **(do not use)**
  - `exit` **(use only in special cases)**
- transitions
  - On a given event, go from a state to another
  - guards **(do not use)**
  - effects **(do not use)**
- an initial transition

rFSM

```
return rfsm.state {
  hello = rfsm.state { entry=function() print("hello") end },
  world = rfsm.state { entry=function() print("world") end },

  rfsm.transition { src='initial', tgt='hello' },
  rfsm.transition { src='hello', tgt='world', events={ 'e_start' } },
  rfsm.transition { src='world', tgt='hello', events={ 'e_restart' } },
}
```



Practical guidelines: in the state machine

- use non-blocking function, possibly without return,
- Use only the `entry` function:
  - A single effect for each transition
  - You can also use self transitions
- use `exit` function for:
  - restore a state (can be done also with an additional state)
  - getting out from a sub-state machine (you do not know exactly from which state you get from)
- do not use
  - `doo`,
  - guards: this should be deligated to a monitoring mechanism
  - effects: this can be used only in place of using functions in states

Practical guidelines: in the systems

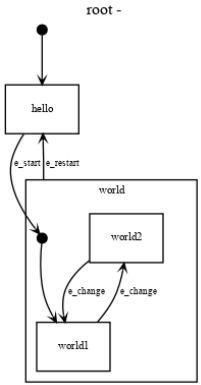
- prepare operations that can query the change of state/request functionality
- instrument your system to fire events, or
- deploy *monitors* that observe the environment or other systems

Hierarchical states

```
return rfsm.state {
 hello = rfsm.state { entry=function() print("hello") end },
 world = rfsm.state {
  exit=function() print("leaving worlds") end,
  world1 = rfsm.state { ask_to_change()},
  world2 = rfsm.state { ask_to_change()},
  rfsm.transition { src='initial', tgt='world1' },
  rfsm.transition { src='world1', tgt='world2', events={ 'e_change' } ↩
      ↪},
  rfsm.transition { src='world2', tgt='world1', events={ 'e_change' } ↩
      ↪},},
 rfsm.transition { src='initial', tgt='hello' },
 rfsm.transition { src='hello', tgt='world', events={ 'e_start' } },
 rfsm.transition { src='world', tgt='hello', events={ 'e_restart' } }
 }
```

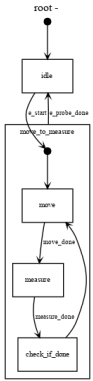
Hierarchical states

What does it happen when in rFSM is in `world` and receives `e_restart`?



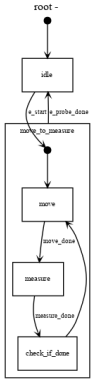
Infinite states

```
return rfsm.state {
  idle = rfsm.state {
  entry=function() idle_config() end,
  exit=function()  cleanup_controller() end   },
  move_to_measure = rfsm.state {
    entry=function()
      iteration=1 ; n_elements, frames=load_probe_poses ()
    end,
    exit=function() cleanup_controller_emergency()   end,
    move = rfsm.state {
      entry=function() move_config(frames[iteration]) end,
      exit=function()  cleanup_controller() end   },
    measure = rfsm.state { do_measure(iteration)},
```



Infinite states (Cont.)

```
14      check_if_done = rfsm.state {
15        entry = function(fsm) if iteration+1>n_elements
            then rfsm.send_events(fsm, "e_probe_done")
            else iteration=iteration+1  end end },
      rfsm.transition { src='initial', tgt='move' },
      rfsm.transition { src='move', tgt='measure', events={'↩
          ↪move_done'}},
20    rfsm.transition { src='measure', tgt='check_if_done', ↩
          ↪events={'measure_done' }},
      rfsm.transition { src='check_if_done', tgt='move' }, },
    rfsm.transition { src='initial', tgt='idle' },
    rfsm.transition { src='idle', tgt='move_to_measure', ↩
        ↪events={ 'e_start' } },
    rfsm.transition { src='move_to_measure', tgt='idle', ↩
        ↪events={ 'e_probe_done' } },
```



Outline

Setup

- Tested in Ubuntu 20.04 + Noetic + Orocos (rtt_ros_integration)
- I assume that you already have Orocos workspace sourced

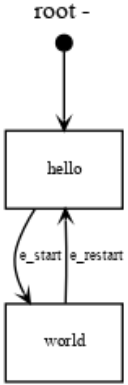
```
mkdir -p ws_rfsm/src
cd ws_rfsm/src
git clone --recursive https://github.com/gborghesan/rtt_ros_utilities.↩
    ↪git
git clone --recursive https://github.com/gborghesan/oro_extra.git
cd ..
catkin_make
source devel/setup.bash
```

- May be that you miss a couple of library to compile everything...
- you can always use the bare bone library as described in the documentation

rFSM utilities – rfsm-sim

Allows to simulate the execution of the fsm from command line;

```
$ rosrun rfsm rfsm-sim ex1.lua
Lua 5.1.5  Copyright (C) 1994-2012 Lua.org, PUC-Rio
rFSM simulator v0.1, type 'help()' to list available↩
    ↪ commands
INFO: created undeclared connector root.initial
> step()
  hello
  active: root.hello(done)
  queue:  e_done_at_root.hello
> step()
  active: root.hello(done)
  queue:
> se("e_start")
> step()
```



rFSM utilities – `rfsm-rviz`

- It makes the figure;
- uses the graphiz lua module for **lua 5.1**; normally you install the 5.2 from debian. There is a binary in the oro-extra repository.

```
$ rosrun rfsm rfsm-viz -h
rfsm-viz <options> -f <file> generate different rFSM representations.
options:
-f <fsm-file>           fsm input file
...

$ rosrun rfsm rfsm-sim ex1.lua -pdf
```

Outline

Deployment

- there is a component written in lua alredy ready.
- it can generate a dot graph with the current state highlighted.
- as also a port to get the state

see https://github.com/gborghesan/oro_extra/blob/master/lua/rtt_components/fsm_component.lua

Deployment

```
fsm_comp_dir = rtt.provides("ros"):find("oro_extra") .. "/lua/rtt_components"
depl:loadComponent("Supervisor", "OCL::LuaComponent")
sup = depl:getPeer("Supervisor")
sup:exec_file(fsm_comp_dir.."/fsm_component.lua")
sup:getProperty("state_machine"):set("fsm_definition.lua")
sup:getProperty("additional_code"):set("fsm_extra.lua")
sup:getProperty("viz_on"):set(false)
sup:addPeer(depl)
sup:addPeer(controller)
sup:configure()
--at this point additional properties are available
sup:getProperty("pose_file"):set(json_file_poses)
sup:start()
depl:connect("controller.eventPort","Supervisor.events",cp)
```

state_machine file

- Like the ones we saw before
- put as less as possible code inside (only function call defined in the next file)

additional_code file

- defines all the addittional properties/ports of the component
- define the functions called in the hooks of the state machine. Mainly, they should be calls to operations of other components.

```
tc           = rtt.getTC()
depl         = tc:getPeer("Deployer")
controller    = depl:getPeer("controller")
monitor    = depl:getPeer("controller")
--add to the component interface
json_pose_prop=rtt.Property("string","pose_file","file with probing motion json file")
tc:addProperty(json_pose_prop)
--functions
function load_probe_poses()
```

additional_code file (Cont.)

```
  -- load the poses from a json file to vector, returns # of poses and↩
      ↪ the vector
end

function idle_config()
    controller:readSpecification("idle__con_config.lua")
    monitor:readSpecification("idle_mon_config.lua")
    monitor:start()
    controller:configure()
    controller:initialize()
    controller:start()
end
```

ROS integration

It is mainly a component and a node:

- a small component (`event_echo`) that echo from topic to a port, to be connected to the event port; it converts from ROS string to normal string.
- a ROS Node `event_sender` with a minimal GUI to generate events.

https://github.com/gborghesan/python_gui

ROS integration – Event Sender

- It allows an user to generate event for *e.g.* start a movement sequence, stop, change operational mode,....
- It is configured with a simple xml files
- every time a button is clicked, a string is written to a topic, from which the `event_echo` is reading.

```
<list>
  <button name='Start' event='e_start' tooltip='↩
      ↪start the robot'/>
  <button name='Stop' event='e_stop'/>
</list>
```