Implementing Long Running Business Processes

Karolis Petrauskas

Co-founder @ Erisata

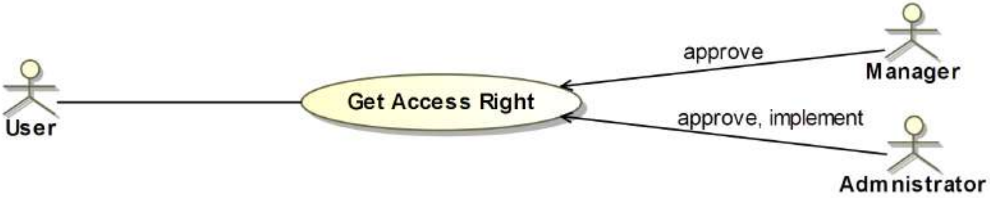
CodeBEAM STO 2019

Business Processes

We consider a business process a process, that is

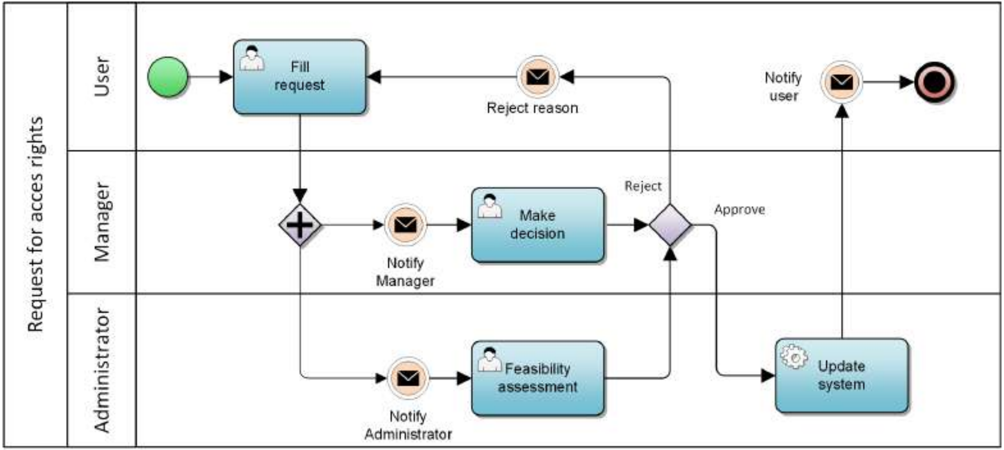
- Understood by the business people (avoids technical details);
- Of high abstraction level (coordinates interaction between multiple parties);
- Crosses several applications (often viewed as a layer of integration platform);
- Takes from milliseconds to years to complete.

Business processes are sometimes implemented using BPM solutions.



Business Processes | Example

Users often define business processes as BPMN diagrams.



Business Processes | In Erlang/OTP?

We found no BPM implementation in Erlang/OTP.
Maybe the standard libraries are enough?

$$\mathit{BusinessProcesses} \cap \mathit{ErlangProcesses} \neq \emptyset ?$$

Erlang/OTP has a number of features making it a good basis for a BPM:

- Every separate **activity runs as a process**, including the business processes.
- Finite **state machines** are sometimes used to implement the business processes, Erlang/OTP supports that by
  - `gen_statem` (previously `gen_fsm`);
  - `plain_fsm` and other.
- **Fault tolerance**, process isolation and high performance.
- **Test frameworks** (especially the Common Test).

Business Processes | What's Missing in Erlang/OTP

The following seems to be missing for implementing business processes:

- Persistence should be done explicitly (sometimes tedious).
- Process migration in a cluster (processes are active, and have side effects).
- Better support for complicated state machines (a lot of states).

We have implemented a library / framework to handle that:

https://github.com/erisata/eproc_core

Main features it provides:

- Automatic persistence on each transition.
- A process registry (distributes processes in a cluster).
- Support for **structured states** (nested and orthogonal).
- Designed for devs and admins.

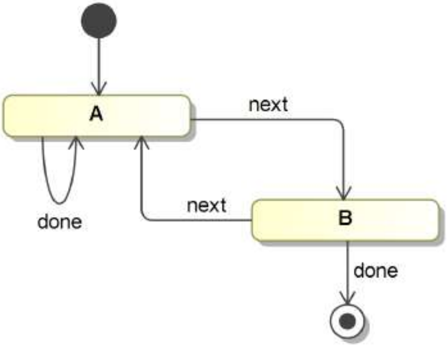
Finite State Machine

A state machine can be described as a relation

$$\mathit{States} \times \mathit{Events} \rightarrow \mathit{States} \times \mathit{Effects}$$

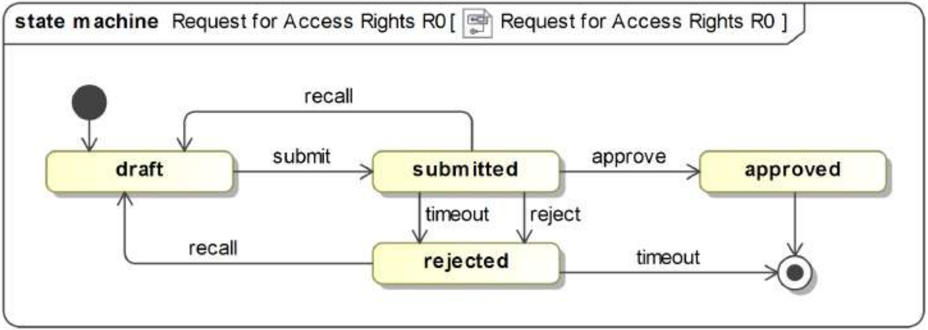
More detailed descriptions are sometimes useful.

- We have added semantics to the state names in the FSM.
- The added semantic is inspired by the FSMs in the UML.
- Structured states allow to manage timers, keys, state entries in more declarative way.



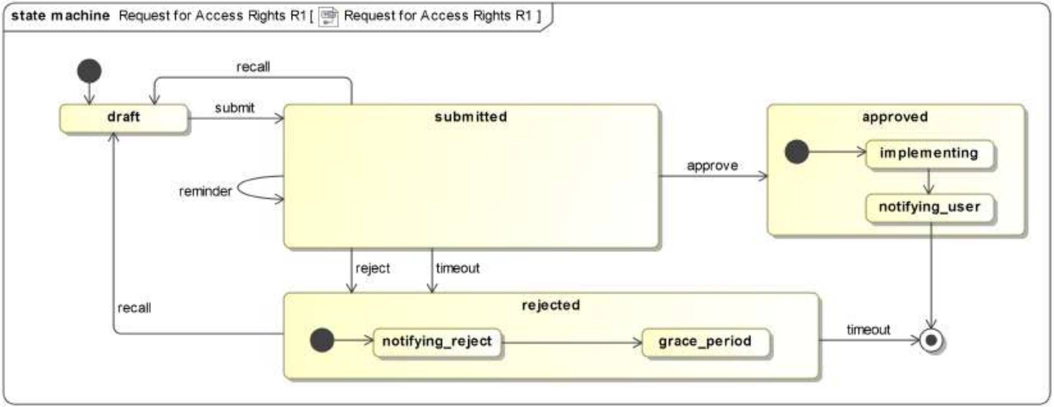
FSM | Example

The BPMN diagrams rarely maps 1-1 to the state machine.



The FSM notation helps to consider more *Event* $\times$ *State* combinations – edge cases.

FSM | Nested States



FSM | Nested States – pattern matching

State entry is usually performed in multiple steps, allowing to **decouple the states**:

```
handle_state(approved, {entry, submitted}, StateData) ->
    {next_state, {approved, implementing}, StateData};

handle_state({approved, implementing}, {entry, submitted}, StateData) ->
    {ok, StateData};
```

Event handling by states can be implemented via pattern matching:

- Handle all substate state events (state entry is not called in this case):

```
handle_state({approved, _}, {event, recall}, StateData) ->
    % It is too late to perform the recall.
    {same_state, StateData};
```

- Handle an event particular substate:

```
handle_state({approved, implementing}, {timer, timeout}, StateData) ->
    {final_state, terminated, StateData};
```

FSM | Nested States – scopes

Nested states are useful to limit **scope of timers**, they are cancelled automatically when the FSM exits the specified scope:

```
handle_state(...) ->
    ...
    ok = eproc_timer:set(step_retry,  1000, retry, {approved, implementing}),
    ok = eproc_timer:set(step_alarm, 10000, alarm, approved),
    ...
```

The same can be done with additional **keys**, that can be used to locate the process while being in the specified scope (`approved`, in this case):

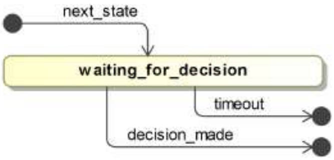
```
handle_state(...) ->
    ...
    ok = eproc_router:add_key({impl_id, ImplId}, approved),
    ...
```

Similar concept of keys is used in BPEL, although usually keys have the scope `'_'`. The timer scoped are used extensively in practice.

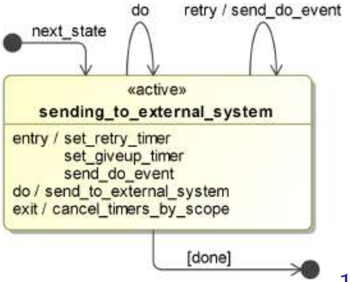
FSM | Active and Passive States

- It appears to be a good practice to separate active and passive states (especially for error handling):
  - Passive states are used to wait for some external events (or timers).
  - Active sates are used to perform some actions (usually involving external resources).
- In order to manage errors, retry and giveup timers are used in active states.
- The active state is left when the do action is successfully completed.
- It appears to be a bad practice to perform an action on a transition (trigger is lost on error).

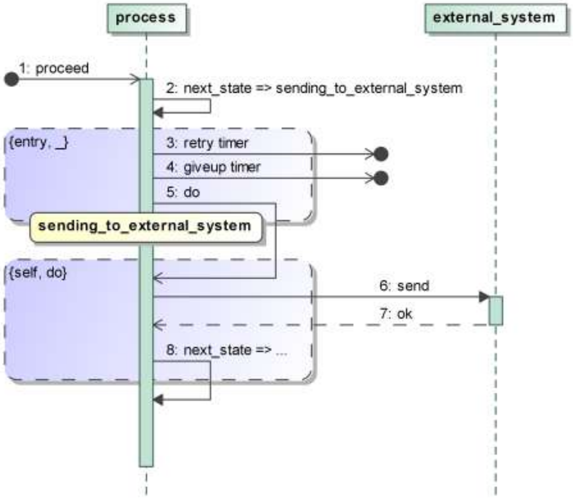
Passive state:



Active state:

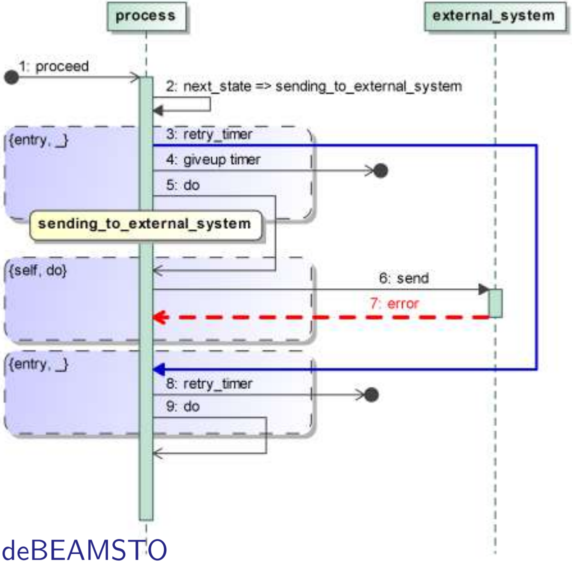


FSM | Active States – separating transactions



- Only the timers and the self event is sent on the state entry.
- The actual operation is performed on the self event do.
- If the operation was successful, go to the next state.

FSM | Active States – handling unexpected errors



- On error, be it a crash, or `{error, Reason}` returned, nothing is done usually.
- The retry timer will be used to retry the action.
- It will set the next retry timer and send the second `do` action.
- The `giveup` timer can be used to handle infinite retry loops.

FSM | Active States – generic handler

The pattern of the active states is very common,
therefore a generic function was defined to handle that.

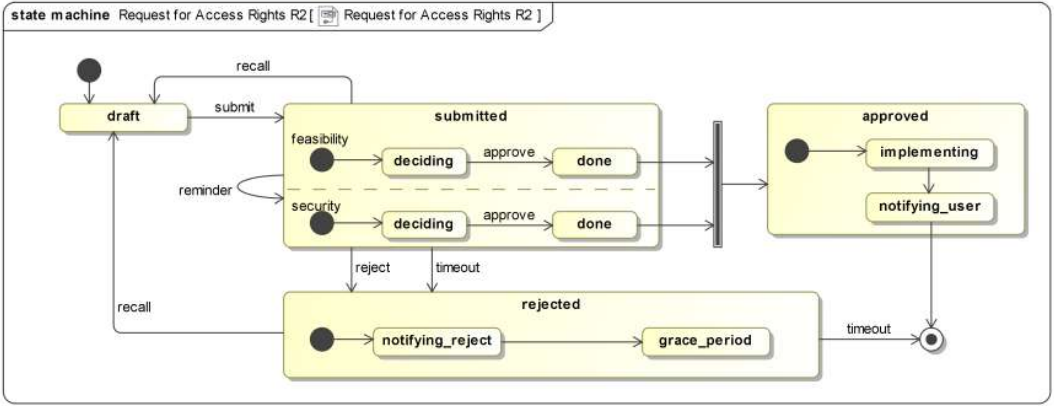
```
handle_state(s, {entry, _}, D) ->
  % Set the retry and giveup timers,
  % send the DO event.
  {ok, D};
handle_state(s, {self, act}, D) ->
  case do_act(D) of
    {ok, ND} -> {next_state, Next, ND}
    _        -> {same_state, D}
  end;
handle_state(s, {timer, retry}, D) ->
  {next_state, s, D};
handle_state(s, {timer, giveup}, D) ->
  {next_state, Other, D};
```

```
handle_state(s, T, D) ->
  gen_active_state(s, T, D, #{
    do     => {act, fun do_act/1},
    retry  => {r, 100, retry},
    giveup => {g, 500, giveup, Other},
    next   => Next
  });
```

This approach has made state definitions more compact, and declarative.

FSM | Orthogonal States

Some states do not have a strict ordering of states, like `submitted` bellow.



FSM | Orthogonal States

The orthogonal states can be represented as tuples with arity > 2, e.g.

```
{submitted, deciding, done}
```

Records can be used for this conveniently:

```
#submitted{
  feasability = deciding,
  security    = done
}
```

```
-record(submitted, {
  feasability = '_' :: deciding | done
  security    = '_' :: deciding | done
}).
-type submitted() :: submitted |
                     #submitted{}.
```

Two ways to specify next state with orthogonal states:

```
{next_state,
    State#submitted{security = done},
    NewData}
```

```
{next_state,
    #submitted{security = done}
    NewData}
```

The first case will re-enter states in both regions, while in the second case, the `{entry, _}` callback will not be called in the `feasibility` region.

FSM | Orthogonal States – scopes

Timers can be defined also in the orthogonal states (as well as keys and other):

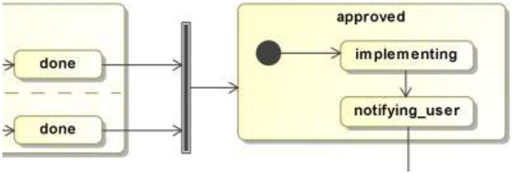
```
eproc_timer:set(step_retry,  1000, retry,  #submitted{feasability = deciding}),
eproc_timer:set(step_giveup, 5000, giveup, submitted),
```

The actual scopes in the case of `retry` is `{submitted, deciding, '_'}`.

Events can be handled for the entire state (using the regular pattern matching):

```
handle_state(#submitted{}, {event, reject}, Data) ->
    {next_state, rejected, Data};
```

Note, the next state in this case is `rejected` instead of `{rejected, notifying_reject}`. This allows to isolate the states.




```
handle_state(#submitted{
                feasability = done,
                security    = done
            }, {entered, _From}, D) ->
    {next_state, approved, D}
```

Order of callbacks:

1. **event** – represents a trigger of a transition.
2. **exit** (recursive) – from the deepest state, to the outermost, that is left.
3. **entry** (recursive) – from the outermost state, to the deepest. Only the state that is entered is named (not '_').
4. **entered** – called when all the states are entered. Here the 1st argument has all the states specified.

FSM | Orthogonal States – sequence of callbacks

A pseudocode listing of callbacks invoked when performing a transition:

$$\text{draft} \xrightarrow{\text{submit}} \text{submitted}$$

```
(draft,      {event, submit},  ) -> {next_state, submitted}
(draft,      {exit,  submitted}) -> ok

(submitted, {entry, draft}) ->
    {next_state, #submitted{feasibility=deciding, security=deciding}}

(#submitted{feasibility=deciding, security='_'     }, {entry,   draft}) -> ok
(#submitted{feasibility='_',      security=deciding}, {entry,   draft}) -> ok
(#submitted{feasibility=deciding, security=deciding}, {entered, draft}) -> ok
```

The entry calls with '_' allows to pattern match state names for a particular region:

```
handle_state(#submitted{security=deciding}, {entry, _}, Data) ->
    % Setup things needed for the state.
```

Questions?