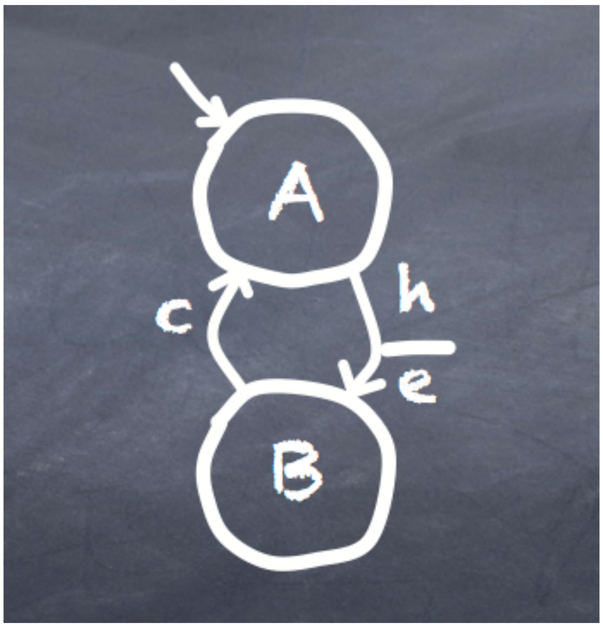
RFSM User Manual - 2.0

J. Sérot



Chapter 1

Introduction

This document is a brief user manual for the RFSM language and compiler.

RFSM is a domain specific language aimed at describing, drawing and simulating *reactive finite state machines*. Reactive FSMs are FSMs for which transitions can only take place at the occurence of *events*.

RFSM has been developed mainly for pedagogical purposes, in order to initiate students into model-based design. It is currently used in courses dedicated to embedded system design both on software and hardware platforms (microcontrolers and FPGA resp.). But RFSM can also be used to generate code (C, SystemC or VHDL) from high-level models, to be integrated to existing applications.

More precisely, RFSM can be used to

- describe FSM-based models and testbenches,
- generate graphical representations of these models (`.dot` format) for visualisation,
- simulate these models, producing `.vcd` files to be displayed with waveform viewers such as `gtkwave`,
- generate C, SystemC and VHDL implementations (including testbenches for simulation)

The RFSM compiler is also used internally by the GRASP application[1] which provides a GUI-based interface to RFSM compiler language.

This document is organized as follows. Chapter 2 is short presentation of the language and its possibilities using simple examples. Chapter 3 is a more throrough presentation describing its syntax in a more systematic way and discussing the main issues related to its semantics. Chapter 4 describes how to use the command-line compiler. Appendices A1, A2 and A3 give some examples of code generated by the C, SystemC and VHDL backends.

Note. The language described in this document is the so-called *standard* RFSM language. The distribution[2] actually contains several variants, all sharing a common *host* language but differing, essentially, in the so-called *guest* language used to describe the conditions and actions attached to transitions. The framework available in the distribution can be used to build one's own variant language. This process is described in the reference manual.

[1] `https://github.com/jserot/grasp`
[2] `https://github.com/jserot/rfsm`

Chapter 2

Overview

This chapter gives an informal introduction to the RFSM language and of how to use it to describe FSM-based systems.

Listing 2.1 is an example of a simple RFSM program[1]. This program is used to describe and simulate the model of a calibrated pulse generator. Given an input clock `H`, with period T_H, it generates a pulse of duration $n \times T_H$ whenever input `E` is set when event H occurs.

Listing 2.1: A simple RFSM program

```
 1  fsm model gensig <n: int> (
 2    in h: event,
 3    in e: bool,
 4    out s: bool)
 5  {
 6    states: E0, E1;
 7    vars: k: int<1:n>;
 8    trans:
 9    | E0 -> E1 on h when e=1 with k:=1, s:=1
10    | E1 -> E1 on h when k<n with k:=k+1
11    | E1 -> E0 on h when k=n with s:=0;
12    itrans:
13    | -> E0 with s:=0;
14  }
15
16  input H : event = periodic (10,0,80)
17  input E : bool = value_changes (0:0, 25:1, 35:0)
18  output S : bool
19
20  fsm g = gensig<3>(H,E,S)
```

The program can be divided in three parts.

The first part (lines 1–14) gives a generic **model** of the generator behavior. The model, named `gensig`, has one parameter, `n`, two inputs, `h` and `e`, of type `event` and `bool` respectively, and one output `s` of type `bool`. Its behavior is specified as a reactive FSM with two states, `E0` and `E1`, and one internal variable `k`. The transitions of this FSM are given after the `trans:` keyword in the form :

[1]This program is provided in the distribution, under directory `examples/std/single/gensig`.

$\boxed{\text{| source\_state −> destination\_state } \textbf{on} \text{ ev } \textbf{when} \text{ guard } \textbf{with} \text{ actions}}$

where

- *ev* is the event trigerring the transition,
- *guard* is a set of (boolean) conditions,
- *actions* is a set of actions performed when the transition is enabled.

The semantics is that the transition is enabled whenever the FSM is in the source state, the event *ev* occurs and all the conditions in the guard are true. The associated actions are then performed and the FSM moves to the destination state. For example, the first transition is enabled whenever an event occurs on input `h` and, at this instant, the value of input `e` is 1. The FSM then goes from state `E0` to state `E1`, sets its internal variable `k` to 1 and its output `s` to 1[2].

The *initial transition* of the FSM is given after the `itrans:` keyword in the form :

$\boxed{\text{| −> initial\_state } \textbf{with} \text{ actions}}$

Here the FSM is initially in state `E0` and the output `s` is set to 0.

Note. In the transitions, the **when** guard and **with** actions are optional and may be omitted.

A graphical representation of the `gensig` model is given in Fig. 2.1 (this representation was automatically generated from the program in Listing 2.1, as explained in Chap. 4).

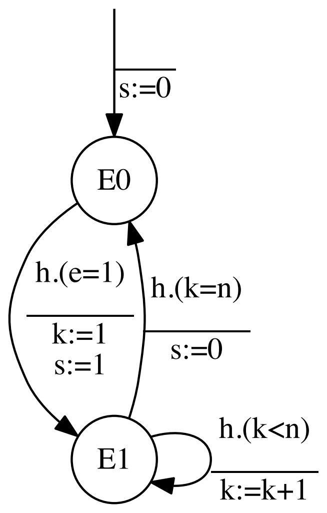


Figure 2.1: A graphical representation of FSM model defined in Listing 2.1

[2]Boolean values `true` and `false` can be denoted 1 and 0 respectively in programs.

Note that, at this level, the value of the parameter `n`, used in the type of the internal variable `k` (line 7) and in the transition conditions (lines 10 and 11) is left unspecified, making the `gensig` model a *generic* one.

The second part of the program (lines 16–18) lists **global inputs and outputs**. For global outputs the declaration simply gives a name and a type. For global inputs, the declaration also specifies the **stimuli** which are attached to the corresponding input for simulating the system. The program of Listing 2.1 uses two kinds of stimuli[3]. The stimuli attached to input `H` are declared as *periodic*, with a period of 10 time units, a start time of 0 and a end time of 80. This means than an event will be produced on this input at time 0, 10, 20, 30, 40, 50, 60, 70 and 80. The stimuli attached to input `E` say that this input will respectively take value 0, 1 and 0 at time 0, 25 and 35 (thus producing a "pulse" of duration 10 time units starting at time 25).

The third and last part of the program (line 20) consists in building the global model of the system by *instanciating* the FSM model(s). Instanciating a model creates a "copy" of this model for which

- the generic parameters (`n` here) are now bound to actual values (3 here),
- the inputs and outputs are connected to the global inputs or outputs.

A graphical representation of the system described in Listing 2.1 is given in Fig. 2.2[4].

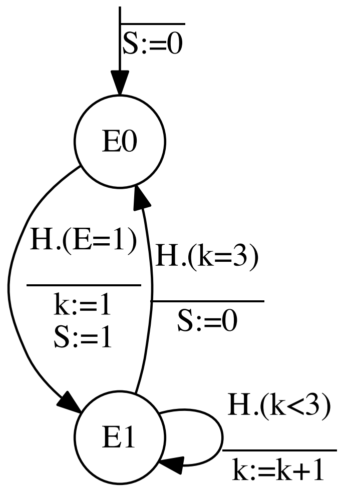


Figure 2.2: A graphical representation of system described in Listing 2.1

Simulating

Simulating the program means computing the reaction of the system to the input stimuli. Simulation can be performed by the RFSM command-line compiler as described in chapter 4. It produces a set of *traces* in VCD (Value Change Dump) format which can visualized using *waveform viewers* such as `gtkwave`. Some simulation results for the program in Listing 2.1 are showed in Fig. 2.3.

[3]See Sec. 3.3 for a complete description of stimuli.

[4]Again, this representation was actually automatically generated from the program in Listing 2.1, as explained in Chap. 4.

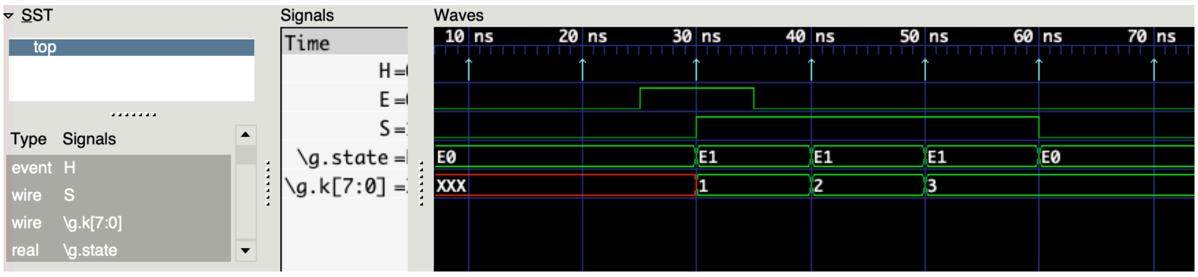


Figure 2.3: Simulation results for the program in Listing 2.1, viewed using `gtkwave`

Code generation

RFSM can also generate code implementing the described systems simulation and/or integration to existing applications.

Currently, three backends are provided :

- a backend generating a C-based implementation of each FSM instance,
- a backend generating a *testbench* implementation in SystemC (FSM instances + stimuli generators),
- a backend generating a *testbench* implementation in VHDL (FSM instances + stimuli generators).

The target language for the C backend is a C-like language augmented with

- a `task` keyword for naming generated behaviors,
- `in`, `out` and `inout` keywords for identifying inputs and outputs,
- a builtin `event` type,
- primitives for handling events : `wait_ev()`, `wait_evs()` and `notify_ev()`.

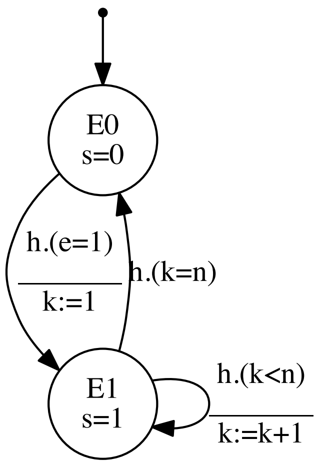
The idea is that the generated code can be turned into an application for a multi-tasking operating system by providing actual implementations of the corresponding constructs and primitives.

For the SystemC and VHDL backends, the generated code can actually be compiled and executed for simulation purpose. The FSM implementations generated by the VHDL backend can also be synthetized to be implemented on hardware using hardware-specific tools[5].

Appendices A1, A2 and A3 respectively give the C and SystemC code generated from the example in Listing 2.1.

Variant formulation

In the automata described in Fig. 2.1 and Listing 2.1, the value of the `s` output is specified by indicating how it changes when transitions are taken (including its initialisation). This is typical of a so-called *Mealy*-style description. In some cases, it is possible – and maybe simpler – to indicate which value this output takes for each state. A equivalent description of that given in Listing 2.1 is obtained, for example, by specifying that `s` is 0 whenever the FSM is in state `E0` and 1 whenever it is in state `E1`.



```
fsm model gensig <n: int> (
  in h: event,
  in e: bool,
  out s: bool)
{
  states: E0 where s=0, E1 where s=1;
  vars: k: int<1:n>;
  trans:
  | E0 -> E1 on h when e=1 with k:=1
  | E1 -> E1 on h when k<n with k:=k+1
  | E1 -> E0 on h when k=n
  itrans:
 | -> E0 with s:=0
}
```

Figure 2.4: A reformulation of the model given in Listing 2.1 and Fig. 2.1 using Moore-style

This style of description, often called *Moore*-style, is illustrated in Fig. 2.4. The value of the `s` output is here attached to states using the `where` clause in the declarations of states.

Note. The `rfsmc` compiler automatically transforms models using Moore-style descriptions models using only Mealy-style ones.

Multi-FSM models

It is of course possible to describe systems composed of several FSM instances.

A first example is given in Listing 2.2 and Fig. 2.5. The system is a simple modulo 8 counter, here described as a combination of three event-synchronized modulo 2 counters[6].

Here a single FSM model (`cntmod2`) is instanciated thrice, as `C0`, `C1` and `C2`. These instances are synchronized using two **shared events**, `R0` and `R1`. Shared events perform *instantaneous synchronisation.* When a FSM *emits* such an event, all transitions triggered by this event are taken, simultaneously with the emitting transition. In the system described in Fig. 2.5, for example, the transition of `C0` (resp. `C1`) from `E1` to `E0` occurs triggers the simultaneous transition of `C1` (resp. `C2)` from `E0` to `E1` and, latter of `C1` (resp. `C2`) from `E1` to `E0`.

[5]We use the QUARTUS toolchain from Intel/Altera.

[6]This program is provided in the distribution, under directory `examples/std/multi/ctrmod8`.

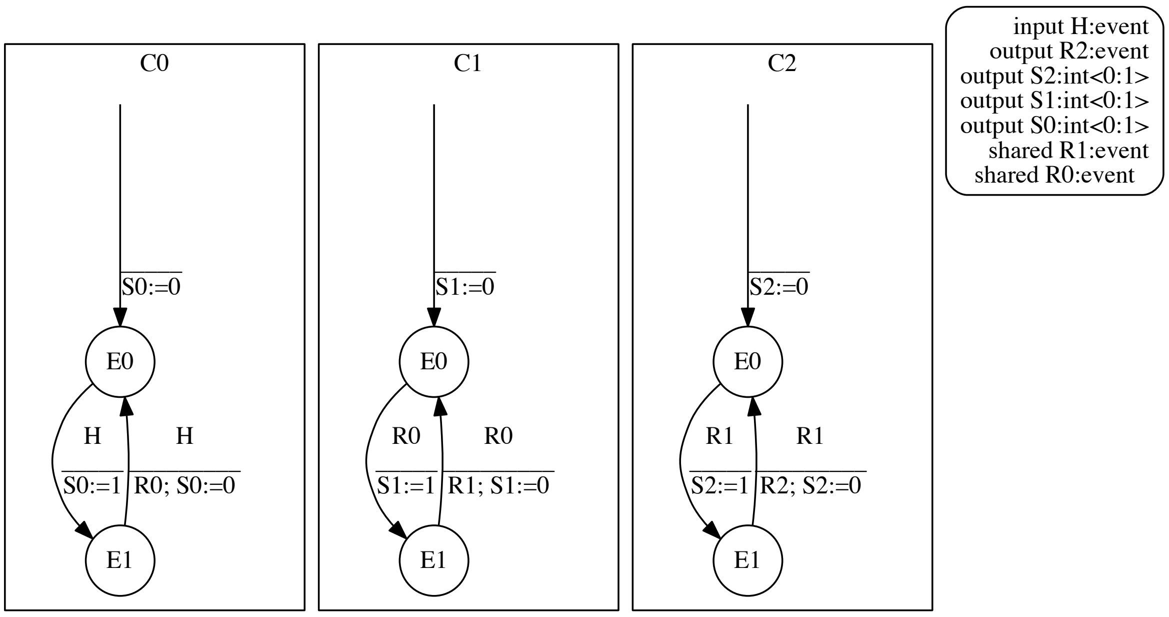


Figure 2.5: Graphical representation of the program of Listing 2.2

Listing 2.2: A program involving three FSM instances synchronized by a shared event

```
fsm model cntmod2(
  in h: event,
  out s: bool,
  out r: event)
{
  states: E0 where s=0, E1 where s=1;
  trans:
  | E0 -> E1 on h
  | E1 -> E0 on h with r;
  itrans:
  | -> E0;
}

input H: event = periodic(10,10,100)
output S0, S1, S2: bool
output R2: event
shared R0, R1: event

fsm C0 = cntmod2(H,S0,R0)
fsm C1 = cntmod2(R0,S1,R1)
fsm C2 = cntmod2(R1,S2,R2)
```

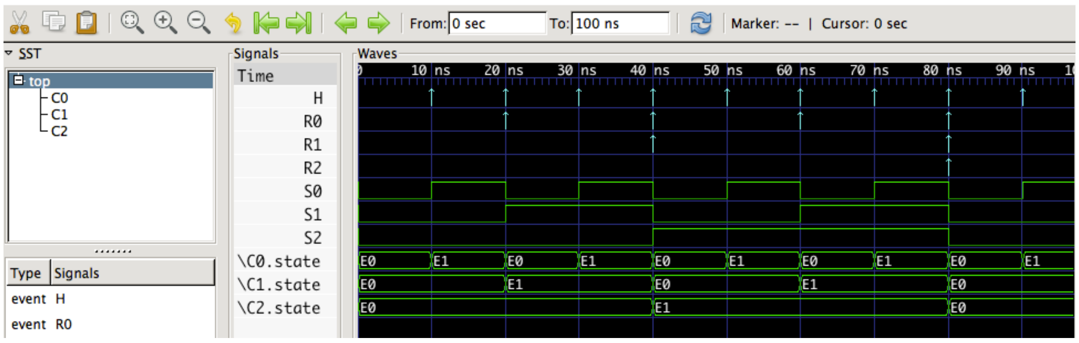


Figure 2.6: Simulation results for the program in Listing 2.5

Simulation results for this program are given in Fig. 2.6.

FSM instances can also interact by means of **shared variables**. This is illustrated in Listing 2.3 and Fig. 2.7[7]. FSM `a1` repeatedly writes the shared variable `c` at each event `h` so that it takes values 1, 2, 3, 4, 1, 2, *etc.* FSM `a2` observes this variable also at each event `h` and simply goes from state `S1` to state `S2` (resp. `S2` to `S1`) when the observed value is 4 (resp. 1).

Listing 2.3: A program involving two FSM instances and a shared variable

```
fsm model A1( in h: event, inout v: int)
{
  states: S1, S2;
  trans:
  | S1 -> S2 on h with v:=1
  | S2 -> S2 on h when v<4 with v:=v+1
  | S2 -> S1 on h when v=4;
  itrans:
  | -> S1 with v:=0;
}

fsm model A2( in h: event, in v: int)
{
  states: S1, S2;
  trans:
  | S1 -> S2 on h when v=4
  | S2 -> S1 on h when v=1;
  itrans:
  | -> S1 ;
}

input h : event = periodic(10,10,100)
shared c : int
fsm a1 = A1(h,c)
fsm a2 = A2(h,c)
```

Simulation results for this program are given in Fig. 2.8.

[7]This program is provided in the distribution, under directory `examples/multi/synv_vp/ex5`.

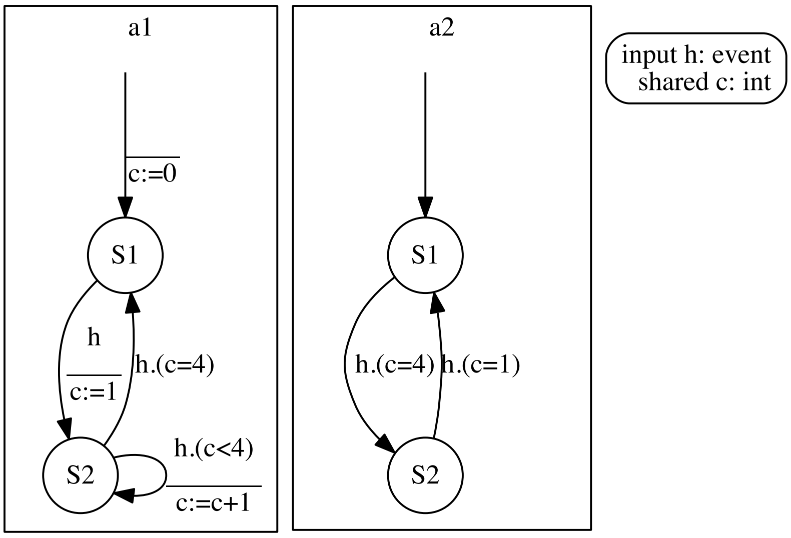


Figure 2.7: Graphical representation of the program of Listing 2.3

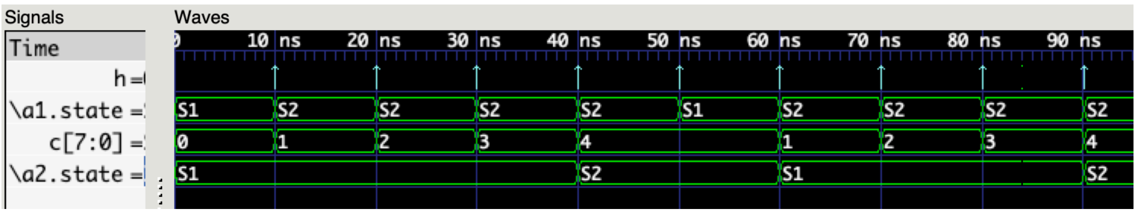


Figure 2.8: Simulation results for the program in Listing 2.7

Chapter 3

The RFSM language

This section describes the syntax and semantics of the *standard* RFSM language in a more systematic, but informal, way. The corresponding formalized descriptions are given in the reference manual.

3.1 Programs

A RSFM program is made of successive sections, containing, respectively

- type declarations,
- constant declarations,
- function declarations,
- FSM model definitions,
- global object definitions,
- FSM instanciations.

Each section is optional[1] but their must appear in the order given above.

3.2 FSM models

An FSM model, introduced by the `fsm model` keywords, describes the interface and behavior of a *reactive finite state machine*. A reactive finite state machine is a finite state machine whose transitions can only be caused by the occurrence of *events*.

fsm model <interface> <body>

The **interface** of the model gives its name, a list of parameters (which can be empty) and a list of inputs and outputs. All parameters and IOs are typed (see Sec. 3.8). Inputs and outputs are explicitely tagged. An IO tagged `inout` acts both as input and output (it can be read and written by the model). Inputs and outputs are listed between `(...)`. Parameters, if present are given between `<...>` and allow the definition of *generic* models. Examples :

[1]Though, obviously, a program with no model definition, and hence no FSM instanciation, is of little interest.

fsm model cntmod8 (**in** h: **event**, **out** s: int<0..7>){ ... }

fsm model gensig<n:int> (**in** h: **event**, **in** e: bit, **out** s: bit) { ... }

fsm model update (**in** top: **event**, **inout** lock: **bool**){ ... }

The model **body**, written between `{...}`, generally comprises four sections :

- a section giving the list of *states*,
- a section introducing local (internal) *variables*,
- a section giving the list of *transition*,
- a section specifying the *initial transition.*

Each section starts with the corresponding keyword (`states:`, `vars:`, `trans:` and `itrans:` resp.) and ends with a semi-colon.

fsm model ... (...) { **states**: ...; **vars**: ...; **trans**: ...; **itrans**: ...; }

3.2.1 States

The `states:` section gives the set of internal states, as a comma-separated list of identifiers (each starting with a uppercase letter). Example :

states: Idle , Wait1, Wait2, Done;

Values for outputs can be attached to states using the `where` keyword. When several assignements are attached to the same state, they are separated using the `and` keyword.

states: Idle , Wait1 **where** s1=0, Wait2 **where** s1=1 **and** s2=0, Done;

3.2.2 Variables

The `vars:` section gives the set of internal variables, each with its type. Example :

vars: cnt: int , stop: **bool**;

The type of a variable may depend on parameters listed in the model interface. Example

fsm gensig<n: int> (...) { ... **vars**: k: int<0:n>; ... }

The `vars:` section may be omitted.

3.2.3 Transitions

The `trans:` section gives the set of transitions between states. Each transition is denoted

```
| src_state -> dst_state on ev when guards with actions
```

where

- *src_state* and *dst_state* respectively designates the source state and destination state,
- *ev* is the event trigerring the transition,
- *guards* is a set a enabling conditions,
- *actions* is a set of actions performed when then transition is enabled.

The semantics is that the transition is enabled whenever the FSM is in the source state, the triggering event occurs and all conditions evaluate to true. The associated actions are then performed and the FSM moves to the destination state.

The triggering event must be listed in the inputs.

Each condition listed in *guards* must evaluate to a boolean value. The guard is true if *all* conditions evaluate to true (conjonctive semantics). The guards may involve inputs and/or internal variables.

The guard can be empty. In this case, the transition is denoted

```
| src_state -> dst_state on ev with actions
```

The **actions** associated to a transition consists in modifications of the outputs and/or internal variables or emissions of events. Modifications of outputs and internal variables are denoted

```
id := expr
```

where *id* is the name of the output (resp. variable) and *expr* an expression involving inputs, outputs and variables and operations allowed on the corresponding types.

The action of emitting of an event is simply denoted by the name of this event.

Examples :

```
S0 -> S1 on top
```

In the above example, the enclosing FSM switches from state `S0` to state `S1` when the event `top` occurs.

```
Idle -> Wait on Clic with ctr:=0, received
```

In the above example, the enclosing FSM switches from state `Idle` to state `Wait`, resetting the internal variable `ctr` to 0 and emitting the event `received` whenever an event occurs on its `Clic` input.

```
Wait -> Wait on Top when ctr<8 with ctr:=ctr+1
```

In the above example, the enclosing FSM stays in state `Wait` but increments the internal variable `ctr` whenever an event `Top` occurs and that, *at this instant*, the value of variable `ctr` is smaller than 8.

Expressions may also involve the C-like ternary conditional operator `?:`. For example, in the example below, the enclosing FSM stays in state `S0` but updates the variable `k` at each occurrence of event `H` so that is incremented if its current value is less than 8 or reset to 0 otherwise.

S0 –> S0 **on** H **with** k:=k<8?k+1:0

The set of actions may be empty. In this case, the transition is denoted :

src_state –> dst_state **on** ev **when** guard

Semantic issues

Sequential vs. synchronous actions. By default, actions are performed *sequentially*, i.e. one after the other. For example, if `x` and `y` are internal variables of the enclosing FSM and respectively have values 1 and 0, then taking this transition

S0 –> S1 **on** H **with** x:=x+1, y:=x∗2

will assign them values 2 and 4 respectively, because the action `x:=x+1` is performed before the action `y:=x*2`.
This interpretation is the most intuitive one and naturally fits with software-based implementations.

Non-determinism and priorities. The FSM models involved in programs should normally be *deterministic*. In other words, a situation where several transitions are enabled at the same instant should normally never arise. But this condition may actually be difficult to enforce, especially for models reacting to several input events. Consider for example, the model described in Listing 3.1. This model describes a (simplified) stopwatch. It starts counting seconds (materialized by event `sec`) as soon as event `startstop` occurs and stops as soon as it occurs again.
The problem is that if both events occur simultaneously then both the transitions at line 10 and 11 are enabled. In fact, here's the error message produced by the compiler when trying to simulate the above program :

```
Error when simulating FSM c: non deterministic transitions found at t=70:
- Running -- H / ctr:=ctr+1; aff:=ctr -> Running
- Running -- StartStop -> Stopped
```

Of course, this could be avoided by modifying the stimuli attached to input `StartStop` so that the `StartStop` and `H` events are never emitted at the same time. But this is, in a sense, cheating, since the `StartStop` event is supposed to modelize user interaction which occur, by essence, at impredictible dates.
The above problem can be solved by assigning *priorities* to transitions. In the current implementation, this is achieved by tagging some transitions as "high priority" transitions[2]. When several transitions are enabled, if one is tagged as "high priority" than it is automatically selected[3].
Syntaxically, tagging a transition is simply achieved by replacing the leading "`|`" by a "`!`". In the case of the example above, the modified program is given in Listing 3.2. Tagging the last transition is here equivalent to give to the `startstop` precedence against the `h` event when the model is in state `Running`.

[2]Future versions may evolve towards a more sophisticated mechanism allowing numeric priorities.
[3]If none (resp. several) is (resp. are) tagged, the conflict remains, of course.

Listing 3.1: A program showing a potentially non-deterministic model

```
fsm model chrono (
      in sec: event,
      in startstop: event,
     out aff: int)
   {
   states: Stopped, Running;
   vars: ctr: int;
   trans:
   | Stopped -> Running on startstop with ctr:=0; aff:=0
   | Running -> Running on sec with ctr:=ctr+1; aff:=ctr
   | Running -> Stopped on startstop;
   itrans:
   |-> Stopped;
   }

input StartStop: event = sporadic(25,70)
input H:event = periodic(10,10,110)
output Aff: int

fsm c = chrono(H,StartStop,Aff)
```

Listing 3.2: A rewriting of the model defined in Listing 3.1

```
fsm model chrono (...)
   {
   ...
   trans:
      ...
      | Running -> Running on sec with ctr:=ctr+1; aff:=ctr
      ! Running -> Stopped on startstop  -- This transition takes priority
          on the others
   itrans: -> Stopped;
   }
...
```

3.2.4 Initial transition

The `itrans:` section specifies the initial transition of the FSM. This transition is denoted :

$$\boxed{\text{| —> init\_state } \textbf{with} \text{ actions}}$$

where *init_state* is the initial state and *actions* a list of actions to be performed when initializing the FSM. The latter can be empty. in this case the initial transition is simply denoted :

$$\boxed{\text{| —> init\_state}}$$

3.2.5 Output values

Output values can be set by either attaching them to states or by updating them on transitions. For a given output `o`, attaching a value `v` to a state `S`, by writing

$$\boxed{\textbf{states}\text{: S } \textbf{where} \text{ o=v, ...}}$$

is equivalent to adding the action

$$\boxed{\text{o:=v}}$$

to each transition ending at state `S`.

The compiler rejects models for which the value of an output is specified both with the former and latter formulation. Stricly speaking, models for which the values specified by each formulation are equivalent could be accepted, but this condition is statically undecidable in general (because values assigned to outputs in transitions may depend of inputs).

3.3 Inputs and outputs

Interface to the external world are represented by `input` and `output` objects.

▶ For outputs the declaration simply gives a name and a type :

$$\boxed{\textbf{output} \text{ name : typ}}$$

▶ For inputs, the declaration also specifies the **stimuli** which are attached to the corresponding input for simulating the system.

$$\boxed{\textbf{input} \text{ name : typ = stimuli}}$$

There are three types of stimuli : periodic and sporadic stimuli for inputs of type `event` and value changes for scalar inputs.

Periodic stimuli are specified with a period, a starting time and an ending time.

$$\boxed{\textbf{periodic}\text{(period,t0,t1)}}$$

Sporadic stimuli are simply a list of dates at which the corresponding input event occurs.

$$\boxed{\textbf{sporadic}\text{(t1 ,..., tn)}}$$

Value changes are given as list of pairs `t:v`, where `t` is a date and `v` the value assigned to the corresponding input at this date.

value_changes(t1:v1,...,tn:vn)

Examples:

input Clk: **event** = **periodic**(10,10,120)

The previous declaration declares `Clk` as a global input producing periodic events with period 10, starting at t=10 and ending at t=100[4].

input Clic: **event** = **sporadic**(25,75,95)

The previous declaration declares `Clic` as a global input producing events at t=25, t=75 and t=95.

input E : **bool** = **value_changes** (0:false, 25:true, 35:false)

The previous declaration declares `E` as a global boolean input taking value `false` at t=0, `true` at t=25 and `false` again at t=35.

3.4 Shared objects

Shared objects are used to represent interconnexions between FSM instances. This situation only occurs when the system model involves several FSM instances and when the input of a given instance is provided by the output of another one.

► For shared objects the declaration simply gives a name and a type :

shared name : typ

Examples:

shared done: **event**
shared ctr: int

The previous declarations declare `done` as a shared event and `ctr` as a shared variable of type `int`.

Semantic issues

Shared objects are typically used to perform some kind synchronisation between FSMs. The precise semantic of this synchronisation depends on the shared object. We here describe it informally. A formal account is given the reference manual.

```
fsm model A1(
  in h: event,
  out e: event)
{
  states: A, B;
  trans:
  | A -> B on h with e
  | B -> A on h;
  itrans:
  | -> A ;
}

fsm model A2(
  in h: event,
  in e: event)
{
  states: A, B;
  trans:
  | A -> B on e
  | B -> A on h;
  itrans:
  | -> A ;
}
```

```
input H : event = sporadic(10,20,30,40)
shared e : event

fsm a1 = A1(H,e)
fsm a2 = A2(H,e)
```

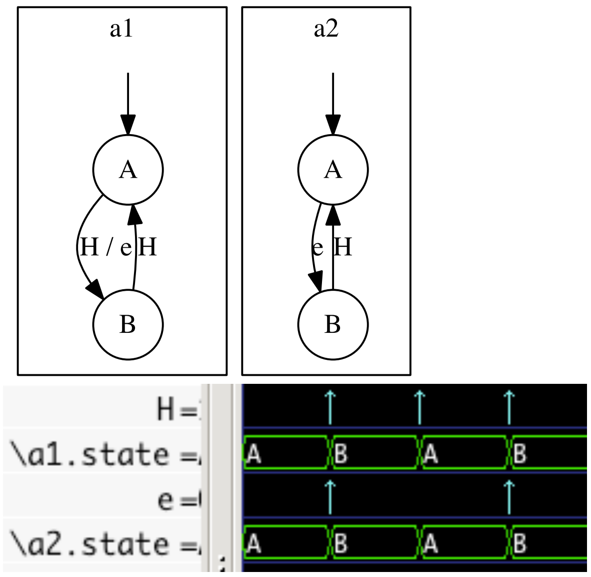


Figure 3.1: Illustration of instantaneous synchronisation

Synchronisation using a shared event

Synchronisation using a shared event is both instantaneous and ephemeral.

Instantaneous means that an event emitted by a FSM when taking a transition can trigger a reaction of another FSM at the same logical instant, the two reactions – that of the "emitting" FSM and that of the "receiving" FSM – being simultaneous. This is illustrated in Fig. 3.1. Here, each occurence of event `H` when `a1` is in state `A` triggers the simultaneous transition of `a2` from state `A` to state `B`.

Ephemeral synchronisation means that if an event emitted by a FSM when taking a transition is not awaited by another FSM it is simply "lost". In other words, events are never memorised. This is illustrated in Fig. 3.2. In this example, the first occurence of event `e` is lost because FSM `a2` is not waiting for it when it is emitted by FSM `a1` at the first occurrence of event `H`. As a result, the transition of `a2` from state `B` to state `C` only occurs at second occurrence of *e*, when `a2` is in state `B`.

Note. The semantics of event synchronisation described here is somehow related to that of *rendez-vous* supported by certain programming languages. But it is definitely not equivalent. The latter enforces that *both* transitions, the emitting and the receiving one, are taken together. This means in particular that if there's no transition waiting for the emitted event, the emitting transition will *not* be taken (in other words, the source FSM will block). This is not the case here. In this situation, and as explained above, the emitted event will be simply ignored ("lost"). Emitting an event can never prevent a transition to be taken in our semantics.

[4]Note that, at this level, there's no need for an absolute unit for time.

```
fsm model A1(
  in h: event,
  out e: event)
{
  states: A, B, C;
  trans:
  | A -> B on h with e
  | B -> C on h
  | C -> A on h;
  itrans:
  | -> A ;
}

fsm model A2(
  in h: event,
  in e: event)
{
  states: A, B, C;
  trans:
  | A -> B on h
  | B -> C on e
  | C -> A on h;
  itrans:
  | -> A ;
}
```

```
input H : event = sporadic(10, 20, 30, 40, 50)
shared e : event

fsm a1 = A1(H,e)
fsm a2 = A2(H,e)
```

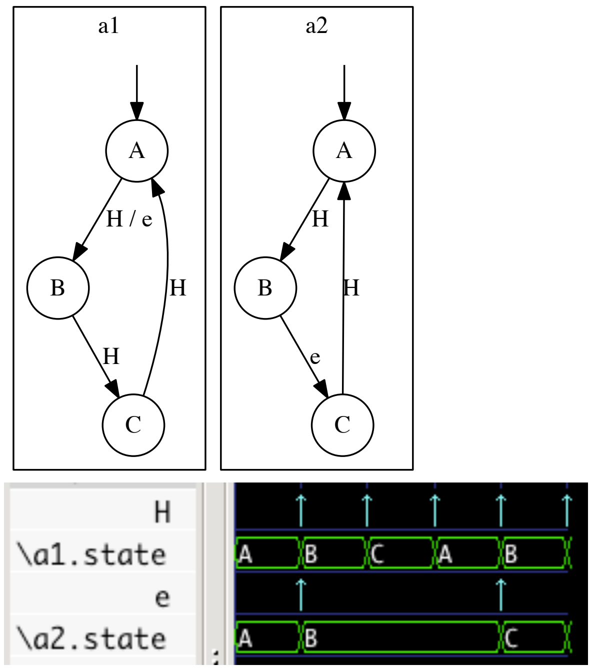


Figure 3.2: Illustration of ephemeral synchronisation

Implementation issues. The semantics of events presented here is that implemented by the `rfsmc` simulator. For various reasons, it may not fully supported all compiler backends. The support of shared events in the SystemC backend, for example, is fragile and has not yet fully tested[5]. It is completely lacking in the VHDL backend (VHDL implementation of FSMs can only be triggered by a single, external, `clock` signal). Finally, the *event* mechanism provided by most of real-time operating systems (as abstracted by `notify_ev()` and `wait_ev()` pseudo-primitives used in the code generated by the C backend) may not obey the semantics provided here (most of OS-supported events are *memorized* when they are not awaited for when emitted in particular). The concept of event provided by the RSFM language must therefore be viewed as a basic and abstract *modeling* tool, to be refined afterwards at the implementation level.

Synchronisation using shared variables

The semantics associated to shared variables is that of *instantaneous broadcast*. This means that any modification of the value of a shared variable by a FSM is immediately viewed by the other FSMs. More precisely, if a reaction of a FSM modifies the value of a shared value, the new value can enable, *in the same global reaction*, the transition of another FSM. This is illustrated in Fig. 3.3. Here, both `a1` and `a2` react to event `H`. When `a1` and `a2` are in state `S1` this is possible only because the modification

[5]It relies on the insertion of zero-time `wait` instructions.

of the shared variable `v` made by `a1` is immediately visible and hence can enable the transition of `a2`.

```
fsm model A1(
  in h: event,
  out v: bool)
{
  states: S1, S2;
  trans:
  | S1 -> S2 on h with v:=1
  | S2 -> S1 on h with v:=0;
  itrans:
  | -> S1 with v:=0;
}
fsm model A2(
  in h: event,
  in v: bool)
{
  states: S1, S2;
  trans:
  | S1 -> S2 on h when v=1
  | S2 -> S1 on h;
  itrans:
  | -> S1 ;
}
```

```
input H: event = sporadic(10,20,30,40)
shared V: bool

fsm a1 = A1(H,V)
fsm a2 = A2(H,V)
```

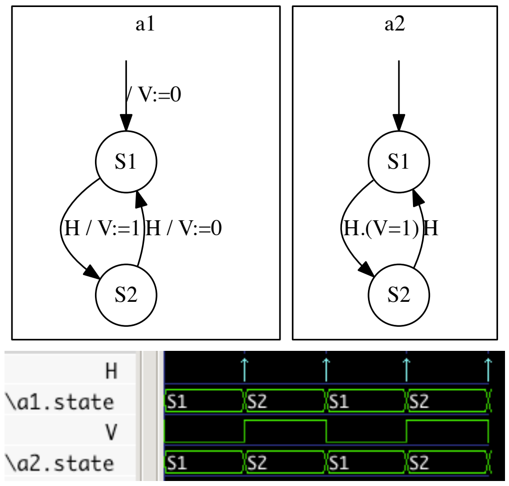


Figure 3.3: Illustration of instantaneous broadcast of shared variables

Because shared variables are, by definition, memorized, they can used to implement *defered synchronisation*, *i.e.* the situation where a FSM emits an event which will be used *later* by another FSM (shared events cannot be used in this case since, as described above, non-awaited events are not memorized and hence lost). This is illustrated in Fig. 3.4. Here, `a1` set variable `v` to 1 when going from state `S1` to `S2` but `a2` only detects this when going from state `S2` to `S3`, reseting the variable to 0 *en passant*. In effect, `a1` has emitted a event which has been memorized and caught latter by `a2`.

Implementation issues. The semantics of instantaneous broadcast for variables is the most intuitive one at the modelisation level is the default one for simulation. However, and as for events, this semantics is not supported by all compiler backends.

For the SystemC backend, support of instantaneous broadcast is supported by means of automatic insertion of zero-time delta-cycles and is therefore fragile.

It is *not* supported by the VHDL backend because shared variables are (currently) implemented as *shared signals*, for which any modification at a given cycle is only visible at the next clock cycle.

Shared variables are implemented as global variables by the C backend. When the corresponding code is used to define concurrent tasks for a real-time operating system, the instantaneous broadcast hypothesis cannot in general be assumed (because the delay separating writes and reads of such a variable depends of the scheduler and cannot be predicted).

3.5 FSM instances

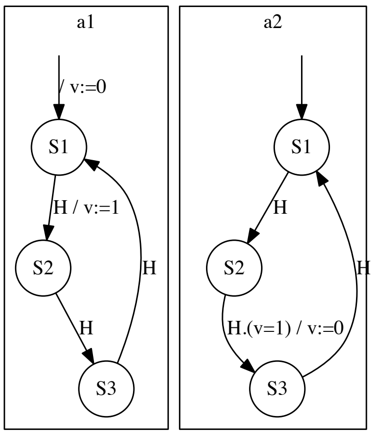
The description of the system is carried out by instanciating previously defined FSM models.

```
fsm model A1(
  in h: event,
  out v: bool)
{
  states: S1, S2, S3;
  trans:
  | S1 -> S2 on h with v:=1
  | S2 -> S3 on h
  | S3 -> S1 on h;
  itrans:
  | -> S1 with v:=0;
}

fsm model A2(
  in h: event,
  inout v: bool)
{
  states: S1, S2, S3;
  trans:
  | S1 -> S2 on h
  | S2 -> S3 on h when v=1
                  with v:=0
  | S3 -> S1 on h;
  itrans:
  | -> S1 ;
}
```

```
input H: event = sporadic(10,20,30)
shared v: bool

fsm a1 = A1(H,v)
fsm a2 = A2(H,v)
```



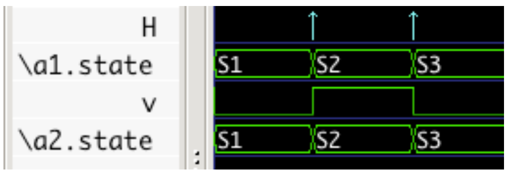


Figure 3.4: Using a shared variable to implement memorized events

Instanciating a model creates a copy of the corresponding FSM for which

- the parameters of the model are bound to their actual value,
- the declared inputs and outputs are connected to global inputs, outputs or shared objects.

The syntax for declaring a model instance is as follows :

fsm inst_name = model_name<param_values>(actual_ios)

where

- *inst_name* is the name of the created instance,
- *model_name* is the name of the instanciated model,
- *param_values* is a comma-separated list of values to be assigned to the formal (generic) parameters,

- *actual_ios* is a comma-separated list of global inputs, outputs or shared objects to be connected to the instanciated model.

Binding of parameter values and IOs is done by position. Of course the number and respective types of the formal and actual parameters (resp. IOs) must match.

For example, the last line of the program given in Listing 2.1

fsm g = gensig<3>(H,E,S)

creates an instance of model `gensig` for which `n=3` and whose inputs (resp. output) are connected to the global inputs (resp. output) `H` and `E` (resp. `S`).

In the current version, paramater values are limited to scalar values (`int`s, `bool`s, `char`s and `float`s).

3.6 Constants

Global constants can be defined using the following syntax :

constant name : <**type**> = <value>

where

- *type* is the type of the defined constant (currently limited to `int`, `bool`, `char`, `float` and arrays of such types,
- *value* is the value of the constant.

Global constants have a global scope and hence can be used in any FSM model or instance.

3.7 Functions

Conditions and actions associated to FSM transitions can use globally defined functions. An example is given in listing 3.3[6]. The FSM described here computes an approximation of the square root of its input `u` using Heron's classical algorithm. Successive approximations are computed in state `Iter` and the end of computation is detected when the square of the current approximation `x` differs from the argument (`a`) from less than a given threshold `eps`. For this, the model uses the global function `f_abs` defined at the beginning of the program. This function computes the absolute value of its argument and is used twice in the definition of the FSM model `heron`, for defining the condition associated to the two transitions going out of state `Iter`.

▶ The general form for a function definition is

function name (<arg_1>:<type_1>, ..., <arg_n>:<type_n>): <type_r> { **return** <expr> }

where

- <arg_i> (resp. <type_i>) is the name (resp. type) of the ith argument,
- <type_r> is the type of value returned by the function,
- <expr> is the expression defining the function value.

▶ Functions can only return one result and cannot use local variables. There are therefore more like *macros* in the C language and are typically used to improve readability of the programs.

[6]This example can be found in directory `examples/std/single/heron` in the distribution.

Listing 3.3: An RFSM program using a global function definition

```
function f_abs(x: float) : float { return x < 0.0 ? -.x : x }

fsm model heron<eps: float>(
  in h: event,
  in start: bool,
  in u: float,
  out rdy: bool,
  out niter: int,
  out r: float)
{
  states: Idle where rdy=1, Iter where rdy=0;
  vars: a: float, x: float, n: int;
  trans:
  | Idle -> Iter on h when start=1 with a:=u, x:=u, n:=0
  | Iter -> Iter on h when f_abs((x*.x)-.a)>=eps with x:=(x+.a/.x)/.2., n
     :=n+1
  | Iter -> Idle on h when f_abs((x*.x)-.a)<eps with r:=x, niter:=n;
  itrans:
  | -> Idle;
}

input H : event = periodic (10,10,200)
input U : float = value_changes (5:2.0)
input Start : bool = value_changes (0:0, 25:1, 35:0)
output Rdy1, Rdy2 : bool
output R1, R2 : float
output Niter : int

fsm h = heron<0.00000001> (H,Start,U,Rdy2,Niter,R2)
```

3.8 Types and type declarations

Types present in RFSM programs belong to two categories : builtin types and user defined types.

Builtin types are : `bool`, `int`, `float`, `char`, `event` and `array`s.

▶ Objects of type `bool` can have only two values : `0` (false) and `1` (true).

▶ Values of type `char` are denoted using single quotes. For example, for a variable `c` having type `char` :

```
c := 'A'
```

They can be converted from/to they internal representation as integers using the "`::`" *cast* operator. For example, if `c` has type `char` and `n` type `int`, then

```
n := 'A'::int ; c:=(n+1)::char
```

assigns value 65 to `n` (ASCII code) and, then, value `'B'` to `c`.

▶ The type `int` can be refined using a *size* or a *range annotation.* The type `int<sz>`, where `sz` is an integer, is the type of integers which can be encoded using `n` bits. The type `int<min:max>`, where both `min` and `max` are integers, is the type of integers whose value ranges from `min` to `max`. The size and range limits, can be given as litteral constants (ex: `8`) or as parameter values[7], as for the type of the variable `k` in Listing 2.1.

▶ Supported operations on values of type `int` are described in Table 3.1. If `n` is an integer and `hi` (resp. `lo`) an integer expression then `n[hi:lo]` designates the value represented by the bits `hi`...`lo` in the binary representation of `n`. Bit ranges can be both read (ex: `x=y[6:2]`) or written (ex: `x[8:4]:=0`). The syntax `n[i]`, where `n` is an integer is equivalent to `n[i:i]`. The *cast* operator (`::`) can be used to combine integers with different sizes (for example, if `n` has type `int<16>` and `m` has type `int<8>`, writing `n:=n+m` is not allowed and must be written, instead, `n:=n+m::int<16>`. Note that the logical "or" operator is denoted "`||`" because the single "`|`" is already used in the syntax.

`+`, `-`, `*`, `/`, `%` (modulo)	arithmetic operations
`>>`, `<<`	(logical) shift right and left
`&`, `||`, `^`	bitwise and, or and xor
`[.:.]`	bit range extraction (ex: `n:=m[5:3]`)
`[.]`	single bit extraction (ex: `b:=m[4]`)
`::`	resize (ex: `n::int<8>`)

Table 3.1: Builtin operations on integers

▶ The operations on values of type `float` are : "`+.`", "`-.`", "`*.`" and "`/.`" (the dot suffix is required to distinguish them from the corresponding operations on `int`s).

▶ Arrays are 1D, fixed-size collections of `int`s, `bool`s, `char`s or `float`s. Indices range from 0 to `n-1` where `n` is the size of the array. For example, `int array[4]` is the type describing arrays of four integers. If `t` is an object with an array type, its cell with index i is denoted `t[i]`.

User defined types are either *type abbreviations*, *enumerations* or *records.*

▶ Type abbreviations are introduced with the following declaration

[7]Provided, of course, that the corresponding parameter as type `int`.

$\boxed{\textbf{type}\ \text{typename} = \text{type\_expression}}$

Each occurrence of the defined type in the program is actually substituted by the corresponding type expression.

► Enumerated types are introduced with the following declaration

$\boxed{\textbf{type}\ \text{typename} = \textbf{enum}\ \{\ \text{C1, ..., Cn}\ \}}$

where `C1`, ..., `Cn` are the enumerated values, each being denoted by an identifier starting with an uppercase letter. For example :

```
type color = { Red, Green, Orange }
```

► Record types are introduced with the following declaration

$\boxed{\textbf{type}\ \text{typename} = \textbf{record}\ \{\ \text{fid1: ty1, ..., fidn: tyn}\ \}}$

where `fid1`, ..., `fidn` and `ty1`, ..., `tyn` are respectively the name and type of each record field For example :

```
type coord = record { x: int, y: int}
```

Individual fields of a value with a record type can be accessed using the classical “dot” notation. For example, with a variable `c` having type `record` as defined above :

```
c.x := c.x+1
```

Chapter 4

Using the RFSM compiler

The RFSM compiler can be used to

- produce graphical representations of FSM models and programs (using the `.dot` format),
- simulate programs, generating execution traces (`.vcd` format),
- generate C, SystemC or VHDL code from FSM models and programs.

This chapter describes how to invoke compiler on the command-line. On Unix systems, this is done from a terminal running a shell interpreter. On Windows, from an MSYS or Cygwin terminal.

The compiler is invoked with a command like :

```
rfsmc [options] source_files
```

There must be at least one source file. If several are given, all happens as if a single one, obtained by concatening all of them, in the given order, was used.

The complete set of options is described in the reference manual.

The set of generated files depends on the selected target. The output file `rfsm.output` contains the list of the generated file.

4.1 Generating graphical representations

```
rfsmc [-options] -dot source_files
```

The previous command generates a graphical representation of each FSM model contained in the given source file(s). If the source file(s) contain(s) FSM instances, involving global IOs and shared objects, it also generates a graphical representation of the the corresponding system.

The graphical representations use the `.dot` format and can be viewed with the `Graphviz` suite of tools[1].

The representation for the FSM model `m` is generated in file `m.dot`. When generated, the representation for the system is written in file `main.dot` by default. The name of this file can be changed with the `-main` option.

By default, the generated `.dot` files are written in the current directory. This can be changed with the `-target_dir` option.

[1]Available freely from `http://www.graphviz.org`.

4.2 Running the simulator

```
rfsmc [-options] -sim source_files
```

The previous command runs simulator on the program described in the given source files, writing an execution trace in VCD (Value Change Dump) format.

The generated `.vcd` file can be viewed using a VCD visualizing application such as `gtkwave`[2].

By default, the VCD file is named `main.vcd`. This name can be changed using the `-main` option.

By default, the VCD file is written in the current directory. This can be changed with the `-target_dir` option.

4.3 Generating C code

```
rfsmc [-options] -ctask source_files
```

For each FSM model `m` contained in the listed source file(s), the previous command generates a file `m.c` containing a C-based implementation of the corresponding behavior.

By default, the generated code is written in the current directory. This can be changed with the `-target_dir` option.

4.4 Generating SystemC code

```
rfsmc [-options] -systemc source_files
```

If the source file(s) only contain(s) FSM *models*, then, for each listed FSM model `m`, the previous command generates a pair of files `m.h` and `m.cpp` containing the interface and implementation of the SystemC module implementing this model.

If the source file(s) contain(s) FSM *instances*, involving global IOs and shared objects, it generates

- for each FSM instance `m`, a pair of files `m.h` and `m.cpp` containing the interface and implementation of the SystemC module implementing this instance,
- for each global input `i`, a pair of files `inp_i.h` and `inp_i.cpp` containing the interface and implementation of the SystemC module describing this input (generating the associated stimuli, in particular),
- a file `main.cpp` containing the description of the *testbench* for simulating the program.

The name of the file containing the *testbench* can be changed with the `main` option.

By default, the generated code is written in the current directory. This can be changed with the `-target_dir` option.

Simulation itself is performed by compiling the generated code and running the executable, using the standard SystemC toolchain. In order to simplify this, the RFSM compiler also generates a customized *Makefile* so that compiling and running the code generated by the SystemC backend can be performed by simply invoking `make`. For this, the compiler simply needs to know where to find the predefined template from which this *Makefile* is built. This is achieved by using the `-lib` option when invoking the compiler. For example, provided that RFSM has been installed in directory `/usr/local/rfsm`, the following command

[2]gtkwave.sourceforge.net

```
rsfmc -systemc -lib /usr/local/rfsm/lib -target_dir ./systemc source_file(s)
```

will write in directory `./systemc` the generated source files and the corresponding `Makefile`. Compiling these files and running the resulting application is then simply achieved by typing

```
cd ./systemc
make
```

Note. The generated *Makefile* uses platform-specific definitions which have been written in a file named `platform` located in RSFM library directory (`/usr/local/rfsm/lib/etc/plaform` in the example above). This file is generated by the installation process from the values given to the `configure` script. Depending on your local SystemC installation, some definitions given in the `platform` file may have to be adusted.

4.5 Generating VHDL code

```
rfsmc [-options] -vhdl source_files
```

If the source file(s) only contain(s) FSM *models*, then, for each listed FSM model `m`, the previous command generates file `m.vhd` containing the entity and architecture describing this model.

If the source file(s) contain(s) FSM *instances*, involving global IOs and shared objects, it generates

- for each FSM instance `m`, a file `m.vhd` containing an entity and architecture description for this instance,
- a file `main_top.vhd` containing the description of the *top level* model of the system,
- a file `main_tb.vhd`containing the description of the *testbench* for simulating the system.

The name of the files containing the *top level* description *testbench* can be changed with the `main` option.

By default, the generated code is written in the current directory. This can be changed with the `-target_dir` option.

The produced files can then compiled, simulated and synthetized using a standard VHDL toolchain[3].

As for the SystemC backend, the RFSM compiler simplifies the compilation and simulation of the generated code by also generating a dedicated *Makefile*. For example, and, again, provided that RFSM has been installed in directory `/usr/local/rfsm`, the following command

```
rsfmc -vhdl -lib /usr/local/rfsm/lib -target_dir ./vhdl source_file(s)
```

will write in directory `./vhdl` the generated source files and the corresponding `Makefile`. Compiling these files and running the resulting application is then simply achieved by typing

```
cd ./vhdl
make
```

[3]We use GHDL for simulation and Altera/Quartus for synthesis.

4.6 Using `rfsmmake`

The current distribution provides a script named `rfsmmake` aiming at easing the use of the RSFM compiler in a command line environment. With this tool, the only thing required is to write a small *project description* (`.pro` file). Invoking `rfsmmake` will then automatically build a top-level *Makefile* which can be used to invoke the compiler, generate code and exploit the generated products.

Suppose, for instance, that the application is made of two source files, `foo.fsm`, containing the FSM model(s), and `main.fsm`, containing the global declarations and FSM instanciations (the so-called *testbench*). Writing the following lines in file `main.pro`

```
SRCS=foo.fsm main.fsm
GEN_OPTS= ...
DOT_OPTS= ...
SIM_OPTS= ...
SYSTEMC_OPTS= ...
VHDL_OPTS= ...
```

and invoking

```
rfsmmake main.pro
```

will generate a file `Makefile` in the current directory. Then, simply typing[4]

- `make dot` will generate the `.dot` and lauch the corresponding viewer,
- `make sim.run` to run the simulation using the interpreter (`make sim.show` to display results),
- `make ctask.code` will invoke the C backend C and generate the corresponding code,
- `make systemc.code` will invoke the SystemC backend and generate the corresponding code,
- `make systemc.run` will invoke the SystemC backend, generate the corresponding code, compile it and run the corresponding simulation,
- `make vhdl.code` will invoke the VHDL backend and generate the corresponding code,
- `make vhdl.run` will invoke the VHDL backend, generate the corresponding code, compile it and run the corresponding simulation,
- `make sim.show` (resp `make systemc.show` and `make vhdl.show`) will display the simulation traces generated by the interpreter (resp. SystemC and VHDL simulation).

[4]Please refer to the generated *Makefile* for a complete list of targets.

Appendix A1 - Example of generated C code

This is the code generated from program given in Listing 2.1 by the C backend.

```
task Gensig<int n>(
  in event h;
  in bool e;
 out bool s;
  )
{
  int<1:n> k;
  enum { E0,E1 } state = E0;
  s = false;
  while ( 1 ) {
    switch ( state ) {
    case E0:
      wait_ev(h);
      if ( e==true ) {
        k = 1;
        s = true;
        state = E1;
        }
      break;
    case E1:
      wait_ev(h);
      if ( k==n ) {
        s = false;
        state = E0;
        }
      else if ( k<n ) {
        k = k+1;
        }
      break;
    }
  }
};
```

Appendix A2 - Example of generated SystemC code

This is the code generated from program given in Listing 2.1 by the SystemC backend.

Listing 4.1: File g.h

```
#include "systemc.h"

SC_MODULE(G)
{
  // Types
  typedef enum { E0,E1 } t_state;
  // IOs
  sc_in<bool> H;
  sc_in<bool> E;
  sc_out<bool> S;
  // Local variables
  t_state state;
  int k;

  void react();

  SC_CTOR(G) {
    SC_THREAD(react);
    }
};
```

Listing 4.2: File g.cpp

```
#include "g.h"
#include "rfsm.h"

void G::react()
{
  state = E0;
  S.write(false);
  while ( 1 ) {
    switch ( state ) {
    case E0:
      wait(H.posedge_event());
      if ( E.read()==true ) {
        k = 1;
```

```
          S.write(true);
          state = E1;
          }
        wait(SC_ZERO_TIME);
        break;
      case E1:
        wait(H.posedge_event());
        if ( k==3 ) {
          S.write(false);
          state = E0;
          }
        else if ( k<3 ) {
          k = k+1;
          }
        wait(SC_ZERO_TIME);
        break;
      }
    }
};
```

Listing 4.3: File inp_H.h

```
#include "systemc.h"

SC_MODULE(Inp_H)
{
  // Output
  sc_out<bool> H;

  void gen();

  SC_CTOR(Inp_H) {
    SC_THREAD(gen);
    }
};
```

Listing 4.4: File inp_H.cpp

```
#include "inp_H.h"
#include "rfsm.h"

typedef struct { int period; int t1; int t2; } _periodic_t;

static _periodic_t _clk = { 10, 0, 80 };

void Inp_H::gen()
{
  int _t=0;
  wait(_clk.t1, SC_NS);
  notify_ev(H,"H");
  _t = _clk.t1;
  while ( _t <= _clk.t2 ) {
    wait(_clk.period, SC_NS);
    notify_ev(H,"H");
    _t += _clk.period;
    }
};
```

Listing 4.5: File inp_E.h

```
#include "systemc.h"

SC_MODULE(Inp_E)
{
  // Output
  sc_out<sc_uint<1> > E;

  void gen();

  SC_CTOR(Inp_E) {
    SC_THREAD(gen);
    }
};
```

Listing 4.6: File inp_E.cpp

```
#include "inp_E.h"
#include "rfsm.h"

typedef struct { int date; int val; } _vc_t;
static _vc_t _vcs[3] = { {0,0}, {25,1}, {35,0} };

void Inp_E::gen()
{
  int _i=0, _t=0;
  while ( _i < 3 ) {
    wait(_vcs[_i].date-_t, SC_NS);
    E = _vcs[_i].val;
    _t = _vcs[_i].date;
    _i++;
    }
};
```

Listing 4.7: File main.cpp

```
#include "systemc.h"
#include "rfsm.h"
#include "inp_H.h"
#include "inp_E.h"
#include "g.h"

int sc_main(int argc, char *argv[])
{
  sc_signal<bool> H;
  sc_signal<bool> E;
  sc_signal<bool> S;
  sc_trace_file *trace_file;
  trace_file = sc_create_vcd_trace_file ("main");
  sc_write_comment(trace_file, "Generated by RFSM v2.0");
  sc_trace(trace_file, H, "H");
  sc_trace(trace_file, E, "E");
  sc_trace(trace_file, S, "S");

  Inp_H Inp_H("Inp_H");
  Inp_H(H);
  Inp_E Inp_E("Inp_E");
  Inp_E(E);

  G g("g");
  g(H,E,S);

  sc_start(100, SC_NS);

  sc_close_vcd_trace_file (trace_file);

  return EXIT_SUCCESS;
}
```

Appendix A3 - Example of generated VHDL code

This is the code generated from program given in Listing 2.1 by the VHDL backend.

Listing 4.8: File g.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.rfsm.all;

entity G is
  port( H: in std_logic;
        E: in std_logic;
        S: out std_logic;
        rst: in std_logic
        );
end entity;

architecture RTL of G is
  type t_state is ( E0, E1 );
  signal state: t_state;
begin
  process(rst, H)
  variable k: integer;
  begin
    if ( rst='1' ) then
      state <= E0;
      S <= '0';
    elsif rising_edge(H) then
      case state is
      when E0 =>
        if ( E = '1' ) then
          k := 1;
          S <= '1';
          state <= E1;
        end if;
      when E1 =>
        if ( k = 3 ) then
          S <= '0';
          state <= E0;
        elsif ( k<3 ) then
          k := k+1;
        end if;
      end case;
      end if;
    end process;
end architecture;
```

Listing 4.9: File main_top.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity main_top is
  port(
        H: in std_logic;
        E: in std_logic;
        S: out std_logic;
        rst: in std_logic        );
end entity;

architecture struct of main_top is

component G
  port(
        H: in std_logic;
        E: in std_logic;
        S: out std_logic;
        rst: in std_logic
        );
end component;


begin
  G0: G port map(H,E,S,rst);
end architecture;
```

Listing 4.10: File main_tb.vhd

```
library ieee;
use ieee.std_logic_1164.all;

use work.rfsm.all;
entity main_tb is
end entity;

architecture struct of main_tb is

component main_top is
  port(
        H: in std_logic;
        E: in std_logic;
        S: out std_logic;
        rst: in std_logic        );
end component;
```

```
signal H: std_logic;
signal E: std_logic;
signal S: std_logic;
signal rst: std_logic;

begin

  inp_H: process
    type t_periodic is record period: time; t1: time; t2: time; end record;
    constant periodic : t_periodic := ( 10 ns, 10 ns, 80 ns );
    variable t : time := 0 ns;
    begin
      H <= '0';
      wait for periodic.t1;
      t := t + periodic.t1;
      while ( t < periodic.t2 ) loop
        H <= '1';
        wait for periodic.period/2;
        H <= '0';
        wait for periodic.period/2;
        t := t + periodic.period;
      end loop;
      wait;
  end process;
  inp_E: process
    type t_vc is record date: time; val: std_logic; end record;
    type t_vcs is array ( 0 to 2 ) of t_vc;
    constant vcs : t_vcs := ( (0 ns,'0'), (25 ns,'1'), (35 ns,'0') );
    variable i : natural := 0;
    variable t : time := 0 ns;
    begin
      for i in 0 to 2 loop
        wait for vcs(i).date-t;
        E <= vcs(i).val;
        t := vcs(i).date;
      end loop;
      wait;
  end process;
  reset: process
  begin
    rst <= '1';
    wait for 1 ns;
    rst <= '0';
    wait for 100 ns;
    wait;
  end process;

  Top: main_top port map(H,E,S,rst);

end architecture;
```