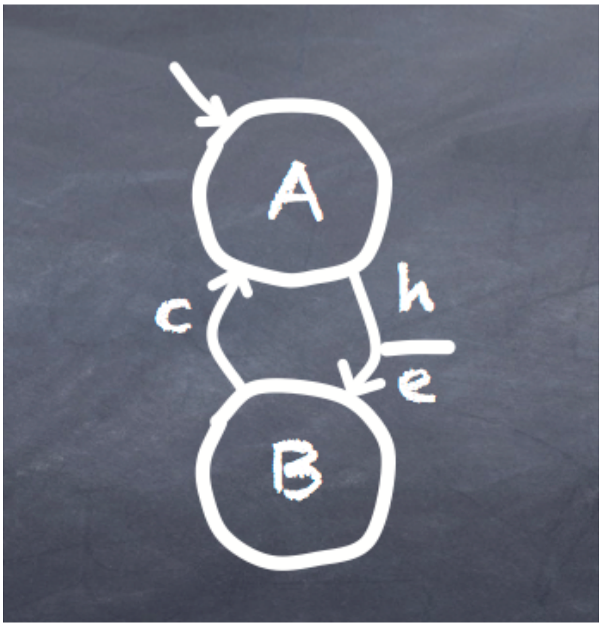
RFSM User Manual - 1.6

J. Sérot



Chapter 1

Introduction

This document is a brief user manual for the RFSM toolset. It is, in its current form, very preliminary, but should suffice for a quick grasp of the provided tools.

RFSM is a set of tools aimed at describing, drawing and simulating *reactive finite state machines*. Reactive FSMs are a 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 to 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.

RFSM is actually composed of three distinct tools :

- a command-line compiler (`rfsmc`),
- a graphical user-interface (GUI) to the compiler,
- a library for the OCaml programming language.

These tools 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)

This document is organized as follows. Chapter 2 is an informal presentation of the RFSM language and of its possible usages. Chapter 3 describes how to use the command-line compiler. Chapter **??** describes the GUI-based application. Appendix A gives the detailed syntax of the language. Appendix B summarizes the compiler options. Appendices C1, C2 and C3 give some examples of code generated by the C, SystemC and VHDL backends.

Chapter 2

Overview

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

2.1 Introductory example

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.

The program can be divided in four 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 :

| source_state −> destination_state **on** ev **when** guard **with** 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` and sets its internal variable `k` and its output `s` to 1. The *initial transition* of the FSM is given after the `itrans:` keyword in the form :

| −> initial_state **with** actions

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

A graphical representation of the `gensig` model is given in Fig. 2.1 (this representation was actually automatically generated from the program in Listing 2.1, as explained in Chap. 3).

[1]This program is provided in the distribution, under directory `examples/single/gensig/v2`.

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<0: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<4>(H,E,S)
```

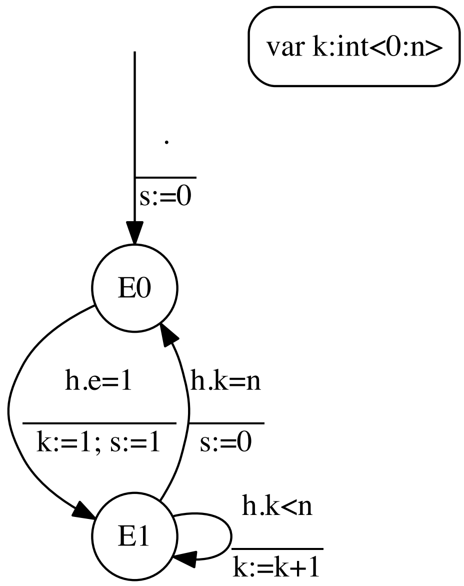


Figure 2.1: A graphical representation of FSM model defined in Listing 2.1

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**[2]. 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 (4 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].

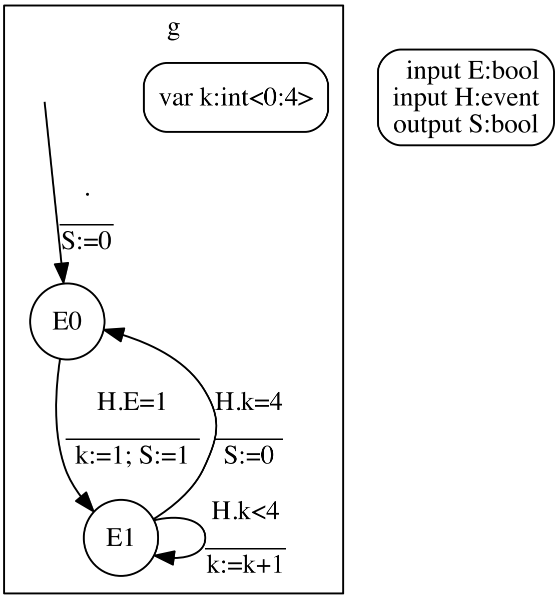


Figure 2.2: A graphical representation of system described in Listing 2.1

Simulating

Simulating the program means computing the reaction of the system to the input stimuli. Simulation can be performed the RFSM command-line compiler or the IDE (see Chap. 3 and **??** resp.). It produces a set of *traces* in VCD (Value Change Dump) format which can visualized using *waveform viewers* such as `gtkwave`. The simulation results for the program in Listing 2.1 are illustrated in Fig. 2.3.

[2]In case of multi-FSM programs, this part will also contains the declaration of *shared* events and variables. See Sec. 2.2.3.

[3]See Sec. 2.2.3 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. 3

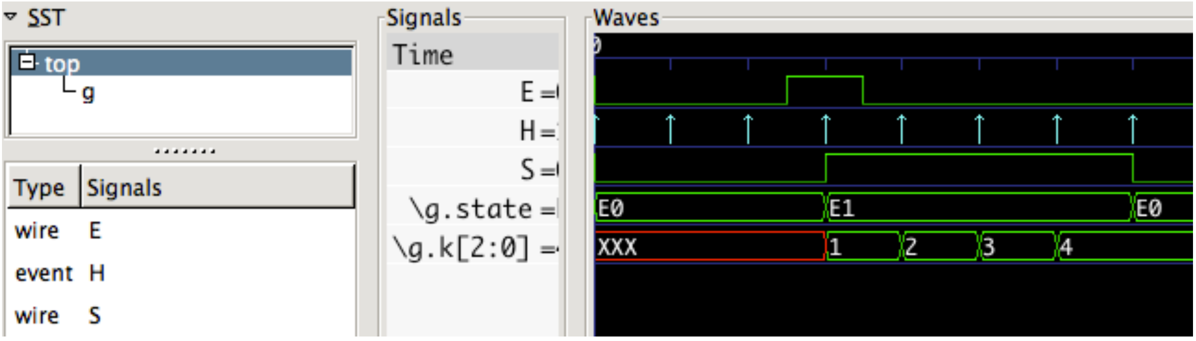


Figure 2.3: Simulation results for the program in Listing 2.1, viewed using `gtkwave`

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 `iinout` 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 and. The FSM implementations generated by the VHDL backend can also be synthetized to be implemented hardware using hardware-specific tools[5].

Appendices C1, C2 and C3 respectively give the C and SystemC code generated from the example in Listing 2.1.

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

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

Table 2.1: Builtin operations on integers

2.2 The RFSM language

This section is more thorough presentation of the RFSM language introduced in the previous section. This presentation is deliberately informal. The complete language syntax can be found in Appendix A.

2.2.1 Types

There are two categories of types : builtin types and user defined types.

Builtin types are : bool, int, float, char, event and arrays.

▶ 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 constants or expressions whose value can be computed as compile time (expressions involving parameter values, as exemplified line 9 in Listing 2.1).

▶ Supported operations on values of type int are described in Table 2.1. 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 mus 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.

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

▶ Arrays are 1D, fixed-size collections of ints, bools or floats. 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

type typename = 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

type typename = **enum** { 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

type typename = **record** { 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

2.2.2 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. 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 `<...>`. Examples :

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**: ...; }

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;

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.

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 event trigerring the transition,
- *guards* is a set a enabling conditions,
- *actions* is a set of actions performed when then transition is enabled.

`int`	`+ - * / mod = != > < >= <=`
`bool`	`= !=`
`enumeration`	`= !=`

Table 2.2: Operations on types

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 set of allowed operations is given in Table 2.2.

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 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

Initial transition

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

| | −> init_state **with** 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 :

| | −> init_state |

2.2.3 Globals

Globals are used to connect model instances to the external world or to other instances.

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 :

output name : typ

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

input 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.

periodic(period,t0,t1)

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

sporadic(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[6].

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.

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

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 (see Section 2.2.4).

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

shared name : typ

2.2.4 Instances and system

The description of the system is carried out by instanciating – and, possibly, inter-connecting – previously defined FSM models.

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<4>(H,E,S)

creates an instance of model `gensig` for which `n=4` and whose inputs (resp. output) are connected to the global inputs (resp. output) `H` and `E` (resp. `S`).

Multi-FSM models

It is of course possible to build a system model as a *composition* of FSM instances. An example is given in Listing 2.2. The system is a simple modulo 8 counter, here described as a combination of three event-synchronized modulo 2 counters[7].

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`.

The graphical representation of the program is given in Fig. 2.4. Simulation results are illustrated in Fig 2.5.

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

Listing 2.2: A multi-model RFSM program

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

input H: event = periodic(10,10,100)
output S0, S1, S2: int<0:1>
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)
```

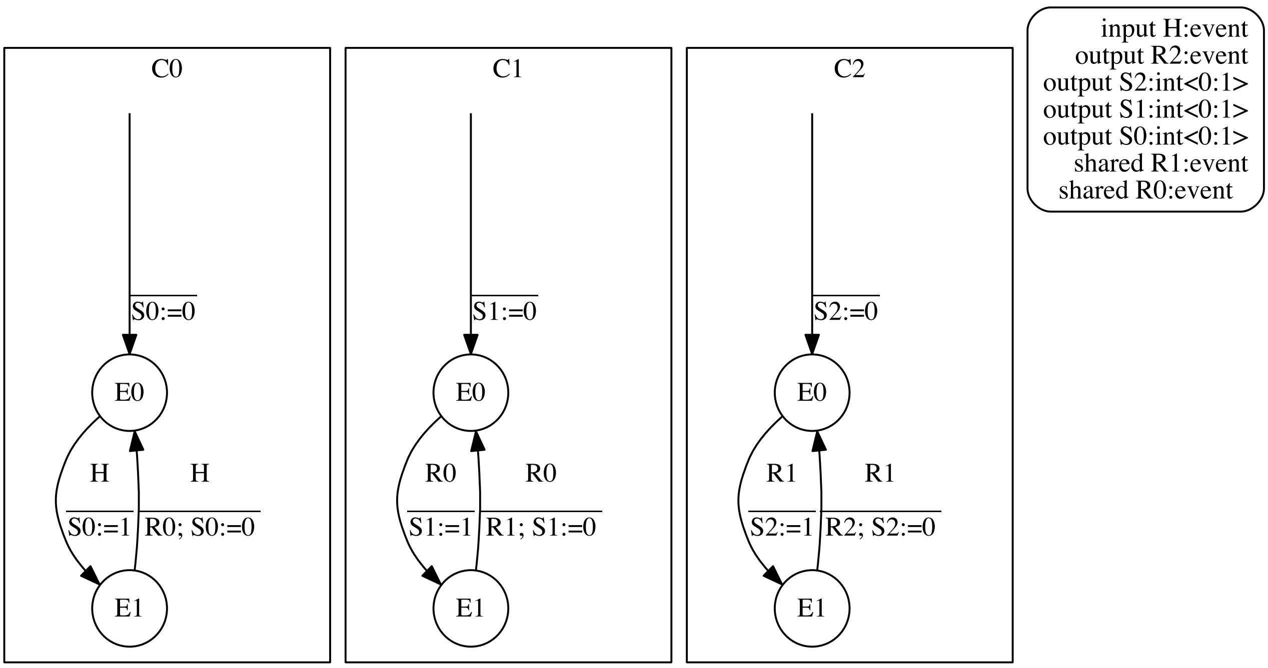


Figure 2.4: A graphical representation of program described in Listing 2.2

Figure 2.5: Simulation results for the program in Listing 2.2

2.2.5 Functions

Conditions and actions associated to FSM transitions can use globally defined functions. An example is given in listing 2.3[8]. The FSM described here computes an approximation 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 so-called *macros* in the C language than full-fledged functions and are typically used to improve readability of the programs.

[8] This example can be found in directory `examples/heron/v2` in the distribution.

Listing 2.3: 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, Iter;
  vars: a: float, x: float, n: int;
  trans:
  | Idle -> Iter on h when start=1 with a:=u, x:=u, rdy:=0, 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, rdy:=1;
  itrans:
  | -> Idle with rdy:=1;
}

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 Rdy : bool
output R : float
output niter : int

fsm heron = Heron<0.00000001> (H,Start,U,Rdy,niter,R)
```

2.2.6 Constants

Global constants can be defined using the following syntax :

$$\boxed{\text{constant name : } \langle \textbf{type} \rangle = \langle \text{value} \rangle}$$

where

- <**type**> is the type of the defined constant (currently limited to `int`, `float` and arrays of `ints` or `floats`,
- <value> is the value of the constant (which must be an `int` or `float` literal or an array of such literals).

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

2.2.7 Semantic issues

This presentation of the language has deliberately focused on syntax. Formalizing the semantics of programs made of reactive finite state machines – and in particular when several of these machines are interacting – is actually far from trivial and will not be carried out here.

Instead, this section will describe some "practical" problems that may arise when simulating such systems and how the language currently addresses them, without delving too much into the underlying semantics issues[9].

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 2.4. 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[0]
- Running--startstop->Stopped[0]
```

Of course, this could be avoided by modifying the stimuli attached to input `StartStop` so that the corresponding events are never emitted at time $t = n \times 10$. But this is, in a sence, cheating, since this event is supposed to modelize user interaction which occur, by essence, at impredictible dates.

The above problem can be solved by assigning a *priority* to transitions. In the current implementation, this is achieved by tagging some transitions as "high priority" transitions[10]. When several transitions are enabled, if one is tagged as "high priority" than it is automatically selected[11].

[9]This is not that these issues do not deserve a formal treatment. Of course, they do ! But we think we this document is not the right place to do it.

[10]Future versions may evolve towards a more sophisticated mechanism allowing numeric priorities.

[11]If none (resp. several) is (resp. are) tagged, the conflict remains, of course.

Listing 2.4: 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)
```

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 2.5. Tagging the last transition is here equivalent to give to the `startstop` precedence against the `h` event when the model is in state `Running`.

Sequential vs. synchronous actions

An important question is whether, when a transition specifying *several actions* to be performed is taken, the corresponding actions are performed sequentially or not.

Consider for example, the following transition, in which `x` and `y` are internal variables of the enclosing FSM :

```
S0 -> S1 on H with x:=x+1, y:=x*2
```

Suppose that the value of variable `x` is 1 just before event `H` occurs. What will the value of variables `x` and `y` after this transition ?

▶ With a **sequential interpretation**, actions are performed sequentially, one after the other, in the order they are specified. With this interpretation, order of execution matters. In the example above, it will assign the value 2 to `x` and 4 to `y`.

▶ With a **synchronous interpretation**, actions are performed in parallel, the value of each variable occuring in right-hand-side expressions being the one *before* the transition. With this interpretation, order of executions does *not* matter. In the example above, it will assign the value 2 to `x` and 2 to `y`.

A sequential interpretation naturally fits a software execution model, in which FSM variables are implemented as program variables and actions as immediate modifications of these variables, whereas

Listing 2.5: A rewriting of the model defined in Listing 2.4

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

a synchronous interpretation reflects hardware execution models, in which FSM variables are typically implemented as registers which are updated in parallel at each clock cycle.

By default, the `rfsmc` compiler relies on a sequential interpretation, both for simulation and code production[12]. But, in certain cases, and in particular when specifying models to be synthetized on hardware, a synchronous interpretation is more natural and/or can lead to more efficient implementations. Switching to a synchronous interpretation is possible by invoking the `rfsmc` compiler with the `-synchronous_actions` option[13].

[12]For the C and SystemC backends, this means that FSM variables are implemented as local variables of the function implementing the FSM model. For the VHDL backend, these variables are implemented as `variable`s withing the process implementing the FSM.

[13]For the VHDL backend, in particular, the `-synchronous_actions` option forces the FSM variables to be implemented as `signal`s.

Chapter 3

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 App. 3.6.

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

3.1 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`.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

3.6 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[4]). 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
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[5]

- `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] The `.pro` file is also used by the GUI described in chapter **??**.

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

Appendix A - Formal syntax of RFSM programs

This appendix gives a BNF definition of the concrete syntax RFSM programs.

The meta-syntax is conventional. Keywords are written in **boldface**. Non-terminals are enclosed in angle brackets (`<` ... `>`). Vertical bars (`|`) indicate alternatives. Constructs enclosed in non-bold brackets (`[` ... `]`) are optional. The notation E^* (resp E^+) means zero (resp one) or more repetitions of E, separated by spaces. The notation E^*_x (resp E^+_x) means zero (resp one) or more repetitions of E, separated by symbol x. Terminals `lid` and `uid` respectively designate identifiers starting with a lowercase and uppercase letter.

⟨program⟩ ::= ⟨decl⟩*

⟨decl⟩ ::= ⟨type_decl⟩
  | ⟨cst_decl⟩
  | ⟨fn_decl⟩
  | ⟨fsm_model⟩
  | ⟨fsm_inst⟩
  | ⟨global⟩

⟨type_decl⟩ ::= **type** lid = ⟨type_expr⟩
  | **type** lid = **enum** { uid$^*_,$ }
  | **type** lid = **record** { ⟨record_field⟩$^+_,$ }

⟨record_field⟩ ::= lid : ⟨type_expr⟩

⟨cst_decl⟩ ::= **constant** lid : ⟨fres⟩ = ⟨const⟩

⟨fn_decl⟩ ::= **function** lid (⟨farg⟩$^*_,$) : ⟨fres⟩ { **return** ⟨fbody⟩ }

⟨farg⟩ ::= lid : ⟨type_expr⟩

⟨fres⟩ ::= ⟨type_expr⟩

⟨fbody⟩ ::= ⟨expr⟩

⟨fsm_model⟩ ::= **fsm model** ⟨id⟩ [⟨params⟩] (⟨io⟩$^*_,$) {
  states : uid$^*_,$;
  [⟨vars⟩]
  trans : ⟨transition⟩* ;
  itrans : ⟨itransition⟩ ;
  }

⟨params⟩ ::= < ⟨param⟩$^*_,$ >

⟨param⟩ ::= lid : ⟨type_expr⟩

⟨io⟩ ::= **in** ⟨io_desc⟩
  | **out** ⟨io_desc⟩
  | **inout** ⟨io_desc⟩

⟨io_desc⟩ ::= lid : ⟨type_expr⟩

⟨vars⟩ ::= **vars** : ⟨var⟩$^*_,$;

⟨var⟩ ::= lid$^+_,$: ⟨type_expr⟩

⟨transition⟩ ::= ⟨trans_mark⟩ uid -> uid **on** lid [⟨guard⟩] [⟨actions⟩]

⟨trans_mark⟩ ::= |
  | !

⟨itransition⟩ ::= | -> uid [⟨actions⟩]

⟨guard⟩ ::= **when** ⟨expr⟩$^{+}_{.}$

⟨actions⟩ ::= **with** ⟨action⟩$^{+}_{,}$

⟨action⟩ ::= lid
| ⟨lhs⟩ := ⟨expr⟩

⟨lhs⟩ ::= lid
| lid [⟨expr⟩]
| lid [⟨expr⟩ : ⟨expr⟩]
| lid . lid

⟨global⟩ ::= **input** ⟨id⟩ : ⟨type_expr⟩ = ⟨stimuli⟩
| **output** ⟨id⟩$^{+}_{,}$: ⟨type_expr⟩
| **shared** ⟨id⟩$^{+}_{,}$: ⟨type_expr⟩

⟨stimuli⟩ ::= **periodic** (**int** , **int** , **int**)
| **sporadic** (**int**$^{*}_{,}$)
| **value_changes** (⟨value_change⟩$^{*}_{,}$)

⟨value_change⟩ ::= **int** : ⟨const⟩

⟨fsm_inst⟩ ::= **fsm** ⟨id⟩ = ⟨id⟩ [< ⟨inst_param_value⟩$^{+}_{,}$ >] (⟨id⟩$^{*}_{,}$)

⟨inst_param_value⟩ ::= ⟨constant⟩
| lid
| [⟨constant⟩$^{+}_{,}$]

⟨type_expr⟩ ::= **event**
| **int** ⟨int_annot⟩
| **float**
| **char**
| **bool**
| lid
| ⟨type_expr⟩ **array** [⟨array_size⟩]

⟨int_annot⟩ ::= ϵ
| < ⟨type_index_expr⟩ >
| < ⟨type_index_expr⟩ : ⟨type_index_expr⟩ >

⟨array_size⟩ ::= ⟨type_index_expr⟩

⟨type_index_expr⟩ ::= ⟨int_const⟩
| lid
| (⟨type_index_expr⟩)
| ⟨type_index_expr⟩ + ⟨type_index_expr⟩

```
               |    ⟨type_index_expr⟩ - ⟨type_index_expr⟩
               |    ⟨type_index_expr⟩ * ⟨type_index_expr⟩
               |    ⟨type_index_expr⟩ / ⟨type_index_expr⟩
               |    ⟨type_index_expr⟩ % ⟨type_index_expr⟩

        ⟨expr⟩ ::=  ⟨simple_expr⟩
               |    ⟨expr⟩ >> ⟨expr⟩
               |    ⟨expr⟩ << ⟨expr⟩
               |    ⟨expr⟩ & ⟨expr⟩
               |    ⟨expr⟩ || ⟨expr⟩
               |    ⟨expr⟩ ^ ⟨expr⟩
               |    ⟨expr⟩ + ⟨expr⟩
               |    ⟨expr⟩ - ⟨expr⟩
               |    ⟨expr⟩ * ⟨expr⟩
               |    ⟨expr⟩ / ⟨expr⟩
               |    ⟨expr⟩ % ⟨expr⟩
               |    ⟨expr⟩ +. ⟨expr⟩
               |    ⟨expr⟩ -. ⟨expr⟩
               |    ⟨expr⟩ *. ⟨expr⟩
               |    ⟨expr⟩ /. ⟨expr⟩
               |    ⟨expr⟩ == ⟨expr⟩
               |    ⟨expr⟩ != ⟨expr⟩
               |    ⟨expr⟩ > ⟨expr⟩
               |    ⟨expr⟩ < ⟨expr⟩
               |    ⟨expr⟩ >= ⟨expr⟩
               |    ⟨expr⟩ <= ⟨expr⟩
               |    ⟨subtractive⟩ ⟨expr⟩
               |    lid ( ⟨expr⟩*, )
               |    lid [ ⟨expr⟩ ]
               |    lid . lid
               |    lid [ ⟨expr⟩ : ⟨expr⟩ ]
               |    ⟨expr⟩ ? ⟨expr⟩ : ⟨expr⟩
               |    ⟨expr⟩ :: ⟨type_expr⟩

 ⟨simple_expr⟩ ::=  lid
               |    ⟨constant⟩
               |    uid
               |    ( ⟨expr⟩ )

    ⟨constant⟩ ::=  int
               |    float
               |    char

 ⟨subtractive⟩ ::=  -
               |    -.

       ⟨const⟩ ::=  ⟨scalar_const⟩
               |    ⟨array_const⟩
               |    ⟨record_const⟩
```

$$
\begin{array}{rcl}
\langle \text{array\_const} \rangle & ::= & \texttt{[}\ \langle \text{const} \rangle^{+}_{,}\ \texttt{]} \\[1em]
\langle \text{record\_const} \rangle & ::= & \{\ \langle \text{record\_field\_const} \rangle^{+}_{,}\ \} \\[1em]
\langle \text{record\_field\_const} \rangle & ::= & \text{lid}\ \mathbf{=}\ \langle \text{scalar\_const} \rangle \\[1em]
\langle \text{scalar\_const} \rangle & ::= & \langle \text{int\_const} \rangle \\
& | & \langle \text{float\_const} \rangle \\
& | & \langle \text{char\_const} \rangle \\
& | & \text{uid} \\[1em]
\langle \text{int\_const} \rangle & ::= & \textbf{int} \\
& | & \texttt{-}\ \textbf{int} \\[1em]
\langle \text{float\_const} \rangle & ::= & \textbf{float} \\
& | & \texttt{-}\ \textbf{float} \\[1em]
\langle \text{char\_const} \rangle & ::= & \textbf{char} \\[1em]
\langle \text{id} \rangle & ::= & \text{lid} \\
& | & \text{uid}
\end{array}
$$

Appendix B - Compiler options

Compiler usage : `rfsmc [options...] files`

-lib	set location of the support library (default: ¡opam_prefix¿/share/rfsm)
-main	set prefix for the generated main files
-dump_static	dump static representation of model(s)/program to stdout
-target_dir	set target directory (default: .)
-dot	generate .dot representation of model(s)/program
-ctask	generate CTask code
-systemc	generate SystemC code
-vhdl	generate VHDL code
-sim	run simulation (generating .vcd file)
-version	print version of the compiler and quit
-dot_no_captions	Remove captions in .dot representation(s)
-dot_actions_nl	write actions with with a separating newline
-trace	set trace level for simulation (default: 0)
-synchronous_actions	interpret actions synchronously
-sc_time_unit	set time unit for the SystemC test-bench (default: SC_NS)
-sc_trace	set trace mode for SystemC backend (default: false)
-stop_time	set stop time for the SystemC and VHDL test-bench (default: 100)
-sc_double_float	implement float type as C++ double instead of float (default: false)
-vhdl_trace	set trace mode for VHDL backend (default: false)
-vhdl_time_unit	set time unit for the VHDL test-bench
-vhdl_ev_duration	set duration of event signals (default: 1 ns)
-vhdl_rst_duration	set duration of reset signals (default: 1 ns)
-vhdl_numeric_std	translate integers as numeric_std [un]signed (default: false)
-vhdl_bool_as_bool	translate all booleans as boolean (default: false)
-vhdl_dump_ghw	make GHDL generate trace files in .ghw format instead of .vcd
-old_syntax	use old (pre-1.5) syntax
-transl_syntax	convert old syntax to new syntax

Appendix C1 - Example of generated C code

This is the code generated from program given in Listing 2.1

```
task g(
  in event h;
  in int e;
 out int s;
  )
{
  int k;
  enum {E0,E1} state = E0;
  s=0;
  while ( 1 ) {
    switch ( state ) {
    case E1:
      wait_ev(h);
      if ( k<4 ) {
        k=k+1;
        }
      else if ( k==4 ) {
        s=0;
        state = E0;
        }
      break;
    case E0:
      wait_ev(h);
      if ( e==1 ) {
        k=1;
        s=1;
        state = E1;
        }
      break;
    }
  }
};
```

Appendix C1 - Example of generated SystemC code

This is the code generated from program given in Listing 2.1

Listing 3.1: File g4.h

```
#include "systemc.h"

SC_MODULE(G)
{
  // Types
  typedef enum { E0, E1 } t_state;
  // IOs
  sc_in<bool> h;
  sc_in<sc_uint<1> > e;
  sc_out<sc_uint<1> > s;
  // Constants
  static const int n = 4;
  // Local variables
  t_state state;
  sc_uint<3>  k;

  void react();

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

Listing 3.2: File g.cpp

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

void G::react()
{
  state = E0;
  s.write(0);
  while ( 1 ) {
    switch ( state ) {
    case E1:
      wait(h.posedge_event());
```

```
      if ( k<4 ) {
        k=k+1;
        }
      else if ( k==4 ) {
        s.write(0);
        state = E0;
        }
      wait(SC_ZERO_TIME);
      break;
    case E0:
      wait(h.posedge_event());
      if ( e.read()==1 ) {
        k=1;
        s.write(1);
        state = E1;
        }
      wait(SC_ZERO_TIME);
      break;
    }
  }
};
```

Listing 3.3: 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 3.4: 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 3.5: 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 3.6: 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 3.7: File tb.cpp

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

int sc_main(int argc, char *argv[])
{
   sc_signal<sc_uint<1> > E;
   sc_signal<bool> H;
   sc_signal<sc_uint<1> > S;
```

```
  sc_trace_file *trace_file;
  trace_file = sc_create_vcd_trace_file ("tb");
  sc_trace(trace_file, E, "E");
  sc_trace(trace_file, H, "H");
  sc_trace(trace_file, S, "S");

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

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

  sc_start(100, SC_NS);

  sc_close_vcd_trace_file (trace_file);

  return EXIT_SUCCESS;
}
```

Appendix C3 - Example of generated VHDL code

This is the code generated from program given in Listing 2.1

Listing 3.8: File g.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library rfsm;
use rfsm.core.all;

entity g is
  port(
        h: in std_logic;
        e: in std_logic;
        s: out std_logic;
        rst: in std_logic
        );
end g;

architecture RTL of g is
  type t_state is ( E0, E1 );
  signal state: t_state;
  signal k: unsigned(2 downto 0);
begin
  process(rst, h)
  begin
    if ( rst='1' ) then
      state <= E0;
      s <= '0';
    elsif rising_edge(h) then
      case state is
      when E1 =>
        if ( k<to_unsigned(4,3) ) then
          k <= k+to_unsigned(1,3);
        elsif ( k = to_unsigned(4,3) ) then
          s <= '0';
          state <= E0;
        end if;
      when E0 =>
        if ( e = '1' ) then
```

```
          k <= to_unsigned(1,3);
          s <= '1';
          state <= E1;
        end if;
    end case;
    end if;
  end process;
end RTL;
```

Listing 3.9: File tb.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library rfsm;
use rfsm.core.all;

entity tb is
end tb;

architecture Bench of tb is

component g
  port(
        h: in std_logic;
        e: in std_logic;
        s: out std_logic;
        rst: in std_logic
        );
end component;

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

begin

  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;
  inp_H: process
    type t_periodic is record period: time; t1: time; t2: time; end record;
```

```
    constant periodic : t_periodic := ( 9 ns, 0 ns, 80 ns );
    variable t : time := 0 ns;
    begin
      H <= '0';
      wait for periodic.t1;
      notify_ev(H,1 ns);
      while ( t < periodic.t2 ) loop
        wait for periodic.period;
        notify_ev(H,1 ns);
        t := t + periodic.period;
      end loop;
      wait;
  end process;

  U0: G port map(H,E,S,rst);

  process

  begin
    rst <= '1';
    wait for 1 ns;
    rst <= '0';
    wait for 100 ns;
    wait;

  end process;
end Bench;
```