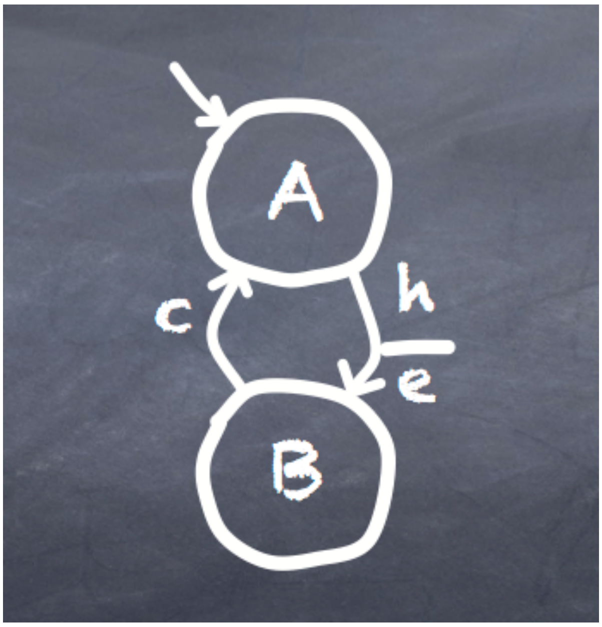
RFSM User Manual - 2.0

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 5 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.

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

The program can be divided in four parts.

The first part (lines 1–15) 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

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

`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`.

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`.

The value of the `s` output here is attached to states using the **where** keyword : this value is 0 when the system is in state `E0` and 1 when the system is in state `E1`.

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 actually automatically generated from the program in Listing 2.1, as explained in Chap. 5).

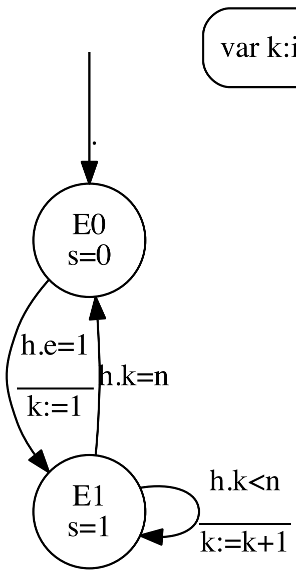


Figure 2.1: A graphical representation of FSM model defined in Listing 2.1

Note that, at this level, the value of the parameter `n`, used in the type of the internal variable `k` (line 8) and in the transition conditions (lines 11 and 12) is left unspecified, making the `gensig` model a *generic* one.

The second part of the program (lines 17–19) 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[2]. 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 21) 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[3].

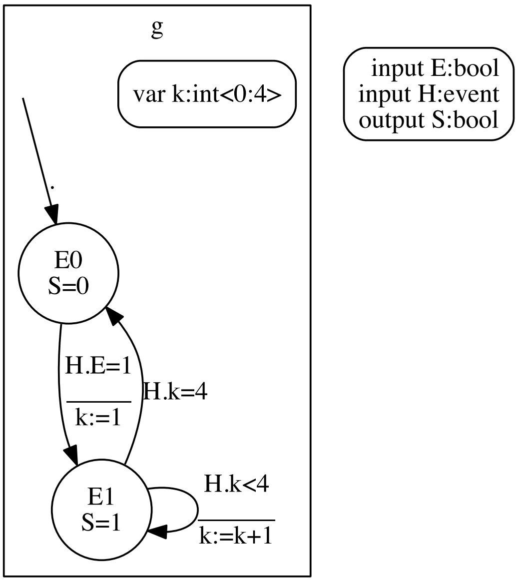


Figure 2.2: A graphical representation of system described in Listing 2.1

Simulating

Simulating the program means computing the reaction of the system to the input stimuli. Simulation can be performed the RFSM command-line compiler as described in chapter 5. It produces a set of

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

[3]Again, this representation was actually automatically generated from the program in Listing 2.1, as explained in Chap. 5

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 showed in Fig. 2.3.

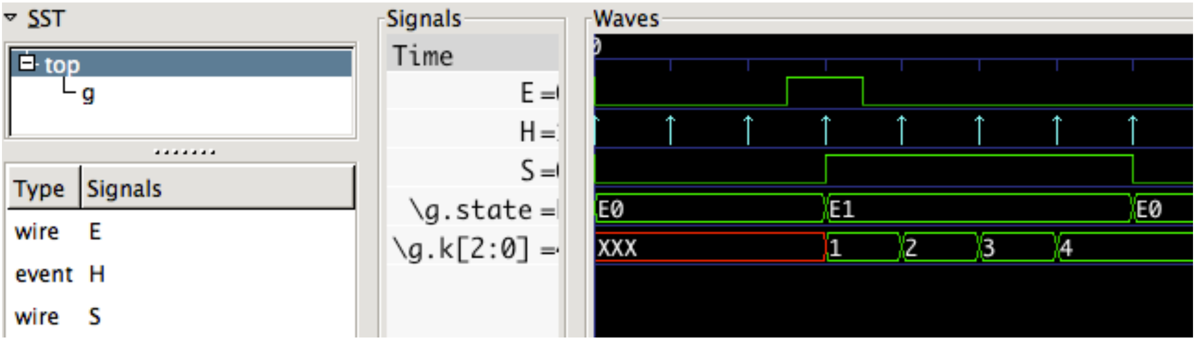


Figure 2.3: Simulation results for the program in Listing 2.1, viewed using `gtkwave`

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.

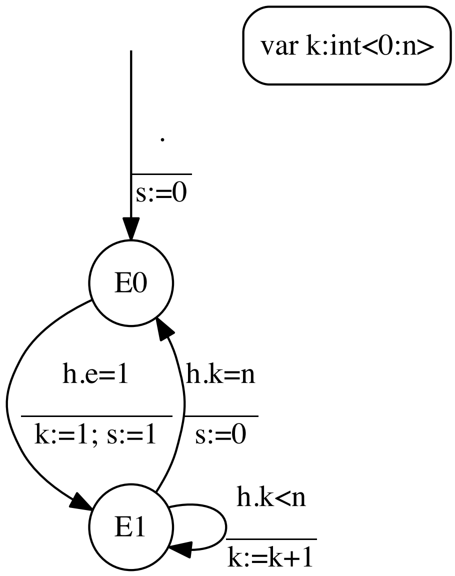
For the SystemC and VHDL backends, the generated code can actually be compiled and executed for simulation purpose and. The FSM implementations generated by the VHDL backend can also be synthetized to be implemented hardware using hardware-specific tools[4].

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

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

Variant formulation

In the automata described in Fig. 2.1 and Listing 2.1, the `s` output is defined by attaching its value to states. This is typical of a so-called *Moore*-style description. Iy is also possible to specify these values by indicating how they are *modified* when some transitions are taken. A equivalent description of that given in Listing 2.1 is obtained, for example, by specifying that `s` is set to 0 on the initial transition and on the transition from `E1` to `E0`, and set to 1 on the transition from `E0` to `E1`. This style of description, often called *Mealy*-style, is illustrated in Fig. 2.4. Note the absence of the `where` clause in the declarations of states and, conversely, the presence of the action s:=0 and s:=1 in the first and third transitions respectively.




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

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

Note. An option of the `rfsmc` compiler (`-normalize`) allows to automatically transform a Moore-style description into a Mealy-style one.

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[5].

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

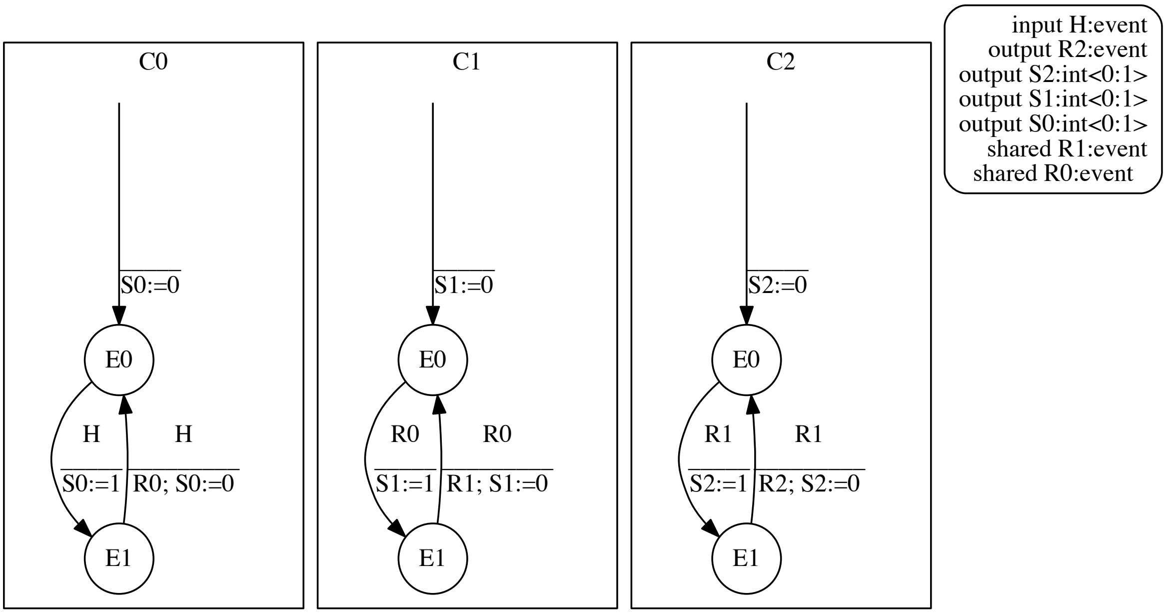


Figure 2.5: Graphical representation of the program of Listing 2.2

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

```
fsm model cntmod2(
  in h: event,
  out s: 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)
```

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

Figure 2.6: Simulation results for the program in Listing 2.5

and Fig. 2.7[6]. 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)
```

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

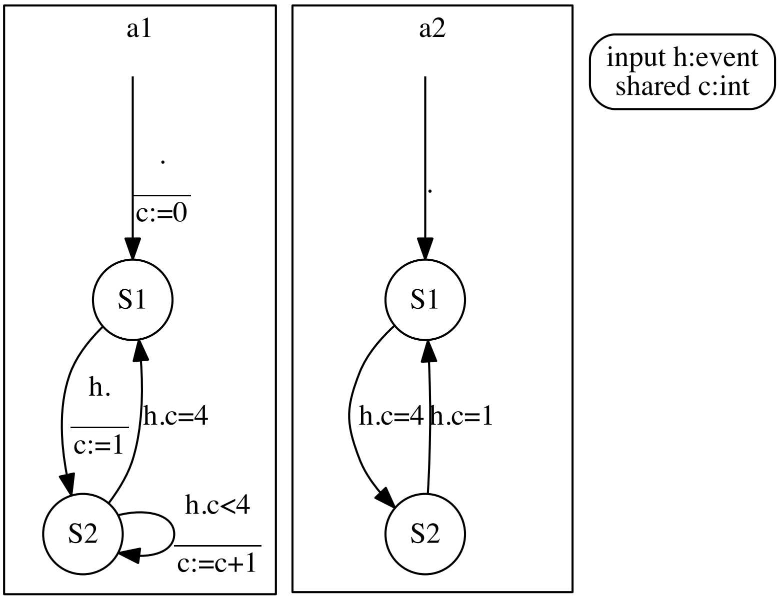


Figure 2.7: Graphical representation of the program of Listing 2.3

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

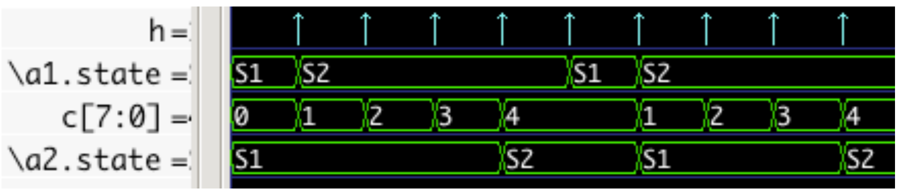


Figure 2.8: Simulation results for the program in Listing 2.7

Chapter 3

Syntax

This section is more thorough presentation of the RFSM language, focusing on its syntax. Semantics issues are discussed in chapter 4. A formal description of the syntax, in BNF, is given in Appendix A.

3.1 Programs

A RSFM program contains declarations of six kinds :

- **types**,
- **constants**,
- **functions**,
- **FSM models**,
- **global objects**,
- **FSM instanciations**.

These declarations may appear in any order in a given program but an object used in a given section must have been declared before.

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 :

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;

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;

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.

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

Table 3.1: Operations on types

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.

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 3.1.

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

Note. 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

states: S **where** o=v, ...

is equivalent to adding the action

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 :

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[1].

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.

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

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`.

3.5 FSM instances

The description of the system is carried out by instanciating 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`).

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`, `float` and arrays of `int`s or `float`s,
- <value> is the value of the constant (which must be an `int` or `float` literal or an array of such literals).

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 3.1[2]. 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 *macros* in the C language than full-fledged functions and are typically used to improve readability of the programs.

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 `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` :

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

Listing 3.1: 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)
```

<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 3.2: Builtin operations on integers

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 3.2. 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

Chapter 4

Formal semantics

We give formal *static* and *dynamic* semantics for a simplified version of the RFSM language, called CORE RFSM. Compared to the "full" RFSM language, it lacks type, constant and function declarations, state valuation and has only basic types. Its abstract syntax is described below. We note X^* (resp. X^+) the repetition of 0 (resp. 1) or more X. The syntax of expressions is deliberately not explicited here.

program ::=	`program` *fsm_model*$^+$ *io_decl*$^+$ *fsm_inst*$^+$	
fsm_model ::=	`fsm model` id *inp** *outp** *state*$^+$ *var** *trans*$^+$ *itrans*	
state ::=	id	
inp, *outp*, *var* ::=	id : *typ*	
trans ::=	$\langle$id, *cond*, *action**, id$\rangle$	$\langle$*src state*, *cond*, *actions*, *dst state*$\rangle$
cond ::=	$\langle$id, *guard**$\rangle$	$\langle$*triggering even*, *guards*$\rangle$
guard ::=	*expr*	*boolean expression*
action ::=	| id	*emit event*
	| id := *expr*	*update local, shared or output variable*
io_decl ::=	*io_cat* `id` : *typ*	
io_cat ::=	`input` | `input` | `shared`	
fsm_inst ::=	`fsm` id *i** *o**	*model*, *IO bindings*
typ ::=	`event` | `int` | `bool`	

4.1 Common definitions

Both the static and static semantics will use *environments*. An **environment** is a (partial) map from *names* to *values*. If Γ is an environment and x a name, we will, classically, note

- $x \in \Gamma$ if $x \in \mathrm{dom}(\Gamma)$,
- $\Gamma(x)$ the value mapped to x in Γ ($\Gamma(x) = \bot$ if $x \notin \Gamma$),
- $\Gamma[x \mapsto v]$ the environment that maps x to v and behaves like Γ otherwise (possibly overriding an existing mapping of x),
- $\emptyset$ the empty environment,

4.2 Static semantics

The static interpretation of a Core Rfsm program is a pair

$$\mathcal{H} = \langle M, C \rangle$$

where

- M is a set of **automata**,
- C is a **context**.

▶ A **context** is a 6-tuple $\langle I_e, I_v, O_e, O_v, H_e, H_v \rangle$ where

- I_e (resp. O_e, H_e) is the set of global inputs (resp. outputs, shared values) with an *event* type,
- I_v (resp. O_v, H_v) is the set of global inputs (resp. outputs, shared values) with a *non-event* type.

▶ An **automaton** $\mu \in M$ is a 3-tuple

$$\mu = \langle \mathcal{M}, q, \mathcal{V} \rangle$$

where

- $\mathcal{M}$ is the associated (static) model,
- q its current state,
- $\mathcal{V}$ an environment giving the current value of its local variables.

▶ A **model** $\mathcal{M}$ is a 6-tuple

$$\mathcal{M} = \langle Q, I, O, V, T, \tau_0 \rangle$$

where

- Q is a (finite) set of *states*,
- I and O are environments respectively mapping input and output names to types,
- V is an environment mapping local variable names to types,
- $I_e = \{x \in I \mid I(x) = \texttt{event}\}$ and $I_v = \{x \in I \mid I(x) \neq \texttt{event}\}$,
- $O_e = \{x \in O \mid O(x) = \texttt{event}\}$ and $O_v = \{x \in O \mid O(x) \neq \texttt{event}\}$,
- $T \subset Q \times C \times \mathcal{S}(A) \times Q$ is a set of **transitions**, where
  - $C = I_e \times 2^{\mathcal{B}(I_v \cup V)}$,
  - $\mathcal{B}(E)$ is the set of *boolean expressions* built from a set of variables E and the classical boolean operators[1],

[1] This set can be formally derived from the abstract syntax.

– $\mathcal{S}(A)$ is the set of *sequences* built from elements of the set A, where a *sequence* $\vec{a}$ is an ordered collection $\langle a_1; \ldots; a_n\rangle$[2],
– $A = \mathcal{U}(O_v \cup V, I_v \cup V) \cup O_e$,
– $\mathcal{U}(E, E')$ is the set of *assignations* of variables taken in a set E by *expressions* built from a set of variable E' and the classical boolean and arithmetic operators and constants[3].

- $\tau_0 \in Q \times \mathcal{S}(\mathcal{U}(O_v \cup V, \emptyset))$ is the **initial transition**.

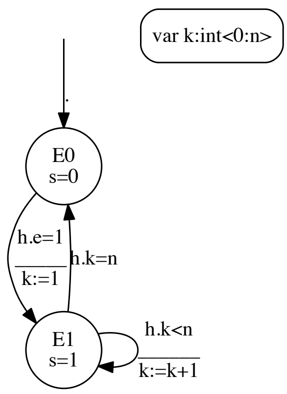
Having $\tau = (q, c, \vec{a}, q')$ in T means that there's a transition from state q to state q' enabled by the condition c and triggering a (possibly empty) sequence $\vec{a}$, where

- the condition $c \in C$ is made of
  – a trigerring event $e \in I_e$,
  – a (possibly empty) set of boolean expressions (guards), involving inputs having a non-event type or local variables,
- the actions in $\vec{a}$ consist either in the emission of an event or the modification of an output or local variable.

The initial transition τ_0 consists in a state (the initial state) and a (possibly empty) sequence of initial actions. Contrary to actions associated to "regular" transitions, initial actions cannot not emit events and the assigned values cannot depend on inputs or local variables.

Example. The model of the automaton depicted below[4] can be formally described as $\mathcal{M} = \langle Q, I, O, V, T, \tau_0\rangle$ where :

- $Q = \{E0, E1\}$
- $I = \{h \mapsto \mathsf{event}, e \mapsto \mathsf{bool}\}$
- $O = \{s \mapsto \mathsf{bool}\}$
- $V = \{k \mapsto \mathsf{bool}\}$
- $T = \{$
  $\langle E0, \langle H, \{e = 0\}\rangle, \langle\rangle, E0\rangle$,
  $\langle E0, \langle H, \{e = 1\}\rangle, \langle s \leftarrow 1; k \leftarrow 1\rangle, E1\rangle$,
  $\langle E1, \langle H, \{k < 3\}\rangle, \langle k \leftarrow k + 1\rangle, E1\rangle$,
  $\langle E1, \langle H, \{k = 3\}\rangle, \langle s \leftarrow 0\rangle, E0\rangle\}$
- $\tau_0 = \langle E0, \langle s \leftarrow 0\rangle\rangle$



[2] For example, if $A = \{1, 2\}$, then $\mathcal{S}(A) = \{\langle\rangle, \langle 1\rangle, \langle 2\rangle, \langle 1; 1\rangle, \langle 1; 2\rangle, \langle 2; 1\rangle, \langle 2; 2\rangle, \langle 1; 2; 1\rangle, \ldots\}$, where $\langle\rangle$ denotes the empty sequence.

[3] Again, this can be formally derived from the abstract syntax.

[4] This model, a calibrated pulse generator has been introduced in Fig. **??** (Chap. **??**).

Rules

▶ Rule PROGRAM gives the static interpretation of a program. The static environment Γ_{M} (resp. Γ_{I}) records the (typed) declarations of models (resp. IOs).

[Program] $\mathit{fsm\_model}^+ \rightarrow \Gamma_{\mathsf{M}}$
$\mathit{io\_decl}^+ \rightarrow \Gamma_{\mathsf{I}}$
$\Gamma_{\mathsf{M}}, \Gamma_{\mathsf{I}} \vdash \ \mathit{fsm\_inst}^+ \rightarrow M$
$C = \mathcal{L}(\Gamma_{\mathsf{I}})\texttt{program}\ \mathit{fsm\_model}^+\ \mathit{io\_decl}^+\ \mathit{fsm\_inst}^+ \rightarrow M, C$

The $\mathcal{L}$ function builds a static context C from the IO environment Γ_{I} :

$$\mathcal{L}(\Gamma_{\mathsf{I}}) = \langle I_e, I_v, O_e, O_v, H_e, H_v \rangle$$

where

$$\begin{array}{rlrl}
I_e &= \{x \in \mathrm{dom}(\Gamma_{\mathsf{I}}) \mid \Gamma_{\mathsf{I}}(x) = \langle \mathrm{input}, \texttt{event} \rangle\} & I_v &= \{x \in \mathrm{dom}(\Gamma_{\mathsf{I}}) \mid \Gamma_{\mathsf{I}}(x) = \langle \mathrm{input}, \tau \rangle,\ \tau \neq \texttt{event}\} \\
O_e &= \{x \in \mathrm{dom}(\Gamma_{\mathsf{I}}) \mid \Gamma_{\mathsf{I}}(x) = \langle \mathrm{output}, \texttt{event} \rangle\} & O_v &= \{x \in \mathrm{dom}(\Gamma_{\mathsf{I}}) \mid \Gamma_{\mathsf{I}}(x) = \langle \mathrm{output}, \tau \rangle,\ \tau \neq \texttt{event}\} \\
H_e &= \{x \in \mathrm{dom}(\Gamma_{\mathsf{I}}) \mid \Gamma_{\mathsf{I}}(x) = \langle \mathrm{shared}, \texttt{event} \rangle\} & H_v &= \{x \in \mathrm{dom}(\Gamma_{\mathsf{I}}) \mid \Gamma_{\mathsf{I}}(x) = \langle \mathrm{shared}, \tau \rangle,\ \tau \neq \texttt{event}\}
\end{array}$$

▶ Rule MODELS gives the interpretation of model declarations, giving an environment Γ_{M}.

[Models] $\forall i \in \{1, \ldots, n\} \qquad \Gamma_{\mathsf{M}}^{i-1},\ \mathit{fsm\_model}_i \rightarrow \Gamma_{\mathsf{M}}^{i}$
$\Gamma_{\mathsf{M}}^0 = \emptyset \quad \Gamma_{\mathsf{M}} = \Gamma_{\mathsf{M}}^n \mathit{fsm\_model}_1, \ldots, \mathit{fsm\_model}_n \rightarrow \Gamma_{\mathsf{M}}$

▶ Rule MODEL gives the interpretation of a single model declaration. It just records the corresponding description in the environment Γ_{M}, after performing some sanity checks, using the valid_model function, not detailed here. This function checks that :

- all variable names occuring in guards are listed as input or local variable,
- all expressions occuring in the guards of a transition have type bool,
- ...

[Model] M=$\langle Q, I, O, V, T, \tau_0 \rangle$ $\ \mathsf{valid\_model}(\mathcal{M})\texttt{fsm model}\ \mathrm{id}\ I\ O\ Q\ V\ T\ \tau_0 \rightarrow \Gamma_{\mathsf{M}}[\mathrm{id} \mapsto \mathcal{M}]$

▶ Rules IOS and IO give the interpretation of IO declarations, producing an environment Γ_{I} binding names to a pair $\langle \mathit{io\_cat}, \mathit{typ} \rangle$.

[IOs] $\forall i \in \{1, \ldots, n\} \qquad \Gamma_{\mathsf{I}}^{i-1},\ \mathit{io\_decl}_i \rightarrow \Gamma_{\mathsf{I}}^{i}$
$\Gamma_{\mathsf{I}}^0 = \emptyset \quad \Gamma_{\mathsf{I}} = \Gamma_{\mathsf{I}}^n \mathit{io\_decl}_1, \ldots, \mathit{io\_decl}_n \rightarrow \Gamma_{\mathsf{I}}$
[IO] $\Gamma_{\mathsf{I}},\ \mathit{cat}\ \texttt{id} : \mathit{typ} \rightarrow \Gamma_{\mathsf{I}}[\texttt{id} \mapsto \langle \mathit{cat}, \mathit{typ} \rangle]$

▶ Rules INSTS gives the interpretation of FSM instance declarations.

[Insts] $\forall i \in \{1, \ldots, n\} \qquad \Gamma_{\mathsf{M}}, \Gamma_{\mathsf{I}} \vdash \ \mathit{fsm\_inst}_i \rightarrow \mu_i$
$M = \langle \mu_1; \ldots; \mu_n \rangle \mathit{fsm\_inst}_1, \ldots, \mathit{fsm\_inst}_n \rightarrow \mathcal{H} = \langle M, C \rangle$

▶ Rule INST gives the interpretation of a single FSM instance as an automaton.

[Inst] $\Gamma_{\mathsf{M}}(\mathrm{id}) = \langle \langle i'_1{:}\tau'_1, \ldots; i'_m{:}\tau'_m \rangle, \langle o'_1{:}\tau''_1, \ldots; o'_n{:}\tau''_n \rangle, Q, V, T, \langle q_0, \vec{a_0} \rangle \rangle$
$\Phi = \{i'_1 \mapsto i_1, \ldots, i'_m \mapsto i_m, o'_1 \mapsto o_1, \ldots, \o'_n \mapsto o_n\}$
$\forall i \in \{1, \ldots, m\} \quad \Gamma_{\mathsf{I}}(i_i) = \langle \mathrm{cat}_i, \tau_i \rangle,\ \mathrm{cat}_i \in \{\texttt{input}, \texttt{shared}\} \wedge \tau_i = \tau'_i$
$\forall i \in \{1, \ldots, n\} \quad \Gamma_{\mathsf{I}}(o_i) = \langle \mathrm{cat}_i, \tau_i \rangle,\ \mathrm{cat}_i \in \{\texttt{output}, \texttt{shared}\} \wedge \tau_i = \tau''_i$
$\mathcal{M}' = \langle \langle i_1{:}\tau'_1, \ldots; i_m{:}\tau'_m \rangle, \langle o_1{:}\tau''_1, \ldots; o_n{:}\tau''_n \rangle, Q, V, \Phi_T(T), \langle q_0, \Phi_A(\vec{a_0}) \rangle \rangle$
$\mu = \langle \mathcal{M}', q_0, \mathcal{I}(V) \rangle \Gamma_{\mathsf{M}}, \Gamma_{\mathsf{I}} \vdash \ \texttt{fsm}\ \mathrm{id}\ \langle i_1; \ldots; i_m \rangle\ \langle o_1; \ldots; o_n \rangle \rightarrow \mu$

Rule INST checks the arity and the type conformance of the inputs and outputs supplied to the instanciated model. The rule builds a *substitution* Φ for binding *local* input and output names to *global* ones. This substitution is applied to each transition (including the initial one) of the resulting automaton using the derived functions Φ_T and Φ_A (not detailed here). The $\mathcal{I}$ function builds an environment from a set of names, initializing each binding with the $\perp$ ("undefined") value :

$$\mathcal{I}(\{x_1, \ldots, x_n\}) = \{x_1 \mapsto \perp, \ldots, x_n \mapsto \perp\}$$

4.3 Dynamic semantic

The dynamic semantics of a CORE RFSM program will be given in terms of (instantaneous) **reactions**

$$\mathcal{C} \vdash \ M, \ \Gamma \xrightarrow[\rho]{\sigma} M', \ \Gamma'$$

meaning

"in the static context $\mathcal{C}$ and given a (dynamic) environment Γ, a set of automata M reacts to a *stimulus* σ leading to an updated set of automata M', an updated environment Γ' and producing a *response* ρ"

Definitions

▶ Given an expression `e` and an environment Γ, $\mathcal{E}_\Gamma[\![\mathtt{e}]\!]$ denotes the value obtained by **evaluating** expression `e` within environment Γ. For example

$$\mathcal{E}_{\{x \mapsto 1, y \mapsto 2\}}[\![x + y]\!] = 3$$

▶ An **event** e is either the occurrence of a *pure event* ϵ or the assignation of a **value** v to a **name** (input, output or local variable) :

$$e = \begin{cases} \epsilon \\ x \leftarrow v \end{cases}$$

▶ An **event set** E is a dated set of events

$$E = \langle t, \{e_1, \ldots, e_n\} \rangle$$

where t gives the occurrence **time** (logical instant).

For example $E = \langle 10, \{h, e \leftarrow 0\} \rangle$ means

"At time t=10, event `h` occurs and (input) e is set to 0" .

The union of *event sets* is defined as

$$\langle t, e \rangle \cup \langle t', e' \rangle = \begin{cases} \langle t, e \cup e' \rangle & \text{if } t = t' \\ \perp & \text{otherwise} \end{cases}$$

▶ A **stimulus** σ (resp. **response** ρ) is just an event set involving inputs (resp. outputs).

Rules

► Given a static description $\mathcal{H} = \langle M, C\rangle$ of a program, the **execution** of this program submitted to a sequence of stimuli $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ is formalized by rule EXEC

[Exec] $C \vdash \ M \rightarrow M_0,\ \Gamma_0$

$\forall i \in \{1, \ldots, n\} \qquad C \vdash \ M_{i-1},\ \Gamma_{i-1} \xrightarrow[\rho_i]{\sigma_i} M_i,\ \Gamma_i \mathcal{H} = \langle M, C\rangle \xrightarrow[\vec{\rho}=\langle\rho_1;\ldots;\rho_n\rangle]{\vec{\sigma}=\langle\sigma_1;\ldots;\sigma_n\rangle} M_n,\ \Gamma_n$

In other words, the execution of the program is described as as a sequence of **instantaneous reactions**, which can be denoted as[5] :

$$M_0,\ \Gamma_0 \xrightarrow[\rho_1]{\sigma_1} M_1,\ \Gamma_1 \rightarrow \ldots \xrightarrow[\rho_n]{\sigma_n} M_n,\ \Gamma_n$$

where

- the *global environment* Γ here records the value of inputs and shared variables[6],
- $\rho_1, \ldots, \rho_n$ is the sequence of responses,
- M_n and Γ_n respectively give the final state of the automata and global environment.

► Rule INIT describes how the initial set of automata M_0 and global environment Γ_0 are initialized by executing the initial transition of each automaton (producing a set of initial responses ρ_0.

[Init] $\forall i \in \{1, \ldots, n\} \quad \mu_i = \langle \mathcal{M}_i, q_i, \mathcal{V}_i\rangle \quad \mathcal{M}_i = \langle ., ., ., ., ., \langle ., \vec{a_i}\rangle\rangle \quad C \vdash \ \mathcal{V}_i,\ \gamma_{i-1} \xrightarrow[\langle 0,\emptyset\rangle]{\vec{a_i},0} \mathcal{V}'_i,\ \gamma_i \quad \mu'_i = \langle \mathcal{M}_i, q_i, \mathcal{V}'_i\rangle$

$C = \langle ., I_v, ., O_v, ., H_v\rangle \quad \gamma_0 = \mathcal{I}(I_v \cup O_v \cup H_v) C \vdash \ M = \{\mu_1, \ldots, \mu_n\} \rightarrow M_0 = \{\mu'_1, \ldots, \mu'_n\},\ \Gamma_0 = \gamma_n$

where the I_v, O_v and H_v sets, taken from the static context C, respectively give the name of inputs, outputs and shared variables.

Note. Rule INIT does *not* produce any response ρ. This is because the initial actions of an automaton cannot emit events hence can only update the its local environment or the global one.

► Rule ACTS describes how a sequence of actions $\vec{a}$ (at time t) updates the local and global environments, possibly emitting a set of responses[7].

[Acts] $\forall i \in \{1, \ldots, n\} \qquad C \vdash \ \mathcal{V}_{i-1},\ \Gamma_{i-1} \xrightarrow[\rho_i]{a_i,\ t} \mathcal{V}_i,\ \Gamma_i$

$\mathcal{V}_0 = \mathcal{V} \quad \Gamma_0 = \Gamma \quad \rho_e = \bigcup_{i=1}^{n} \rho_i C \vdash \ \mathcal{V},\ \Gamma \xrightarrow[\rho_e]{\langle a_1;\ldots;a_n\rangle,\ t} \mathcal{V}_n,\ \Gamma_n$

Note. The definition of rule ACTS given above enforces a *sequential interpretation* of actions. For example

$$\{x \mapsto 1,\ s \mapsto \bot\},\ \Gamma \xrightarrow{\langle x \leftarrow x+1; s \leftarrow x\rangle, t} \{x \mapsto 2,\ s \mapsto 2\}, \Gamma$$

Rule ACTS could easily be reformulated to describe other interpretations, such as a *synchronous* one, in which all RHS values are first evaluated and then assigned to LHS in parallel[8].

► Rules ACTUPDL and ACTUPDG respectively describe the effect of an action updating a local or global variable (shared variable or output)[9].

[5] Omitting context C, which is constant during an execution.

[6] This environment is required to handle events describing modifications of these values, as described below (see rule REACTUPD).

[7] This set of responses is always empty when rule ACTS is invoked in the context of INIT.

[8] As happens in hardware synchronous implementations for example.

[9] The effect of an action emitting an event will be described by rules ACTEMITS and ACTEMITG, given latter.

[ActUpdL] $\mathrm{x} \in \mathrm{dom}(\mathcal{V}) \quad v = \mathcal{E}_{\mathcal{V} \cup \Gamma}[\![\mathsf{e}]\!] C \vdash \ \mathcal{V},\ \Gamma \xrightarrow[\langle t, \emptyset \rangle]{x \leftarrow \mathsf{e},\ t} \mathcal{V}[x \mapsto v],\ \Gamma$ [ActUpdG] $\mathrm{x} \in \mathrm{dom}(\Gamma) \quad v = \mathcal{E}_{\mathcal{V} \cup \Gamma}[\![\mathsf{e}]\!] C \vdash \ \mathcal{V},\ \Gamma \xrightarrow[\langle t, \emptyset \rangle]{x \leftarrow \mathsf{e},\ t} \mathcal{V},\ \Gamma[x \mapsto$

▶ Rule React describes how a program M within a global environment Γ (instantaneously) reacts to a stimulus (event set) σ, producing a response (event set) ρ, an updated program M' and an updated environment Γ'.

[React] $\sigma_e, \sigma_v = \Sigma(\sigma) \qquad C \vdash \ M,\ \Gamma \xrightarrow{\sigma_v} M,\ \Gamma_v \qquad C \vdash \ M,\ \Gamma_v \xrightarrow[\rho_e]{\sigma_e} M',\ \Gamma' C \vdash \ M,\ \Gamma \xrightarrow[\rho_e]{\sigma} M',\ \Gamma'$

where the function Σ partitions a *event set* into one containing the stimuli corresponding to *pure* events (ϵ) and another containing those corresponding to updates to global inputs :

$$\Sigma(\langle t, \{e_1, \ldots, e_n\}\rangle) = \langle t, \{e_i \mid e_i = \epsilon_i\}\rangle, \quad \langle t, \{e_i \mid e_i = x_i \leftarrow v_i\}\rangle$$

▶ Rule ReactUpd describes how a program M within a global environment Γ reacts to set of events describing updates to global inputs. These updates are just recorded in the environment and do not produce responses, nor trigger any reaction of the automata :

[ReactUpd] $\mathrm{C} \vdash \ M,\ \Gamma \xrightarrow{\sigma_v = \langle t, \{x_1 \leftarrow v_1, \ldots, x_m \leftarrow v_m\}\rangle} M,\ \Gamma[x_1 \mapsto v_1] \ldots [x_m \mapsto v_m]$

▶ Rule ReactEv describes how a program reacts to a set of pure events.

[ReactEv] $\forall i \in \{1, \ldots, n\} \quad C \vdash \ \mu_{\pi(i)},\ \Gamma_{i-1} \xrightarrow[\rho_i]{\sigma_i} \mu'_{\pi(i)},\ \Gamma_i \qquad \sigma_i = \sigma_{i-1} \cup \rho_i$

$$\Gamma_0 = \Gamma \qquad \sigma_0 = \sigma_e \qquad \rho_e = \bigcup_{i=1}^{n} \rho_i \qquad \Gamma' = \Gamma_n C \vdash \ M = \{\mu_1, \ldots, \mu_n\}, \Gamma \xrightarrow[\rho]{\sigma_e = \langle t, \{\epsilon_1, \ldots, \epsilon_m\}\rangle}_e M' = \{\mu'_1, \ldots, \mu'_n\},\ \Gamma'$$

Each automaton reacts separately but *in a specific order*. This order is derived from the dependencies between automata. We say that an automaton μ' *depends on* another automaton μ at a given instant t, and note

$$\mu \leq \mu'$$

if the reaction of μ at this instant can trigger or modify the reaction of μ' at the same instant. Concretely, this happens when μ and μ' are respectively in states q and q' and there's (at least) one pair of transitions (τ, τ') starting respectively from q and q' so that

- τ' is triggered by an event emitted by τ, or
- a variable occuring in the guards associated to τ' is written by the actions associated to τ.

The function π used in ReactEv is a permutation of $\{1, \ldots, n\}$ defined so that

$$\mu_{\pi(1)} \leq \mu_{\pi(2)} \leq \cdots \leq \mu_{\pi(n)}$$

Having the automata of M react in the order $\pi(1), \ldots, \pi(n)$ ensures that any event emitted or local variable update performed by an automaton during a given reaction is effectively perceived by any other automaton *at the same reaction*, a principle called *instantaneous broadcats.*

The permutation π can easily be computed by a *topological sort* of the dependency graph derived from the conditions expressed above. In practice, this will be carried out by a static analysis of the program.

▶ Rules React1, React0 and ReactN describe how a single automaton reacts to a set of pure events, updating both its internal and global states and producing another set of (pure) events in response.

[React1] $\Delta_{\Gamma\cup\mathcal{V}}(\mathcal{M},q,e)=\{\tau\}$
$C\vdash\ \mu,\ \Gamma \xrightarrow[\rho_e]{\tau,\ t} \mu', \Gamma' C\vdash\ \mu=\langle\mathcal{M},q,\mathcal{V}\rangle,\ \Gamma \xrightarrow[\rho_e]{\sigma_e=\langle t,e\rangle} \mu',\ \Gamma'$

[React0] $\Delta_{\Gamma\cup\mathcal{V}}(\mathcal{M},q,e)=\emptyset C\vdash\ \mu=\langle\mathcal{M},q,\mathcal{V}\rangle,\ \Gamma \xrightarrow[\langle t,\emptyset\rangle]{\sigma_e=\langle t,e\rangle} \mu,\ \Gamma$

[ReactN] $\Delta_{\Gamma\cup\mathcal{V}}(\mathcal{M},q,e)=\{\tau_1,\ldots,\tau_n\}$
$\tau=\mathsf{choice}(\{\tau_1,\ldots,\tau_n\})$
$C\vdash\ \mu,\ \Gamma \xrightarrow[\rho_e]{\tau,\ t} \mu', \Gamma' C\vdash\ \mu=\langle\mathcal{M},q,\mathcal{V}\rangle,\ \Gamma \xrightarrow[\rho_e]{\sigma_e=\langle t,e\rangle} \mu',\ \Gamma'$

Given a automaton modelised by $\mathcal{M}$ and currently in state q, $\Delta_\Gamma(\mathcal{M},q,e)$, where $e=\{\epsilon_1,\ldots,\epsilon_n\}$, returns the set of *fireable* transitions, *i.e.* all the transitions triggered by the event set e starting from q and for which the all the associated boolean guards evaluate, in environment Γ, to true.

$$\Delta_\Gamma(\mathcal{M},q,\{\epsilon_1,\ldots,\epsilon_n\})=\bigcup_{i=1}^{n}\Delta_\Gamma(\mathcal{M},q,\epsilon_i)$$

where

$$\Delta_\Gamma(\langle .,.,.,.,T,.\rangle,q,\epsilon)=\{(q_s,c,a,q_d)\in T \mid q=q_s\wedge c=\langle\epsilon,\{e_1,\ldots,e_n\}\rangle\wedge\forall i\in\{1,\ldots,n\}\quad \mathcal{E}_\Gamma[\![e_i]\!]=\mathsf{true}\}$$

Rule React1 describes the case when the set of events triggers exactly *one* transition of the automaton. Its state and local variables are updated according to the actions listed in the transition and the remaining actions are used to generated the set of responses.

Rule React0 describes the case when the set of events does not trigger any transition. The automaton and the global environment are left unchanged.

Rule ReactN describes the case when the set of events triggers more than one transition. This situation corresponds to a non-deterministic behavior of the automaton. The function choice is here used to choose one transition[10].

▶ Rule Trans describes the effect of performing a transition, updating the automaton local and global states and returning a set of (pure) events as responses.

[Trans] $\mu=\langle\mathcal{M},q,\mathcal{V}\rangle \qquad \tau=\langle q,c,\vec{a},q'\rangle$
$C\vdash\ \mathcal{V},\ \Gamma \xrightarrow[\rho_e]{\vec{a},t}_a \mathcal{V}',\ \Gamma'$
$\mu'=\langle\mathcal{M},q',\mathcal{V}'\rangle C\vdash\ \mu,\ \Gamma \xrightarrow[\rho_e]{\tau,\ t} \mu',\ \Gamma'$

▶ Rules ActEmitS and ActEmitG complement the rules ActUpdL and ActUpdG given previously by describing the effect of an action emitting a shared or output event. The H_e set, taken from context C, is here used to distinguish between to to. The formers can trigger the reaction of other(s) automaton(s), the latters are just ignored here (see note below).

[ActEmitS] C=$\langle .,.,.,.,H_e,.\rangle \quad \epsilon\in H_e C\vdash\ \mathcal{V},\ \Gamma \xrightarrow[\langle t,\{\epsilon\}\rangle]{\epsilon,\ t} \mathcal{V},\ \Gamma$

[ActEmitG] C=$\langle .,.,.,.,H_e,.\rangle \quad \epsilon\notin H_e C\vdash\ \mathcal{V},\ \Gamma \xrightarrow[\langle t,\emptyset\rangle]{\epsilon,\ t} \mathcal{V},\ \Gamma$

Note. The semantics described here only defines how a program execution progresses, from the initial program to the final program state M. In practice, an interpreter will also build a *trace* of such an execution, recording all significant events (stimuli, responses, state moves, *etc.*). Building such a trace is easily performed by modifying the semantic rules given above. It has not been done here for the sake of simplicity.

[10] This can done, for example, by adding a *priority* to each transition.

Chapter 5

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. 5.6.

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

5.1 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`.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 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 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.

$$
\begin{array}{rcl}
\langle\text{program}\rangle & ::= & \langle\text{decl}\rangle^{*} \\[1ex]
\langle\text{decl}\rangle & ::= & \langle\text{type\_decl}\rangle \\
& | & \langle\text{cst\_decl}\rangle \\
& | & \langle\text{fn\_decl}\rangle \\
& | & \langle\text{fsm\_model}\rangle \\
& | & \langle\text{fsm\_inst}\rangle \\
& | & \langle\text{global}\rangle \\[1ex]
\langle\text{type\_decl}\rangle & ::= & \textbf{type}\ \text{lid} = \langle\text{type\_expr}\rangle \\
& | & \textbf{type}\ \text{lid} = \textbf{enum}\ \{\ \text{uid}^{*}_{,}\ \} \\
& | & \textbf{type}\ \text{lid} = \textbf{record}\ \{\ \langle\text{record\_field}\rangle^{+}_{,}\ \} \\[1ex]
\langle\text{record\_field}\rangle & ::= & \text{lid}\ \texttt{:}\ \langle\text{type\_expr}\rangle \\[1ex]
\langle\text{cst\_decl}\rangle & ::= & \textbf{constant}\ \text{lid}\ \texttt{:}\ \langle\text{fres}\rangle = \langle\text{const}\rangle \\[1ex]
\langle\text{fn\_decl}\rangle & ::= & \textbf{function}\ \text{lid}\ \texttt{(}\ \langle\text{farg}\rangle^{*}_{,}\ \texttt{)}\ \texttt{:}\ \langle\text{fres}\rangle\ \{\ \textbf{return}\ \langle\text{fbody}\rangle\ \} \\[1ex]
\langle\text{farg}\rangle & ::= & \text{lid}\ \texttt{:}\ \langle\text{type\_expr}\rangle \\[1ex]
\langle\text{fres}\rangle & ::= & \langle\text{type\_expr}\rangle \\[1ex]
\langle\text{fbody}\rangle & ::= & \langle\text{expr}\rangle \\[1ex]
\langle\text{fsm\_model}\rangle & ::= & \textbf{fsm model}\ \langle\text{id}\rangle\ [\langle\text{params}\rangle]\ \texttt{(}\ \langle\text{io}\rangle^{*}_{,}\ \texttt{)}\ \{ \\
& & \textbf{states}\ \texttt{:}\ \langle\text{state}\rangle^{*}_{,}\ \texttt{;} \\
& & [\langle\text{vars}\rangle] \\
& & \textbf{trans}\ \texttt{:}\ \langle\text{transition}\rangle^{*}\ \texttt{;} \\
& & \textbf{itrans}\ \texttt{:}\ \langle\text{itransition}\rangle\ \texttt{;} \\
& & \} \\[1ex]
\langle\text{state}\rangle & ::= & \text{uid}\ [\textbf{where}\ \langle\text{oval}\rangle^{+}_{\textbf{and}}] \\[1ex]
\langle\text{oval}\rangle & ::= & \text{lid} = \langle\text{constant}\rangle \\[1ex]
\langle\text{params}\rangle & ::= & \texttt{<}\ \langle\text{param}\rangle^{*}_{,}\ \texttt{>} \\[1ex]
\langle\text{param}\rangle & ::= & \text{lid}\ \texttt{:}\ \langle\text{type\_expr}\rangle \\[1ex]
\langle\text{io}\rangle & ::= & \textbf{in}\ \langle\text{io\_desc}\rangle \\
& | & \textbf{out}\ \langle\text{io\_desc}\rangle \\
& | & \textbf{inout}\ \langle\text{io\_desc}\rangle \\[1ex]
\langle\text{io\_desc}\rangle & ::= & \text{lid}\ \texttt{:}\ \langle\text{type\_expr}\rangle \\[1ex]
\langle\text{vars}\rangle & ::= & \textbf{vars}\ \texttt{:}\ \langle\text{var}\rangle^{*}_{,}\ \texttt{;} \\[1ex]
\langle\text{var}\rangle & ::= & \text{lid}^{+}_{,}\ \texttt{:}\ \langle\text{type\_expr}\rangle
\end{array}
$$

⟨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⟩ ::= -
                   |  -.
```

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

Appendix B - Compiler options

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

-main	set prefix for the generated main files
-dump_typed	dump typed representation of model(s)/program to stdout
-dump_static	dump static representation of model(s)/program to stdout
-target_dir	set target directory (default: .)
-lib	set location of the support library (default: ¡opam_prefix¿/share/rfsm)
-dot	generate .dot representation of model(s)/program
-sim	run simulation (generating .vcd file)
-ctask	generate CTask code
-systemc	generate SystemC code
-vhdl	generate VHDL code
-version	print version of the compiler and quit
-show_models	generate separate representations for uninstanciated FSM models
-dot_qual_ids	print qualified identifiers in DOT representations
-dot_no_captions	Remove captions in .dot representation(s)
-dot_short_trans	Print single-line transition labels (default is multi-lines)
-dot_abbrev_types	Print abbreviated types (default is to print definitions)
-sim_trace	set trace level for simulation (default: 0)
-vcd_int_size	set default int size for VCD traces (default: 8)
-synchronous_actions	interpret actions synchronously
-normalize	move output assignations from states to transitions
-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

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 bool e;
 out bool s;
  )
{
  int k;
  enum {E0,E1} state = E0;
  while ( 1 ) {
    switch ( state ) {
    case E1:
      s = true;
      wait_ev(h);
      if ( k<4 ) {
        k=k+1;
        }
      else if ( k==4 ) {
        state = E0;
        }
      break;
    case E0:
      s = false;
      wait_ev(h);
      if ( e==true ) {
        k=1;
        state = E1;
        }
      break;
    }
  }
};
```

Appendix C1 - Example of generated SystemC code

This is the code generated from program given in Listing 2.1

Listing 5.1: 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 5.2: File g.cpp

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

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

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

Listing 5.3: 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 5.4: 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 5.5: 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 5.6: 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 5.7: 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 5.8: File g.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.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: unsigned(2 downto 0);
  begin
    if ( rst='1' ) then
      state <= E0;
    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
          state <= E0;
        end if;
      when E0 =>
        if ( e = '1' ) then
          k := to_unsigned(1,3);
          state <= E1;
        end if;
    end case;
```

```
    end if;
  end process;
  process(state)
  begin
    case state is
    when E0 =>
      s <= '0';
    when E1 =>
      s <= '1';
    end case;
  end process;
end architecture;
```

Listing 5.9: 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;
```