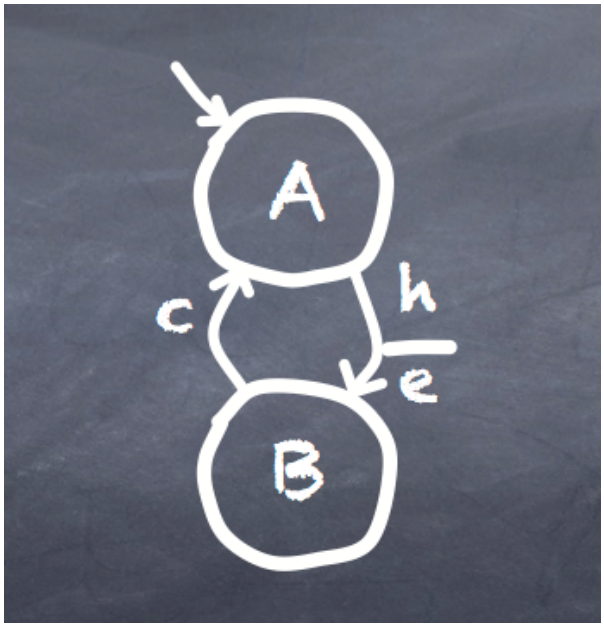
RFSM User Manual - 1.0

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 3 describes how to use the command-line compiler. Chapter 4 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 (line 1) introduces a **type declaration**. The type `bit` is defined as a synonym of type `int<0..1>`, *i.e.* the type of integers with values between 0 and 1.

The second part (lines 3–15) 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 `bit` respectively, and one output `s` of type `bit`. 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 `--` *condition* `|` *actions* `->` *destination state*

where *condition* is the condition trigerring the transition and *actions* is a list of actions performed when then transition is enabled. The semantics is that the transition is enabled whenever the FSM is in the source state and the triggering condition is 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 actions* `->` *initial state*

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).

Note that, at this level, the value of the parameter `n`, used in the type of the internal variable `k` (line 9) and in the transition conditions (lines 12 and 13) is left unspecified, making the `gensig` model a *generic* one.

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

Listing 2.1: A simple RFSM program

```
1  type bit = int<0..1>
2
3  fsm model gensig<n:int> (
4    in h: event,
5    in  e: bit,
6    out s: bit)
7    {
8    states: E0, E1;
9    vars: k: int<0..n>;
10   trans:
11     E0 -- h.e=1 | k:=1; s:=1 -> E1,
12     E1 -- h.k<n | k:=k+1 -> E1,
13     E1 -- h.k=n | s:=0 -> E0;
14   itrans: | s:=0 -> E0;
15   }
16
17 input H : event = periodic (10,0,80)
18 input E : bit = value_changes (0:0, 25:1, 35:0)
19 output S : bit
20
21 fsm g4 = gensig<4>(H,E,S)
```

The third part of the program (lines 17–19) 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 last part of the program (line 21) consists in building the global model of the system by *instanciating* the FSM models. 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].

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 4 resp.). It produces

[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

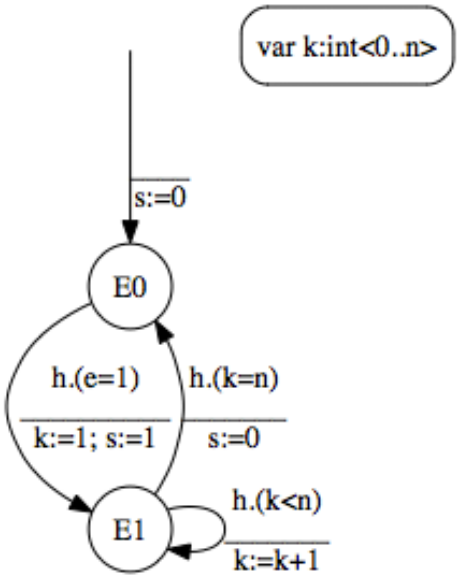


Figure 2.1: A graphical representation of FSM model defined in Listing 2.1

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.

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.

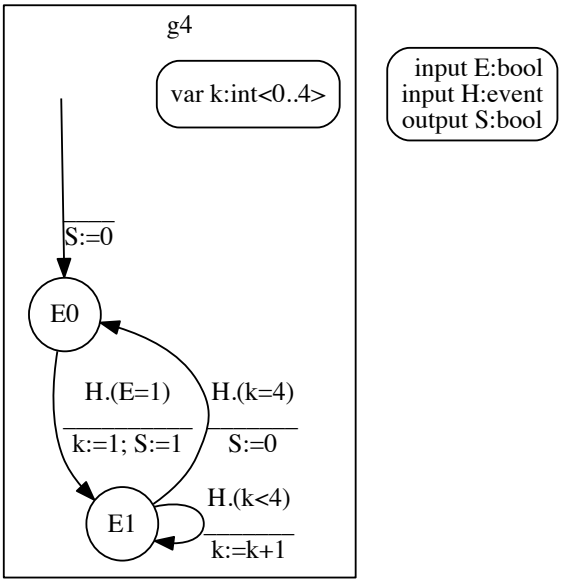


Figure 2.2: A graphical representation of system described in Listing 2.1

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.

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` and `event`.

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

▶ The type `int` can be refined using a *range annotation*. The type `int<lo..hi>` designates integer values ranging from `lo` to `hi`. The range limits `lo` and `hi` 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).

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

▶ Type abbreviations are introduced with the following declaration

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

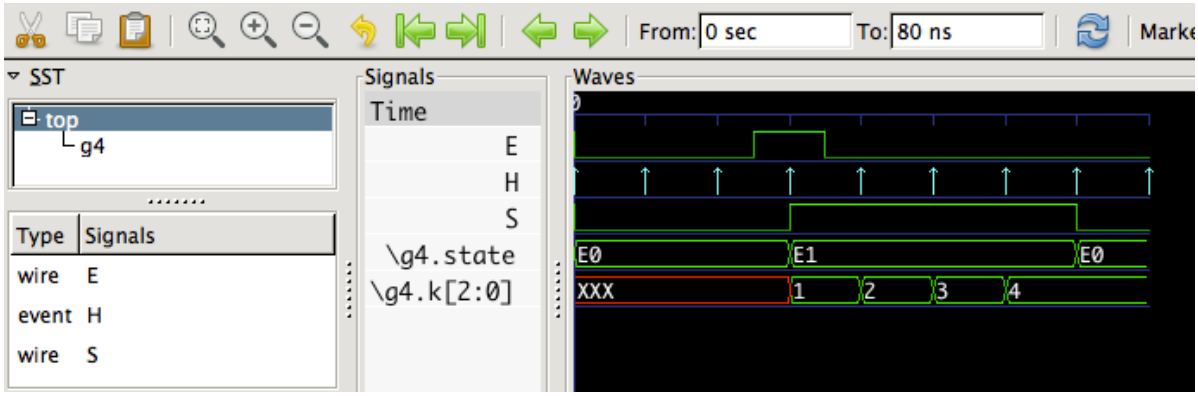


Figure 2.3: Simulation results for the program in Listing 2.1, viewed using `gtkwave`

type typename = type_expression

Each occurrence of the defined type in the program is actually substituted by the corresponding type expression. Type expressions in type abbreviations are currently limited to builtin types. An example of type abbreviation has been given in the program of Listing 2.1.

Enumerated types are introduced with the following declaration

type typename = { 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 }

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 −− condition | actions −> dst_state

where

- *src_state* and *dst_state* respectively designates the source state and destination state,
- *condition* is the condition trigerring the transition,
- *actions* is a list of actions performed when then transition is enabled.

The semantics is that the transition is enabled whenever the FSM is in the source state and the triggering condition is true. The associated actions are then performed and the FSM moves to the destination state.

A **condition** must involve exactly one *triggering event* and, possibly, a conjunction of boolean conditions called *guards*. The triggering event must be listed in the inputs. The guards may involve inputs and/or internal variables.

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

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

Table 2.1: Operations on types

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 2.1.

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

Examples :

S0 −− top −> S1

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

Idle −− Clic | ctr:=0; Received −> Wait

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 −− Top.ctr<8 | ctr:=ctr+1 −> Wait

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 −− H | k:=k<8?k+1:0 −> S0

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

src_state −− condition −> dst_state

Initial transition

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

| actions −> init_state

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 last section of an RFSM program constructs the description of the system by instanciating – and, possibly, inter-connecting – the previously 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 g4 = 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

```
1  fsm model cntmod2 (
2    in h: event,
3    out s: int<0..1>,
4    out r: event)
5    {
6    states: E0, E1;
7    trans:
8      E0 -- h | s:=1 -> E1,
9      E1 -- h | r; s:=0 -> E0;
10   itrans: | s:=0 -> E0;
11   }
12
13 input H: event = periodic(10,10,100)
14 output S0: int<0..1>
15 output S1: int<0..1>
16 output S2: int<0..1>
17 output R2: event
18
19 shared R0: event
20 shared R1: event
21
22 fsm C0 = cntmod2(H,S0,R0)
23 fsm C1 = cntmod2(R0,S1,R1)
24 fsm C2 = cntmod2(R1,S2,R2)
```

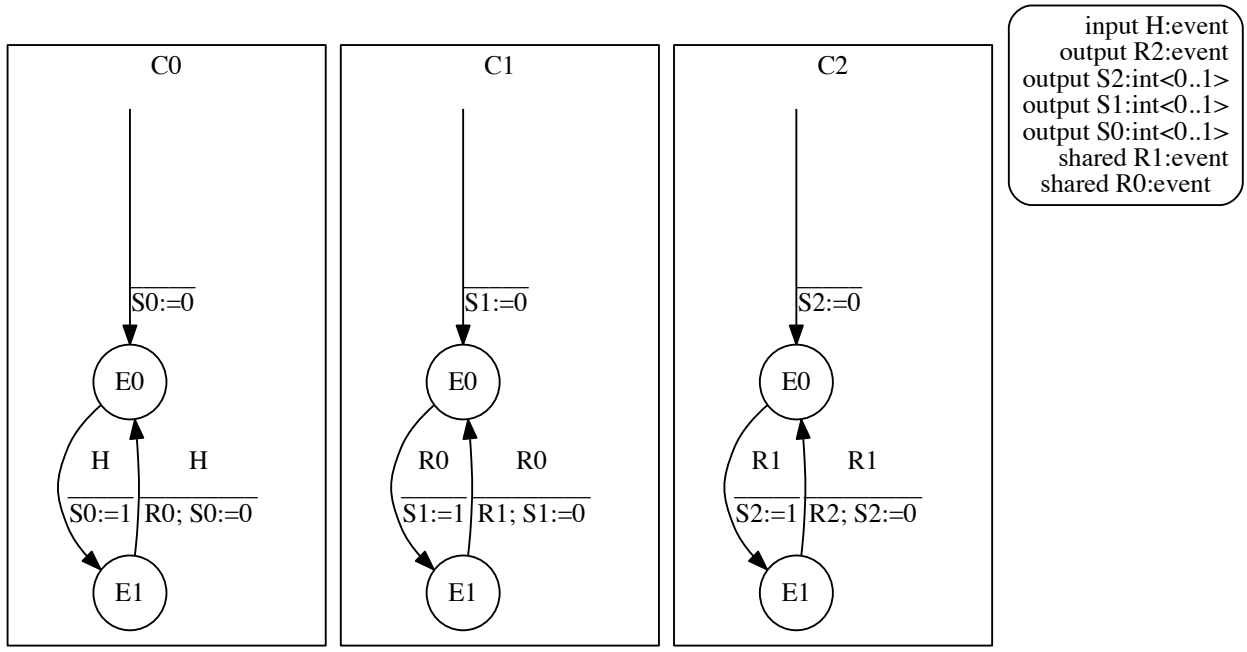


Figure 2.4: A graphical representation of program described in Listing 2.2

2.2.5 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[8].

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 2.3. 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 c1: non deterministic transitions found at t=70:
- Running--h|ctr:=ctr+1; aff:=ctr->Running[0]
- Running--startstop->Stopped[0]
```

[8]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.

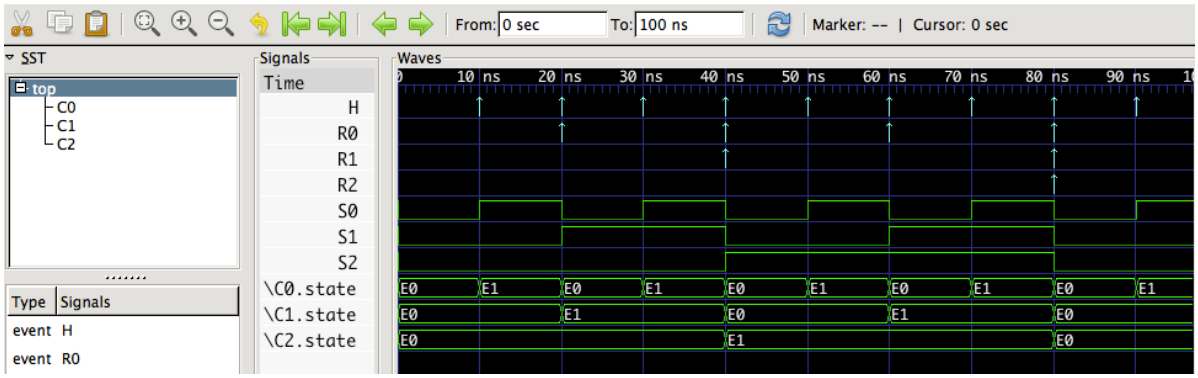


Figure 2.5: Simulation results for the program in Listing 2.2

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[9]. When several transitions are enabled, if one is tagged as "high priority" than it is automatically selected[10].

Syntaxically, tagging a transition is simply achieved by prefixing it with a "`*`". In the case of the example above, the modified program is given in Listing 2.4. 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 −− H | x:=x+1; y:=x∗2 −−> S1

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 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.

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

Listing 2.3: A program showing a potentially non-deterministic model

```
1  fsm model chrono (
2       in sec: event,
3       in startstop: event,
4      out aff: int)
5    {
6    states: Stopped, Running;
7    vars: ctr: int;
8    trans:
9      Stopped -- startstop | ctr:=0; aff:=0 -> Running,
10     Running -- sec | ctr:=ctr+1; aff:=ctr -> Running,
11     Running -- startstop -> Stopped;
12   itrans: -> Stopped;
13   }
14
15 input StartStop: event = sporadic(25,70)
16 input H:event = periodic(10,10,110)
17 output Aff: int
18
19 fsm c1 = chrono(H, StartStop, Aff)
```

Listing 2.4: A rewriting of the model defined in Listing 2.3

```
1  fsm model chrono (...)
2    {
3    ...
4    trans:
5      ...
6      Running -- sec | ctr:=ctr+1; aff:=ctr -> Running,
7      *Running -- startstop -> Stopped;
8    itrans: -> Stopped;
9    }
10 ...
```

By default, the `rfsmc` compiler relies on a sequential interpretation, both for simulation and code production[11]. 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[12].

Note. As a syntactic reminder, list of actions are printed in diagrams using "`;`" as a separator when using a sequential interpretation and using "`,`" when using a synchronous interpretation.

[11]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.

[12]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 programs (using the `.dot` format),
- simulate programs, generating execution traces (`.vcd` format),
- generate C, SystemC or VHDL code from 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] file
```

where `file` is the name of the file containing the source code (by convention, this file should be suffixed `.fsm`).

The complete set of options is described in App. 4.

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 a graphical representation of the program

```
rfsmc -dot foo.fsm
```

The previous command generates graphical representations of the program contained in file `foo.cph` in `.dot` format. This representation can be viewed with the `Graphviz` suite of tools[1].

By default, the command generates a single file `foo_top.dot` containing the top level representation of the system (with one "box" for each FSM instance). By passing the `-dot_fsm_insts` option (resp. `-dot_fsm_models`), it is possible to obtain separate `.dot` files for each FSM instance (resp. model).

3.2 Running the simulator

```
rfsmc -sim foo.fsm
```

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

The previous command runs simulator on the program contained in file `foo.fsm`, writing an execution trace in VCD (Value Change Dump) format in file `run.vcd`.

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

The name of the `vcd.file` can be changed using the `-vcd` option.

3.3 Generating C code

```
rsfmc -ctask foo.fsm
```

For each FSM instance `f` contained in file `foo.fsm`, the previous command generates a file `f.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

The minimal command for invoking the SystemC backend is :

```
rsfmc -systemc foo.fsm
```

This will generate the SystemC code corresponding the program contained in file `foo.fsm`, *i.e.* write the following files :

- for each FSM instance `f`, a pair of files `f.h` and `f.cpp` containing the interface and implementation of the SystemC module describing this instance,
- for each 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 `tb.cpp` containing the description of the *testbench* corresponding to the program for simulation.

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 list of *Makefile* targets to be appended to a predefined *Makefile* so that compiling and running the code generated by the SystemC backend can be performed by simply invoking `make` on this *Makefile*. For this, the RFSM compiler simply needs to know where this predefined *Makefile* has been installed. 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

```
rsfmc -systemc -lib /usr/local/rfsm/lib -target_dir ./systemc  foo.fsm
```

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. Of course, you may have to adjust some definitions in the file `.../lib/etc/Makefile.systemc` to reflect the specifities of your local SystemC installation.

[2] gtkwave.sourceforge.net

3.5 Generating VHDL code

The minimal command for invoking the VHDL backend is :

```
rsfmc -vhdl foo.fsm
```

This will generate the VHDL code corresponding the program contained in file `foo.fsm`, *i.e.* write the following files :

- for each FSM instance `f`, a file `f.vhd` containing the entity and architecture describing this instance,
- a file `tb.vhd` containing the description of the *testbench* corresponding to the program for simulation.

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

Concerning simulation, and as for the SystemC backend, the process can be grealy simplified by using a pair of *Makefile*s, one predefined and the other generated by the compiler. 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  foo.fsm
```

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
```

Note. As for the SystemC backend, for this to work, you may have to adjust some definitions in the file

3.6 Using `make`

The current distribution provides, in `.../lib/etc` directory, a file `Makefile.app` aiming at easing the invokation of the RFSM compiler and the exploitation of the generated products.

Basically, if this file has been installed, let say in directory `/usr/local/rfsm/lib/etc` and if you create the following Makefile in the working directory (that containing the RFSM source file)

```
APP= # to be set to the name of the source file (ex: foo)
DOT_OPTS= # to be adjusted if necessary
SIM_OPTS= # to be adjusted if necessary
SYSTEMC_OPTS= # to be adjusted if necessary
VHDL_OPTS= # to be adjusted if necessary
include /usr/local/rfsm/lib/etc/Makefile.app
```

then, for example, 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),

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

[4] Please refer to the file `Makefile.app` itself for a complete list of targets.

- `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).

Chapter 4

The Graphical User Interface

This chapter describes the RFSM IDE[1]. This IDE basically provides a Graphical user Interface (GUI) to the `rfsmc` compiler described in chapter 3.

The GUI allows

- writing, reading and editing of RFSM programs,
- generating and viewing graphical representations of these programs,
- running simulations,
- generating C, SystemC and VHDL code.

Note. This chapter supposes that the IDE has been correctly installed. If not, refer to the installation guide provided in the RFSM distribution.

First, **launch the RFSM application** by clicking on its icon in the installation directory or directly from the Windows *Start* menu.

The application main window is shown in Fig. 4.1. The main elements are (with corresponding areas labeled in red in Fig. 4.1) :

1. a menubar
2. four buttons for file manipulation; from left to right
   - create a new file,
   - open an existing file,
   - save a file,
   - save all files.
3. five buttons to invoke the compiler for (from left to right)
   - generating graphical representations of the current program and visualize it,
   - simulating the current program and visualize it,
   - generating C code from the current program,

[1]Screenshots used in this chapter show the Windows version of the RFSM IDE. The IDE can also be built and used on Unix-based systems (Linux, MacOS).

- generating SystemC code from the current program,
- generating VHDL code from the current program (button VHDL).

4. a tab for viewing and editing input source files,
5. a tab for viewing output files,
6. a log area, displaying issued command and outputs from the compiler.

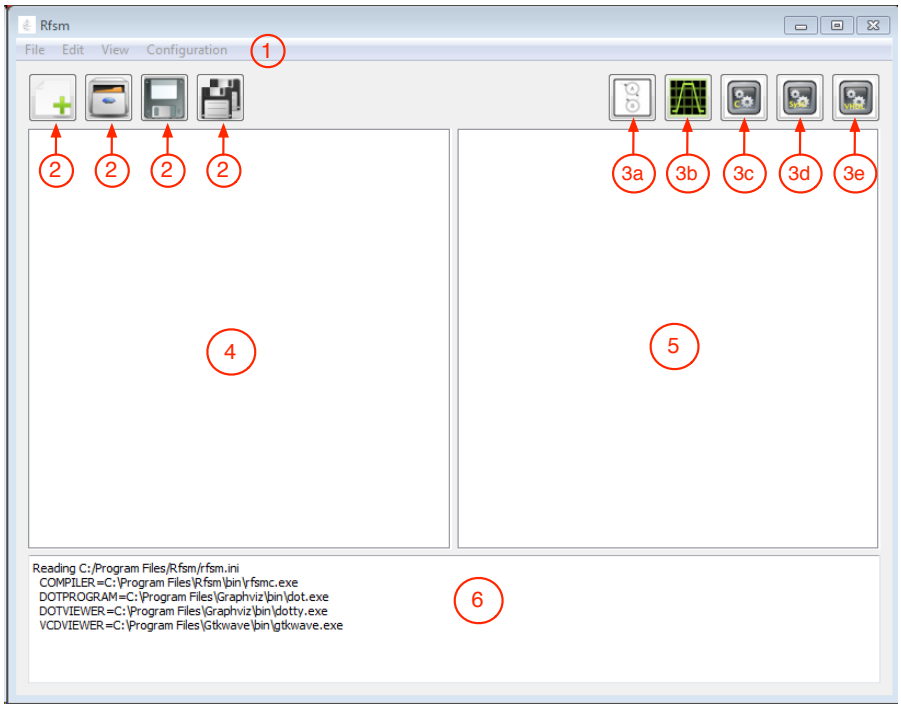


Figure 4.1: Main window of the RFSM application

Invoke the [Configuration:Compiler and Tools] menu item and check that the specified paths are right (see Fig. 4.2). They should respectively point to

- the location of the `rfsmc` compiler (`<install>/bin/rfsmc`, where `<install>` is the RFSM installation directory, as specified during the installation process),
- the location of the program to invoke for processing `.dot` files,
- the location of the program to invoke for viewing `.dot` files,
- the location of the program to invoke for viewing `.vcd` traces.

If the specified paths are not correct[2], adjust them and click Ok.

Create a new source file or **open an existing file** by clicking on the New file (resp. Open File) button or invoking the corresponding item of the File menu. A new tab will appear, either blank or

[2]This may be the case, for example, if you have changed the program to view graphs and/or images since RFSM was installed.

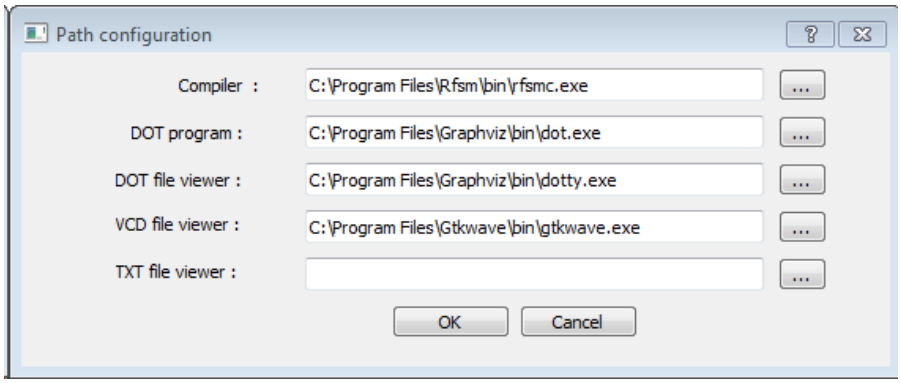


Figure 4.2: Path configuration window

containing the text of the opened file. This file can be freely edited and saved. Fig. 4.3 shows the GUI after opening a source file (the opened file is located in directory `examples/single/mousectlr` of the distribution).

To **generate the graphical representation of the program**, click on the Graph button (numbered 3a in Fig. 21). This will

- invoke the `rfsmc` compiler with the adequate option(s),
- generate the `.dot` result file (in the same directory as the source file),
- view this result by invoking the graph visualisation program specified in [Configuration : Compiler and Tools] window.

The result is displayed in Fig. 4.4.

For **simulating the program**, invoke the compiler by clicking on the SIM button (numbered 2 in Fig. 21). This will run the program, generate results in the file `run.vcd` and launch the VCD viewer specified in [Configuration : Compiler and Tools] window. The result is displayed in Fig. 4.5.

For **generating the C, SystemC or VHDL** code, click on the corresponding buttons (numbered 3c, 3d and 3e in Fig. 4.1). The result files will be generated in sub-directories named `./ctask`, `./systemc` and `./vhdl` and displayed as separate tabs on the right, as illustrated in Fig. 4.6, for example.

Options to be passed to RFSM compiler can be set and inspected by invoking Compilater options item of the Configuration menu, as illustrated in Fig. 4.7. These options are documented in Appendix B.

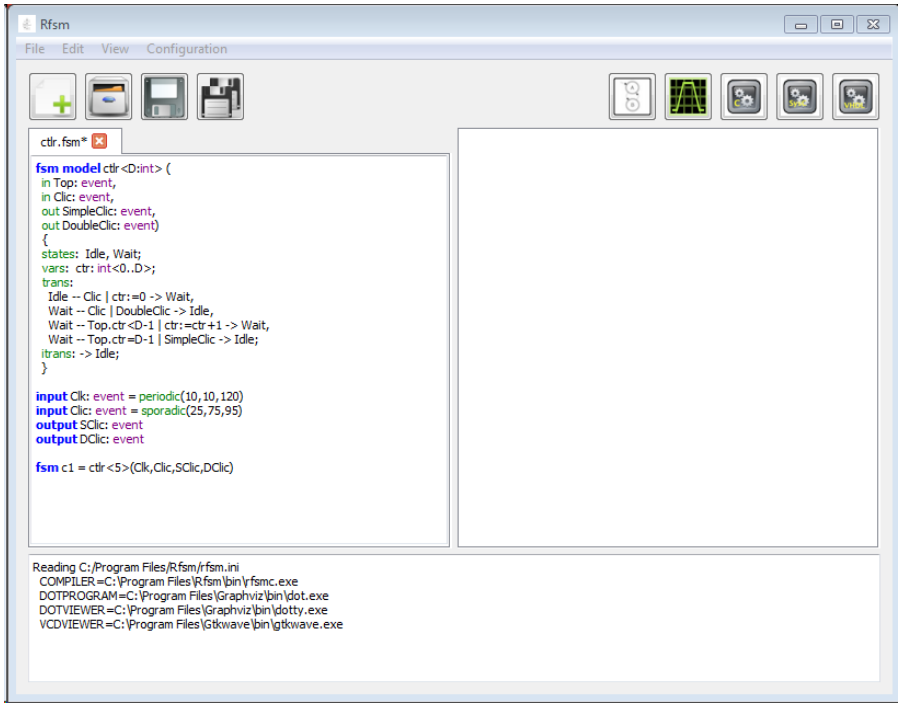


Figure 4.3: Editing source program

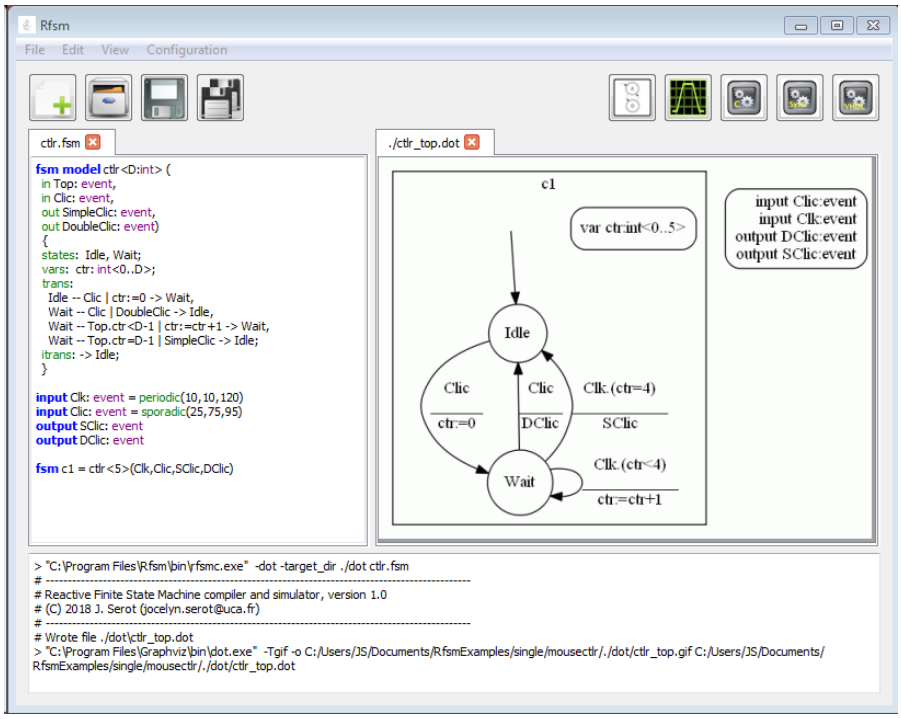


Figure 4.4: Viewing the graphical representation of the program

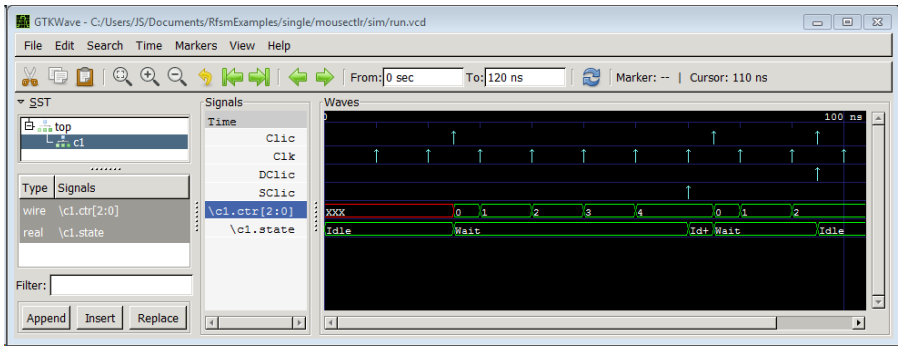


Figure 4.5: Viewing the simulation result

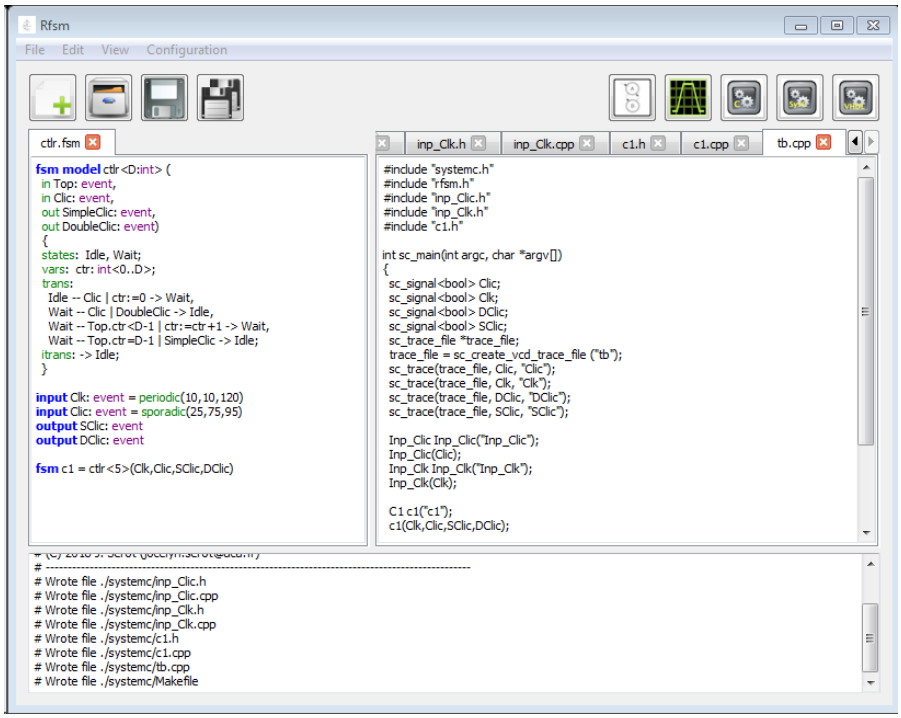


Figure 4.6: After generating SystemC code

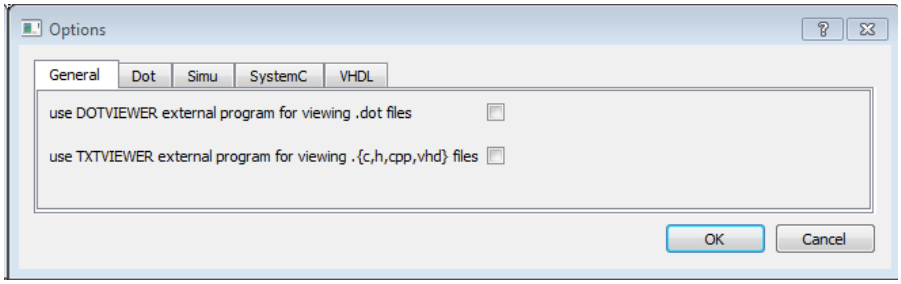


Figure 4.7: The options setting dialog

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 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. Terminal `int` designates a positive or nul integer.

⟨program⟩ ::= ⟨type_decl⟩* ⟨fsm_model⟩+ ⟨global⟩+ ⟨fsm_inst⟩+ EOF

⟨type_decl⟩ ::= `type` lid `=` ⟨typ⟩
| `type` lid `=` `{` uid$^{*}_{,}$ `}`

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

⟨params⟩ ::= `<` ⟨param⟩$^{*}_{,}$ `>`

⟨param⟩ ::= lid `:` ⟨typ⟩

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

⟨io_desc⟩ ::= lid `:` ⟨typ⟩

⟨vars⟩ ::= `vars` `:` ⟨var⟩$^{*}_{,}$ `;`

⟨var⟩ ::= lid `:` ⟨typ⟩

⟨transition⟩ ::= [`*`] uid `--` ⟨condition⟩ [⟨actions⟩] `->` uid

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

⟨condition⟩ ::= lid
| lid `.` ⟨guard⟩$^{+}_{.}$

⟨guard⟩ ::= ⟨guard_expr⟩

⟨actions⟩ ::= `|` ⟨action⟩$^{+}_{;}$

⟨action⟩ ::= lid
| lid `:=` ⟨expr⟩

⟨global⟩ ::= `input` ⟨id⟩ `:` ⟨typ⟩ `=` ⟨stimuli⟩
| `output` ⟨id⟩ `:` ⟨typ⟩
| `shared` ⟨id⟩ `:` ⟨typ⟩

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

⟨value_change⟩ ::= int `:` ⟨const⟩

⟨fsm_inst⟩ ::= `fsm` ⟨id⟩ `=` ⟨id⟩ [`<` $\text{int}^{+}_{,}$ `>`] `(` $\langle\text{id}\rangle^{*}_{,}$ `)`

⟨typ⟩ ::= `event`
| `int` [⟨int_range⟩]
| `bool`
| lid

⟨int_range⟩ ::= `<` ⟨type_index_expr⟩ `..` ⟨type_index_expr⟩ `>`

⟨type_index_expr⟩ ::= int
| 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⟩

⟨guard_expr⟩ ::= ⟨expr⟩ `=` ⟨expr⟩
| ⟨expr⟩ `!=` ⟨expr⟩
| ⟨expr⟩ `>` ⟨expr⟩
| ⟨expr⟩ `<` ⟨expr⟩
| ⟨expr⟩ `>=` ⟨expr⟩
| ⟨expr⟩ `<=` ⟨expr⟩

⟨expr⟩ ::= int
| ⟨bool⟩
| lid
| uid
| `(` ⟨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⟩

⟨const⟩ ::= int
| ⟨bool⟩
| uid

⟨bool⟩ ::= true
| false

$$
\begin{array}{lcl}
\langle \text{id} \rangle & ::= & \text{lid} \\
& | & \text{uid}
\end{array}
$$

Appendix B - Compiler options

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

-lib	set location of the support library (default: /usr/local/rfsm/lib)
-dump_model	dump model in text format to stdout
-target_dir	set target directory (default: .)
-dot	generate top-level .dot representation(s)
-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
-dot_no_captions	Remove captions in .dot representation(s)
-dot_fsm_insts	generate .dot representation of all FSM instances
-dot_fsm_models	generate .dot representation of all FSM models
-dot_actions_nl	write actions with with a separating newline
-trace	set trace level for simulation (default: 0)
-vcd	set name of .vcd output file when running simulation (default: run.vcd)
-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)
-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)

Appendix C1 - Example of generated C code

This is the code generated from program given in Listing 2.1

```
task g4(
  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 4.1: File g4.h

```
#include "systemc.h"

SC_MODULE(G4)
{
  // 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(G4) {
    SC_THREAD(react);
    }
};
```

Listing 4.2: File g4.cpp

```
#include "g4.h"
#include "rfsm.h"

void G4::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 4.3: File inp_H.h

```
#include "systemc.h"

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

  void gen();

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

Listing 4.4: File inp_H.cpp

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

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

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

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

```
    wait(_clk.period, SC_NS);
    notify_ev(H,"H");
    _t += _clk.period;
    }
};
```

Listing 4.5: File inp_E.h

```
#include "systemc.h"

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

  void gen();

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

Listing 4.6: File inp_E.cpp

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

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

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

Listing 4.7: File tb.cpp

```
#include "systemc.h"
#include "rfsm.h"
#include "inp_E.h"
#include "inp_H.h"
#include "g4.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);

  G4 g4("g4");
  g4(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 4.8: File g4.vhd

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

entity g4 is
  port(
        h: in std_logic;
        e: in std_logic;
        s: out std_logic;
        rst: in std_logic
        );
end g4;

architecture RTL of g4 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 4.9: 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 g4
  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: G4 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;
```