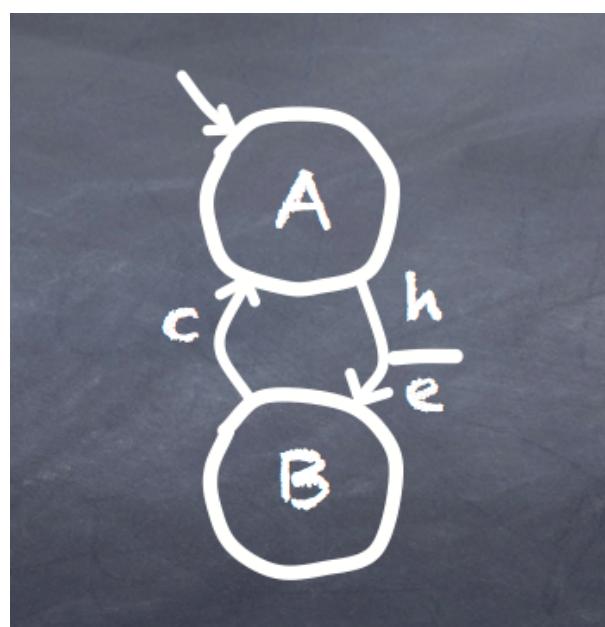


RFSM Reference Manual - 2.0

J. Sérot



Chapter 1

Formal syntax of RFSM programs

This appendix gives a BNF definition of the concrete syntax RFSM programs. As stated in the introduction, this syntax is that of the so-called *standard* RFSM language. Variant languages will essentially differ in the definition of the `<type_decl>`, `<type_expr>`, `<expr>`, `<constant>`, and `<const>` syntactical categories.

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

```

⟨program⟩ ::= ⟨type_decl⟩*
             ⟨cst_decl⟩*
             ⟨fn_decl⟩*
             ⟨fsm_model⟩*
             ⟨fsm_model⟩*
             ⟨global⟩*
             ⟨fsm_inst⟩*

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

⟨record_field⟩ ::= lid : ⟨type_expr⟩

⟨cst_decl⟩ ::= constant lid : ⟨type_expr⟩ = ⟨const⟩

⟨fn_decl⟩ ::= function lid ( ⟨farg⟩* ) : ⟨type_expr⟩ { return ⟨expr⟩ }

⟨farg⟩ ::= lid : ⟨type_expr⟩

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

⟨state⟩ ::= uid [where⟨outp_val⟩and]

⟨outp_val⟩ ::= lid = ⟨scalar_const⟩

⟨params⟩ ::= < ⟨param⟩* >

⟨param⟩ ::= lid : ⟨type_expr⟩

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

⟨io_desc⟩ ::= lid : ⟨type_expr⟩

⟨vars⟩ ::= vars : ⟨var⟩* ;

⟨var⟩ ::= lid+ : ⟨type_expr⟩

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

```

$\langle \text{rule_prefix} \rangle ::= | \quad | \quad !$
 $\langle \text{condition} \rangle ::= \mathbf{on} \text{ lid } [\langle \text{guards} \rangle]$
 $\langle \text{guards} \rangle ::= \mathbf{when} \langle \text{expr} \rangle^+$
 $\langle \text{actions} \rangle ::= \mathbf{with} \langle \text{action} \rangle^+;$
 $\langle \text{action} \rangle ::= \text{lid}$
 $\quad | \quad \langle \text{lval} \rangle := \langle \text{expr} \rangle$
 $\langle \text{lval} \rangle ::= \text{lid}$
 $\quad | \quad \text{lid } [\langle \text{expr} \rangle]$
 $\quad | \quad \text{lid } [\langle \text{expr} \rangle : \langle \text{expr} \rangle]$
 $\quad | \quad \text{lid . lid}$
 $\langle \text{itransition} \rangle ::= | \dashrightarrow \text{uid } [\langle \text{actions} \rangle]$
 $\langle \text{global} \rangle ::= \mathbf{input} \langle \text{id} \rangle : \langle \text{type_expr} \rangle = \langle \text{stimuli} \rangle$
 $\quad | \quad \mathbf{output} \langle \text{id} \rangle^+ : \langle \text{type_expr} \rangle$
 $\quad | \quad \mathbf{shared} \langle \text{id} \rangle^+ : \langle \text{type_expr} \rangle$
 $\langle \text{stimuli} \rangle ::= \mathbf{periodic} (\text{int} , \text{int} , \text{int})$
 $\quad | \quad \mathbf{sporadic} (\text{int}^*)$
 $\quad | \quad \mathbf{value_changes} (\langle \text{value_change} \rangle^*)$
 $\langle \text{value_change} \rangle ::= \mathbf{int} : \langle \text{stim_const} \rangle$
 $\langle \text{fsm_inst} \rangle ::= \mathbf{fsm} \langle \text{id} \rangle = \langle \text{id} \rangle [< \langle \text{param_value} \rangle^+ >] (\langle \text{id} \rangle^*)$
 $\langle \text{param_value} \rangle ::= \langle \text{scalar_const} \rangle$
 $\quad | \quad \text{lid}$
 $\langle \text{type_expr} \rangle ::= \mathbf{event}$
 $\quad | \quad \mathbf{int} \langle \text{int_annot} \rangle$
 $\quad | \quad \mathbf{float}$
 $\quad | \quad \mathbf{char}$
 $\quad | \quad \mathbf{bool}$
 $\quad | \quad \text{lid}$
 $\quad | \quad \langle \text{type_expr} \rangle \mathbf{array} [\langle \text{array_size} \rangle]$
 $\langle \text{int_annot} \rangle ::= \epsilon$
 $\quad | \quad < \langle \text{type_size} \rangle >$
 $\quad | \quad < \langle \text{type_size} \rangle : \langle \text{type_size} \rangle >$
 $\langle \text{array_size} \rangle ::= \langle \text{type_size} \rangle$

```

⟨type_size⟩ ::= int
              | lid

⟨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⟩ : ⟨expr⟩ ]
          | lid ( ⟨expr⟩* )
          | lid . lid
          | ⟨expr⟩ ? ⟨expr⟩ : ⟨expr⟩
          | ⟨expr⟩ :: ⟨type_expr⟩

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

⟨subtractive⟩ ::= -
                 | -. .

⟨scalar_const⟩ ::= int
                  | bool
                  | float
                  | char

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

```

```

⟨array_const⟩ ::= [ ⟨const⟩+ ]
⟨stim_const⟩ ::= ⟨scalar_const⟩
|   ⟨scalar_const⟩ :: ⟨type_expr⟩
|   uid
|   ⟨record_const⟩
⟨record_const⟩ ::= { ⟨record_field_const⟩+ }
⟨record_field_const⟩ ::= lid = ⟨stim_const⟩
⟨id⟩ ::= lid
|   uid

```

Chapter 2

Formal semantics

We here give the formal *static* and *dynamic* semantics for a simplified version of the RFSM language, called CORE RFSM. Compared to the “standard” RFSM language¹, 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 explicated here.

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

2.1 Common definitions

Both the static and dynamic 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 $x \in \text{dom}(\Gamma)$,
- $\Gamma(x)$ the value mapped to x in Γ ($\Gamma(x) = \perp$ if $x \notin \Gamma$),

¹Described in the User Manual.

- $\Gamma[x \mapsto v]$ the environment that maps x to v and behaves like Γ otherwise (possibly overriding an existing mapping of x),
- \emptyset the empty environment,

2.2 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,
- $I_e = \{x \in I \mid I(x) = \text{event}\}$ and $I_v = \{x \in I \mid I(x) \neq \text{event}\}$,
- $O_e = \{x \in O \mid O(x) = \text{event}\}$ and $O_v = \{x \in O \mid O(x) \neq \text{event}\}$,
- $T \subset Q \times C \times \mathcal{S}(A) \times Q$ 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 operators²,
- $\mathcal{S}(A)$ is the set of *sequences* built from elements of the set A , where a *sequence* \vec{a} is an ordered collection $\langle a_1; \dots; a_n \rangle$ ³,
- $A = \mathcal{U}(O_v \cup V, I_v \cup V) \cup O_e$,
- $\mathcal{U}(E, E')$ is the set of *assignments* of variables taken in a set E by *expressions* built from a set of variable E' and the classical boolean and arithmetic operators and constants⁴.
- $\tau_0 \in Q \times \mathcal{S}(\mathcal{U}(O_v \cup V, \emptyset))$ is the **initial transition**.

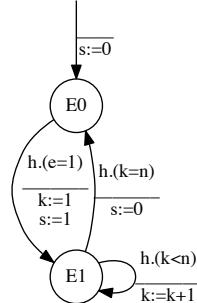
Having $\tau = (q, c, \vec{a}, q')$ in T means that there's a transition from state q to state q' enabled by the condition c and triggering a (possibly empty) sequence \vec{a} , where

- the condition $c \in C$ is made of
 - a triggering 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 emit events and the assigned values cannot depend on inputs or local variables.

Example. The model of the automaton depicted below⁵ can be formally described as $\mathcal{M} = \langle Q, I, O, V, T, \tau_0 \rangle$ where :

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



²This set can be formally derived from the abstract syntax.

³For example, if $A = \{1, 2\}$, then $\mathcal{S}(A) = \{\langle \rangle, \langle 1 \rangle, \langle 2 \rangle, \langle 1; 1 \rangle, \langle 1; 2 \rangle, \langle 2; 1 \rangle, \langle 2; 2 \rangle, \langle 1; 2; 1 \rangle, \dots\}$, where $\langle \rangle$ denotes the empty sequence.

⁴Again, this can be formally derived from the abstract syntax.

⁵This model, a calibrated pulse generator has been introduced in the *Overview* chapter of the user manual.

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).

$$\frac{\begin{array}{c} fsm_model^+ \rightarrow \Gamma_M \\ io_decl^+ \rightarrow \Gamma_I \\ \Gamma_M, \Gamma_I \vdash fsm_inst^+ \rightarrow M \\ C = \mathcal{L}(\Gamma_I) \end{array}}{\text{program } fsm_model^+ \text{ } io_decl^+ \text{ } fsm_inst^+ \rightarrow M, C} \quad (\text{PROGRAM})$$

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

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

where

$$\begin{array}{ll} I_e = \{x \in \text{dom}(\Gamma_I) \mid \Gamma_I(x) = \langle \text{input}, \text{event} \rangle\} & I_v = \{x \in \text{dom}(\Gamma_I) \mid \Gamma_I(x) = \langle \text{input}, \tau \rangle, \tau \neq \text{event}\} \\ O_e = \{x \in \text{dom}(\Gamma_I) \mid \Gamma_I(x) = \langle \text{output}, \text{event} \rangle\} & O_v = \{x \in \text{dom}(\Gamma_I) \mid \Gamma_I(x) = \langle \text{output}, \tau \rangle, \tau \neq \text{event}\} \\ H_e = \{x \in \text{dom}(\Gamma_I) \mid \Gamma_I(x) = \langle \text{shared}, \text{event} \rangle\} & H_v = \{x \in \text{dom}(\Gamma_I) \mid \Gamma_I(x) = \langle \text{shared}, \tau \rangle, \tau \neq \text{event}\} \end{array}$$

- Rule MODELS gives the interpretation of model declarations, giving an environment Γ_M .

$$\frac{\forall i \in \{1, \dots, n\} \quad \begin{array}{c} \Gamma_M^{i-1}, fsm_model_i \rightarrow \Gamma_M^i \\ \Gamma_M^0 = \emptyset \quad \Gamma_M = \Gamma_M^n \end{array}}{fsm_model_1, \dots, fsm_model_n \rightarrow \Gamma_M} \quad (\text{MODELS})$$

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

TODO: TBC

$$\frac{\mathcal{M} = \langle Q, I, O, V, T, \tau_0 \rangle \quad \text{valid_model}(\mathcal{M})}{\text{fsm model id } I \text{ } O \text{ } Q \text{ } V \text{ } T \text{ } \tau_0 \rightarrow \Gamma_M[\text{id} \mapsto \mathcal{M}]} \quad (\text{MODEL})$$

- Rules IOs and IO give the interpretation of IO declarations, producing an environment Γ_I binding names to a pair $\langle io_cat, typ \rangle$.

$$\frac{\forall i \in \{1, \dots, n\} \quad \begin{array}{c} \Gamma_I^{i-1}, io_decl_i \rightarrow \Gamma_I^i \\ \Gamma_I^0 = \emptyset \quad \Gamma_I = \Gamma_I^n \end{array}}{io_decl_1, \dots, io_decl_n \rightarrow \Gamma_I} \quad (\text{IOs})$$

$$\frac{}{\Gamma_I, \text{ cat id : } typ \rightarrow \Gamma_I[\text{id} \mapsto \langle \text{cat}, typ \rangle]} \quad (\text{IO})$$

- Rules INSTS gives the interpretation of FSM instance declarations.

$$\frac{\forall i \in \{1, \dots, n\} \quad \Gamma_M, \Gamma_I \vdash fsm_inst_i \rightarrow \mu_i \quad M = \langle \mu_1; \dots; \mu_n \rangle}{fsm_inst_1, \dots, fsm_inst_n \rightarrow \mathcal{H} = \langle M, C \rangle} \quad (\text{INSTS})$$

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

$$\frac{\begin{array}{c} \Gamma_M(\text{id}) = \langle \langle i'_1 : \tau'_1, \dots, i'_m : \tau'_m \rangle, \langle o'_1 : \tau''_1, \dots, o'_n : \tau''_n \rangle, Q, V, T, \langle q_0, \vec{a}_0 \rangle \rangle \\ \Phi = \{i'_1 \mapsto i_1, \dots, i'_m \mapsto i_m, o'_1 \mapsto o_1, \dots, o'_n \mapsto o_n\} \\ \forall i \in \{1, \dots, m\} \quad \Gamma_I(i_i) = \langle \text{cat}_i, \tau_i \rangle, \text{cat}_i \in \{\text{input, shared}\} \wedge \tau_i = \tau'_i \\ \forall i \in \{1, \dots, n\} \quad \Gamma_I(o_i) = \langle \text{cat}_i, \tau_i \rangle, \text{cat}_i \in \{\text{output, shared}\} \wedge \tau_i = \tau''_i \\ \mathcal{M}' = \langle \langle i_1 : \tau'_1, \dots, i_m : \tau'_m \rangle, \langle o_1 : \tau''_1, \dots, o_n : \tau''_n \rangle, Q, V, \Phi_T(T), \langle q_0, \Phi_A(\vec{a}_0) \rangle \rangle \\ \mu = \langle \mathcal{M}', q_0, \mathcal{I}(V) \rangle \end{array}}{\Gamma_M, \Gamma_I \vdash \text{fsm id } \langle i_1; \dots; i_m \rangle \langle o_1; \dots; o_n \rangle \rightarrow \mu} \quad (\text{INST})$$

Rule INST checks the arity and the type conformance of the inputs and outputs supplied to the instantiated 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 :

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

2.3 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 , $\mathcal{E}_\Gamma[e]$ 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

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

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).

TODO: Input et output par rapport à quoi : automate ou contexte ?

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 $\vec{\sigma} = \sigma_1, \dots, \sigma_n$ is formalized by rule EXEC

$$\frac{\begin{array}{c} C \vdash M \rightarrow M_0, \Gamma_0 \\ \forall i \in \{1, \dots, n\} \quad C \vdash M_{i-1}, \Gamma_{i-1} \xrightarrow[\rho_i]{\sigma_i} M_i, \Gamma_i \end{array}}{\mathcal{H} = \langle M, C \rangle \xrightarrow[\vec{\rho} = \langle \rho_1; \dots; \rho_n \rangle]{\vec{\sigma} = \langle \sigma_1; \dots; \sigma_n \rangle} M_n, \Gamma_n} \text{(EXEC)}$$

In other words, the execution of the program is described as a sequence of **instantaneous reactions**, which can be denoted as⁶ :

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

where

- the *global environment* Γ here records the value of inputs and shared variables⁷,
- ρ_1, \dots, ρ_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).

⁶Omitting context C , which is constant during an execution.

⁷This environment is required to handle events describing modifications of these values, as described below (see rule REACTUPD).

$$\frac{\begin{array}{c} \forall i \in \{1, \dots, n\} \quad \mu_i = \langle \mathcal{M}_i, q_i, \mathcal{V}_i \rangle \quad \mathcal{M}_i = \langle ., ., ., ., \langle ., \vec{a}_i \rangle \rangle \quad C \vdash \mathcal{V}_i, \gamma_{i-1} \xrightarrow[\langle 0, \emptyset \rangle]{\vec{a}_i, 0} \mathcal{V}'_i, \gamma_i \quad \mu'_i = \langle \mathcal{M}_i, q_i, \mathcal{V}'_i \rangle \\ C = \langle ., I_v, ., O_v, ., H_v \rangle \quad \gamma_0 = \mathcal{I}(I_v \cup O_v \cup H_v) \end{array}}{C \vdash M = \{\mu_1, \dots, \mu_n\} \rightarrow M_0 = \{\mu'_1, \dots, \mu'_n\}, \Gamma_0 = \gamma_n} \quad (\text{INIT})$$

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 responses⁸.

$$\frac{\begin{array}{c} \forall i \in \{1, \dots, n\} \quad C \vdash \mathcal{V}_{i-1}, \Gamma_{i-1} \xrightarrow[\rho_i]{a_i, t} \mathcal{V}_i, \Gamma_i \\ \mathcal{V}_0 = \mathcal{V} \quad \Gamma_0 = \Gamma \quad \rho_e = \bigcup_{i=1}^n \rho_i \end{array}}{C \vdash \mathcal{V}, \Gamma \xrightarrow[\rho_e]{\langle a_1; \dots; a_n \rangle, t} \mathcal{V}_n, \Gamma_n} \quad (\text{ACTS})$$

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

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

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 parallel⁹.

► Rules ACTUPDL and ACTUPDG respectively describe the effect of an action updating a local or global variable (shared variable or output)¹⁰.

$$\frac{x \in \text{dom}(\mathcal{V}) \quad v = \mathcal{E}_{\mathcal{V} \cup \Gamma}[\epsilon]}{C \vdash \mathcal{V}, \Gamma \xrightarrow[\langle t, \emptyset \rangle]{x \leftarrow \epsilon, t} \mathcal{V}[x \mapsto v], \Gamma} \quad (\text{ACTUPDL}) \quad \frac{x \in \text{dom}(\Gamma) \quad v = \mathcal{E}_{\mathcal{V} \cup \Gamma}[\epsilon]}{C \vdash \mathcal{V}, \Gamma \xrightarrow[\langle t, \emptyset \rangle]{x \leftarrow \epsilon, t} \mathcal{V}, \Gamma[x \mapsto v]} \quad (\text{ACTUPDG})$$

► 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' .

$$\frac{\sigma_e, \sigma_v = \Sigma(\sigma) \quad C \vdash M, \Gamma \xrightarrow{\sigma_v} M, \Gamma_v \quad C \vdash M, \Gamma_v \xrightarrow[\rho_e]{\sigma_e} M', \Gamma'}{C \vdash M, \Gamma \xrightarrow[\rho_e]{\sigma} M', \Gamma'} \quad (\text{REACT})$$

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

⁸This set of responses is always empty when rule ACTS is invoked in the context of INIT.

⁹As happens in hardware synchronous implementations for example.

¹⁰The effect of an action emitting an event will be described by rules ACTEMITS and ACTEMITG, given latter.

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

- 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 :

$$\frac{}{C \vdash M, \Gamma \xrightarrow{\sigma_v = \langle t, \{x_1 \leftarrow v_1, \dots, x_m \leftarrow v_m\} \rangle} M, \Gamma[x_1 \mapsto v_1] \dots [x_m \mapsto v_m]} \quad (\text{REACTUPD})$$

- Rule REACTEV describes how a program reacts to a set of pure events.

$$\frac{\begin{array}{c} \forall i \in \{1, \dots, n\} \quad C \vdash \mu_{\pi(i)}, \Gamma_{i-1} \xrightarrow[\rho_i]{\sigma_i} \mu'_{\pi(i)}, \Gamma_i \quad \sigma_i = \sigma_{i-1} \cup \rho_i \\ \Gamma_0 = \Gamma \quad \sigma_0 = \sigma_e \quad \rho_e = \bigcup_{i=1}^n \rho_i \quad \Gamma' = \Gamma_n \end{array}}{C \vdash M = \{\mu_1, \dots, \mu_n\}, \Gamma \xrightarrow[\rho_e]{\sigma_e = \langle t, \{\epsilon_1, \dots, \epsilon_m\} \rangle} M' = \{\mu'_1, \dots, \mu'_n\}, \Gamma'} \quad (\text{REACTEV})$$

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 occurring in the guards associated to τ' is written by the actions associated to τ .

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

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

Having the automata of M react in the order $\pi(1), \dots, \pi(n)$ 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 broadcasts*.

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.

$$\frac{\Delta_{\Gamma \cup \mathcal{V}}(\mathcal{M}, q, e) = \{\tau\} \quad C \vdash \mu, \Gamma \xrightarrow[\rho_e]{\tau, t} \mu', \Gamma'}{C \vdash \mu = \langle \mathcal{M}, q, \mathcal{V} \rangle, \Gamma \xrightarrow[\rho_e]{\sigma_e = \langle t, e \rangle} \mu', \Gamma'} \quad (\text{REACT1})$$

$$\frac{\Delta_{\Gamma \cup \mathcal{V}}(\mathcal{M}, q, e) = \emptyset \quad C \vdash \mu = \langle \mathcal{M}, q, \mathcal{V} \rangle, \Gamma \xrightarrow{\langle t, e \rangle} \mu, \Gamma}{\langle t, \emptyset \rangle} \quad (\text{REACT0})$$

$$\frac{\begin{array}{c} \Delta_{\Gamma \cup \mathcal{V}}(\mathcal{M}, q, e) = \{\tau_1, \dots, \tau_n\} \\ \tau = \text{choice}(\{\tau_1, \dots, \tau_n\}) \\ C \vdash \mu, \Gamma \xrightarrow[\rho_e]{\tau, t} \mu', \Gamma' \end{array}}{C \vdash \mu = \langle \mathcal{M}, q, \mathcal{V} \rangle, \Gamma \xrightarrow[\rho_e]{\sigma_e = \langle t, e \rangle} \mu', \Gamma'} \quad (\text{REACTN})$$

Given a automaton modelised by \mathcal{M} and currently in state q , $\Delta_\Gamma(\mathcal{M}, q, e)$, where $e = \{\epsilon_1, \dots, \epsilon_n\}$, 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**.

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

where

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

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 transition¹¹.

► 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.

$$\frac{\begin{array}{c} \mu = \langle \mathcal{M}, q, \mathcal{V} \rangle \quad \tau = \langle q, c, \vec{a}, q' \rangle \\ C \vdash \mathcal{V}, \Gamma \xrightarrow[\rho_e]{\vec{a}, t} \mathcal{V}', \Gamma' \\ \mu' = \langle \mathcal{M}, q', \mathcal{V}' \rangle \end{array}}{C \vdash \mu, \Gamma \xrightarrow[\rho_e]{\tau, t} \mu', \Gamma'} \quad (\text{TRANS})$$

► 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).

$$\frac{C = \langle ., ., ., ., H_e, . \rangle \quad \epsilon \in H_e}{C \vdash \mathcal{V}, \Gamma \xrightarrow[\langle t, \{\epsilon\} \rangle]{\epsilon, t} \mathcal{V}, \Gamma} \quad (\text{ACTEMITS}) \quad \frac{C = \langle ., ., ., ., H_e, . \rangle \quad \epsilon \notin H_e}{C \vdash \mathcal{V}, \Gamma \xrightarrow[\langle t, \emptyset \rangle]{\epsilon, t} \mathcal{V}, \Gamma} \quad (\text{ACTEMITG})$$

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.

¹¹This can done, for example, by adding a *priority* to each transition.

Chapter 3

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: <code>jopam_prefix;/share/rfsm</code>) |
| -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 uninstantiated FSM models |
| -server_mode | run the compiler in server mode (to be used by the grasp app) |
| -verbose | run the compiler in verbose mode (including server) |
| -socket_path | set socket path when running in server mode |
| -dot_qual_ids | print qualified identifiers in DOT representations |
| -dot_captions | Add 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) |
| -dot_boxed | Draw FSM instances in boxes |
| -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 |
| -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 |

Chapter 4

Building language variants

Following the approach described in [1] for example, the RFSM compiler is implemented in a *modular* way. The language is split into a *host* language, describing the general structure and behavior of FSMs (states, transitions, . . .) and a *guest* language¹ describing the syntax and semantics of *expressions* used in transition guards and actions. Technically, this “separation of concern” is realized by providing the host language in the form of a *functor* taking as argument the module implementing the guest language.

This approach makes it fairly easy to produce variants of the “standard” RFSM language – with dedicated type systems and expression languages typically – by simply defining the module defining the guest language and applying the aforementioned functor. Actually, the “standard”, RFSM language was designed using this approach, starting from a very simple “core” guest language gradually enriched with new features at the expression level².

The directory `src/guests/templ` in the distribution provides the basic structure for deploying this approach. In practice, to implement a language variant, one has to

- write the implementation of the guest language in the form of a collection of modules in the `lib` subdirectory³,
- write the lexer and the parser for this guest language (expressions, type expressions, . . .),
- build the compiler by simply invoking `make`

To illustrate this process, we describe in the sequel the implementation of a very simple language language for which the guest language has only two types, ‘event’ and ‘bool’, and expressions are limited to boolean constants and variables⁴. We focus here on the most salient features. The `Readme` file in the `templ` directory describes the procedure in details. Other examples, given in `src/guests/core` and `src/guests/others`⁵, can also be used as guidelines.

4.1 Implementing the Guest module

The module implementing the guest language must match the following signature :

¹“Base language” in the terminology of [1].

²Traces of the incremental design process can be found in the `srs/guests/core`, `src/guests/others/szdints` and `src/guests/others/szvars` directories for example.

³These modules will be encapsulated in a single module and the latter will be passed to the host functor to build the target language.

⁴This language is essentially that provided in `src/guests/others/min`.

⁵And, of course, in `src/guests/std`.

```

module type T = sig
  module Info : INFO
  module Types : TYPES
  module Syntax : SYNTAX with module Types = Types
  module Typing : TYPING with module Syntax = Syntax and module Types = Types
  module Value : VALUE with type typ = Types.typ
  module Static : STATIC with type expr = Syntax.expr and type value = Value.t
  module Eval : EVAL with module Syntax = Syntax and module Value = Value
  module Ctask: CTASK with module Syntax = Syntax
  module Systemc: SYSTEMC with module Syntax = Syntax and module Static = Static
    and type value = Value.t
  module Vhdl: VHDL with module Syntax = Syntax and module Static = Static and
    type value = Value.t
  module Error : ERROR
  module Options : OPTIONS
end

```

where

- module **Info** gives the name and version of the guest language,
- module **Syntax** describes the (abstract) syntax of the guest language,
- modules **Types** and **Typing** respectively describe the types and the typing rules of the guest language,
- module **Static** describes the static semantics of the guest language (basically, the interpretation of model parameters),
- modules **Value** and **Eval** respectively describe the values and the dynamic semantics manipulating these values,
- modules **CTask**, **Systemc** and **Vhdl** respectively describe the guest-level part of the C, SystemC and VHDL backends,
- module **Error** describes how guest-specific errors are handled,
- module **Options** describes guest-specific compiler options.

Listings 4.1, 4.2, 4.4 and 4.6 respectively show the contents of the **Types**, **Syntax**, **Value** and **Eval** modules for the **mini** language. The definition of non essential functions has been omitted⁶.

Listing 4.1: Module **Guest.Types** (excerpt)

```

type typ =
| TyEvent
| TyBool
| TyUnknown

let no_type = TyUnknown

let is_event_type (t: typ) = match t with TyEvent -> true | _ -> false
let is_bool_type (t: typ) = match t with TyBool -> true | _ -> false

let pp_typ ?(abbrev=false) fmt (t: typ) = ...

```

⁶See the corresponding source files in `src/guests/others/mini/lib` for a complete listing.

In module `Types`, the key definition is that of type `typ`, which describes the guest-level types, *i.e.* the types which can be attributed to guest-level expressions and variables. The type `TyUnknown` is used to define the value `no_type` which is attributed by the host language to (yet) untyped syntax elements.

Listing 4.2: Module `Guest.Syntax` (excerpt)

```
module Types = Types

module Location = Rfsm.Location
module Annot = Rfsm.Annot
module Ident = Rfsm.Ident

let mk ~loc x = Annot.mk ~loc ~typ:Types.no_type x

(* Type expressions *)

type type_expr = (type_expr_desc,Types.typ) Annot.t
and type_expr_desc = TeConstr of string (* name, no args here *)

let is_bool_type (te: type_expr) = ...
let is_event_type (te: type_expr) = ...

let pp_type_expr fmt (te: type_expr) = ...

(* Expressions *)

type expr = (expr_desc,Types.typ) Annot.t
and expr_desc =
| EVar of Ident.t
| EBool of bool

let vars_of_expr (e: expr) = ...
and pp_expr fmt (e: expr) = ...

(* L-values *)

type lval = (lval_desc,Types.typ) Annot.t
and lval_desc = Ident.t

let lval_var (l: lval) = ...
let vars_of_lval (l: lval) = ...
let is_simple_lval (l: lval) = true

let mk_simple_lval (v: Ident.t) =
  Annot.{ desc=v; typ=Types.no_type; loc=Location.no_location }

let pp_lval fmt l = ...
```

The module `Syntax` uses the modules `Location`, `Annot` and `Ident` provided by the `Rfsm` host library. These modules provide types and functions to handle source code locations, syntax annotations and identifiers respectively. The type `('a,Types.typ) Annot.t` is associated to syntax nodes of type `'a`. The types `type_expr`, `expr` and `lval` respectively describe guest-level type expressions, expressions and l-values. L-values are used in the definition of actions. In this language they are limited to simple identifiers (ex: `x:=<expr>`) but the guest language can use other forms (like in the RFSM “standard” language which support arrays and records as l-values).

Listing 4.3: Module `Guest.Values` (excerpt)

```
type typ =
| TyEvent
| TyBool
| TyUnknown

let no_type = TyUnknown

let is_event_type t = match t with TyEvent -> true | _ -> false
let is_bool_type t = match t with TyBool -> true | _ -> false
let mk_type_fun ty_args ty_res = ...

let pp_typ ?(abbrev=false) fmt t = ...
```

Listing 4.4: Module `Guest.Value` (excerpt)

```
type t =
| Val_bool of bool
| Val_unknown

let default_value ty = match ty with
| _ -> Val_unknown

exception Unsupported_vcd of t

let vcd_type (v: t) = match v with
| Val_bool _ -> Rfsm.Vcd_types.TyBool
| _ -> raise (Unsupported_vcd v)

let vcd_value (v: t) = match v with
| Val_bool v -> Rfsm.Vcd_types.Val_bool v
| _ -> raise (Unsupported_vcd v)

let pp fmt (v: t) = ...
```

The module `Guest.Value` defines the values, associated to guest-level expressions by the dynamic semantics. The value `Val_unknown` is used to represent undefined or uninitialized value. The functions `vcd_type` and `vcd_value` provide the interface to the VCD backend, generating simulation traces : they should return a VCD compatible representation (defined in the host library `Vcd_types` module) of a value.

Listing 4.5: Module `Guest.Eval` (excerpt)

```
module Syntax = Syntax
module Value = Value
module Env = Rfsm.Env
module Annot = Rfsm.Annot

type env = Value.t Env.t

exception Illegal_expr of Syntax.expr
exception Uninitialized of Rfsm.Location.t

let mk_env () = Env.init []
```

```

let upd_env lval v env = Env.upd lval.Annot.desc v env

let lookup ~loc v env =
  match Rfsm.Env.find v env with
  | Value.Val_unknown -> raise (Uninitialized loc)
  | v -> v

let eval_expr env e = match e.Annot.desc with
  | Syntax.EVar v -> lookup ~loc:e.Annot.loc v env
  | Syntax.EBool i -> Val_bool i

let eval_bool env e =
  match eval_expr env e with
  | Val_bool b -> b
  | _ -> raise (Illegal_expr e)

let pp_env fmt env = ...

```

The module `Guest.Eval` defines the guest-level dynamic semantics, *i.e.* the definition of the dynamic environment `env`, used to bind identifiers to values and of the functions `eval_expr` and `eval_bool` used to evaluate guest-level expressions. The presence of a distinct, `eval_bool` function is required because the boolean type and expressions are not part of the host-level syntax. The definition of the `env` type here uses that provided by the RFSM host library but any definition matching the corresponding signature would do.

4.2 Implementing the guest language parser

The parser for the guest language defines the concrete syntax for guest-level type expressions, expressions and l-values. It is written in a separate `.mly` file which will be combined with the parser for the host language⁷. There are only two guest specific tokens here, denoting the boolean constants `true` and `false`. The `open` directive in the prologue section gives access to the abstract syntax definitions given in `Guest.Syntax` module. The function `mk`, defined in this module, builds annotated syntax nodes, inserting the source code location.

Listing 4.6: File `guest_parser.mly`

```

%token TRUE
%token FALSE

%{
open Mini.Top.Syntax
%}

%%
%public type_expr:
| tc = LID { mk ~loc:$sloc (TeConstr tc) }

%public lval:
| v = LID { mk ~loc:$sloc (mk_ident v) }

%public expr:

```

⁷Using `menhir` facility to split parser specifications into multiple files.

```

| v = LID { mk ~loc:$sloc (EVar (mk_ident v)) }
| c = scalar_const { c }

%public scalar_const:
| TRUE { mk ~loc:$sloc (EBool true) }
| FALSE { mk ~loc:$sloc (EBool false) }

%public const:
| c = scalar_const { c }

%public stim_const:
| c = scalar_const { c }

```

4.3 Implementing the guest language lexer

The `ocamllex` tool, used for defining the lexer, does not support multi-file definitions. The lexer for the guest-specific part of the target language is therefore supplied in the form of code fragments to be inserted in the host lexer definition file⁸. These fragments are listed in two separate files, which must be supplied in the `bin` subdirectory of the guest directory :

- the file `guest_kw.mll` contains the lexer definition of the guest-specific keywords,
- the file `guest_rules.mll` contains the guest-specific rules.

In the case of the `mini` language defined here, the latter is empty and the former only contains the two following lines.

```
"true", TRUE;
>false", FALSE;
```

4.4 Building the compiler

Defining the target language implementation and building the associated compiler is then simply obtained by the two following functor applications⁹ :

```

module L = Rfsm.Host.Make(Mini.Top)

module Compiler =
  Rfsm.Compiler.Make
    (L)
    (Lexer)
    (struct include Parser type program = L.Syntax.program end)

```

Invoking the compiler now boils down to executing

```
let _ = Printexc.print Compiler.main ()
```

⁸Technically, this insertion is performed using the `cppo` tool.

⁹For technical reasons, these two statements are placed in distinct files, `binlang.ml` and `binrfsmc.ml`.

Bibliography

- [1] X. Leroy. *A Modular Module System*. J. Functional Programming, 10(3):269-303, 2000.