A Structure

A.1 System

In general, we deal with systems composed from active objects (process instances) which are executed in parallel and interact asynchronously[1] through message (signal instances) passing via communication buffers (signalroute instances). Therefore, system specifications consist of generic components, including dynamic ones such as processes, signalroutes and signals, and static ones such as (shared) variables, data types, constant values, and external procedures.

system-decl ::=
  system *system-id* **;**
    { *system-component* }*
  endsystem ;

system-component ::=
  process-decl |
  signalroute-decl |
  signal-decl |
  procedure-decl |
  var-decl |
  type-decl |
  const-decl

A.2 Process

Processes are defined as extended finite state-machines. At execution, process instances could be created and destroyed dynamically. Each process instance has a unique identifier number (the pid of the instance) and an input fifo buffer storing all the incoming messages (received and not yet consummed by the instance). Process specifications include the name of the process, the number of initial instances (existing in the initial configuration of the system), the list of formal parameters (...), and various process components – the most important being the set of local variables which define the local memory and the (hierarchical) set of states (with the associated transitions) defining the behaviour of the process. In addition, (local) data type, constant or procedure definitions could also be included inside process specifications.

process-decl ::=
  process *process-id* **(** *const* **) ;**
    [**fpar** *fpar-decl* { **,** *fpar-decl* }* **;**]
    { *process-component* }*

[1]further development may include also synchronous interactions (e.g. rendez-vous, remote procedure calls, etc.)

endprocess ;

process-component ::=
state |
procedure-decl |
var-decl |
type-decl |
const-decl

A.3 Signalroute

Signalroutes denote communication paths between processes. At execution, signalroute instances could be created and destroyed dynamically. Again, each signalroute instance has a unique identifier number (the pid of the instance). Signalroute specifications include the name of the signalroute, the initial number of instances, the source and destination endpoints (process or environment) and the set of signals which could be transported on it. In addition, several options are available to refine the signalroute behaviour: the queueing policy (fifo or multiset), the reliability (reliable or lossy), the delivering policy ([point-to-point] peer, multicast or unicast) and the delaying policy ([immediate] urgent, [overall] delay or [transmission] rate).

signalroute-decl ::=
signalroute *signalroute-id* **(** *const* **)**
{ *signalroute-option* }*
from { *process-id* | **env** } **to** { *process-id* | **env** }
with *signal-id* { **,** *signal-id* }* **;**

signalroute-option ::=
#fifo | **#multiset** |
#reliable | **#lossy** |
#peer | **#multicast** | **#unicast** |
#urgent | **#delay[l,u]** | **#rate[l,u]**

A.4 Signal

Signals are used to communicate between processes. At execution, signal instances are sent (possible through signalroutes) from one process instance to some other instance(s). Signal specifications include the signal name and the list of parameter types carried by the signal.

signal-decl ::=
signal *signal-id* **(** [*type-id* { **,** *type-id* }*] **)** **;**

B Behaviour

B.1 State

States are the main structuring concept of the process behaviour. In general, state specifications include the (outgoing) transitions to other states (i.e. the behavior of that state). If needed, state specificatons may also include nested (sub)states which, in turn, could have their own transitions (and so on) thus refining/completing the overall behaviour of the state/process. In addition, state specifications include the set of (deferred) saved signals, which have to be postponed (if received), and the time progress condition (the temporal constraint) which limit the stay in that particular state. Furthermore, some options are available to define respectively the initial state of the process (or of compound states) and the stability (stable or unstable) which control process execution steps (atomic, un-interruptive from one stable state to another).

state ::=
  state *state-id* { *state-option* }* **;**
    [**tpc** *constraint*]
    [**save** *signal-id* { **,** *signal-id* }*]
    { *state-component* }*
  endstate ;

state-component ::=
  transition |
  state

state-option ::=
  #start |
  #stable | **#unstable**

B.2 Transition

Transitions are process reactions in response to stimulis. Transition stimuli are combinations of the following three elements : enabledness of some untimed guard (provided expression), enabledness of some timed guard (when constraint) and the presence of some signal in the input buffer of the process instance (input signal). Transition reactions are classical programs built over elementary actions using sequential composition, and respectively conditional (if-then-else) and loop (while) control-flow statements. Elementary actions are discussed below. A transition must end either with a nextstate action (which means passing control to that state), either with a stop action (which means the destruction of the instance). In addition, each transition has a deadline eager, delayable or lazy denoting the priority of the transition with respect to the progress of time.

transition ::=

[**deadline** { **eager** | **delayable** | **lazy** } **;**]
[**provided** *expression* **;**]
[**when** *constraint* **;**]
[**input** *signal-id* **(** [*expression* { **,** *expression* }*] **)** **;**]
{ *statement* }*
terminator

statement ::=
action |
if *expression* **then** { *statement* }* [**else** { *statement* }*] **endif** |
while *expression* **do** { *statement* }* **endwhile**

terminator ::=
nextstate *state-id* **;** |
stop ;

B.3 Action

The available actions which could be performed inside transitions include: skip (internal action), informal (observable action), task (variable assignments), set (clock setting), reset (clock and variable resetting), output (signal sending, possible via signalroutes), call (procedure calls), fork (process/signalroute instance creation), kill (process/signalroute instance destruction). In particular, note that call and fork actions return values which could be used in assignments.

action ::=
skip ; |
informal *string* **;** |
task *expression* **:=** *expression* **;** |
set *expression* **:=** *expression* **;** |
reset *expression* **;** |
output *signal-id* **(** [*expression* { **,** *expression* }*]**)**
[**via** *signalroute-id*] [**to** *expression*] **;** |
[*expression* **:=**] **call** *procedure-id* **(** [*expression* { **,** *expression* }*] **)** **;** |
[*expression* **:=**] **fork** *process-id* **(** [*expression* { **,** *expression* }*] **)** **;** |
[*expression* **:=**] **fork** *signalroute-id* **(** [*expression* { **,** *expression* }*] **)** **;** |
kill *expression* **;**

C Data

C.1 Constant

Constants are used to name some interesting values in the system specification. Therefore, constants are used mainly in expressions, but they can also be used to define parameterized specifications through data types (e.g. the size of arrays) or system configurations (e.g. the initial number of instances). Particular constants are *self* which denote the pid of the instance and *nil* which denotes the null pid.

const-decl ::=
    const *const-id* [**=** *const*] **;**

const ::=
    true | **false** |
    integer |
    float |
    self | **nil** |
    const-id

C.2 Type

Five predefined types are available to the user : boolean, integer, float, pid and clock. In addition, the user could define its own data types using classical type definition mechanisms such as enumeration, range (domains), record (structures) and array. It could also define even complex types using the abstract type definition mechanism – only the signature of the abstract type is defined, and in order to use it, an external implementation must be provided.

type-decl ::=
    type *type-id* **=** *type* **;**

type ::=
    enum *const-id* { **,** *const-id* }* **endenum** |
    range *const* **..** *const* |
    array **[** *const* **..** *const* **]** **of** *type-id* |
    record { *field-decl* }* **endrecord** |
    string **[** *const* **]** **of** *type-id* |
    abstract { *function-decl* }* **endabstract** |
    type-id

field-decl ::=
    field-id *type-id* **;**

function-decl ::=
    type-id *function-id* **(** [*type-id* { **,** *type-id* }*] **)** **;**

C.3 Variable

Variables, including formal parameters, have name and type. ...

var-decl ::=
  var *var-id* *type-id* { **private** | **public** } **;**

fpar-decl ::=
  [**in** | **inout** | **out**] *fpar-id* *type-id*

C.4 Expression

Expressions are built on top of constants and variables using a rich set of operators, including standard boolean operators, arithmetic operators, and comparison operators. Also, standard '.' and '[]' operators are used to access respectively fields in structured variables, or elements in arrays. In addition, we define a cast-operator allowing to convert, for instance, pid or enum values to integers (and back) and the *active* operator to test the activity of clocks. Finally, conditional expressions are allowed too.

expression ::=
  const |
  variable-id | *fpar-id* |
  expression **.** *field-id* | *expression* **[** *expression* **]** |
  function-id **(** [*expression* { **,** *expression* }*] **)** |
  un-op *expression* |
  expression *bin-op* *expression* |
  cast-op *expression* |
  expression **?** *expression* **:** *expression* |
  (*expression* **)**

un-op ::=
  not |
  active |
  + | −

bin-op ::=
  or | **and** |
  = | <> | < | <= | >= | > |
  + | − | ∗ | / | %

cast-op ::=
  { *type-id* | *process-id* | *signalroute-id* **}**

C.5 Constraint

Constraints are particular conditions on clocks. Basically, they are conjunctions of atomic-constraints, which are comparison between respectively clocks or clock differences and discrete (integer-type) expressions.

constraint ::=
    atomic-constraint { **and** *atomic-constraint* }*

atomic-constraint ::=
    expression comp-op expression |
    expression − *expression comp-op expression*

comp-op ::=
    = | < | <= | >= | >

D External

D.1 Procedure

Procedures are the mean to import and use external code into IF specifications. They are intended to describe intensive data transformation, executed without interaction in some instance. Procedure specifications include their signature (procedure name, formal parameters, return type) and respectively their actual code (written directly in the external language).

procedure-decl ::=
    procedure *procedure-id* **;**
        [**fpar** *fpar-decl* { **,** *fpar-decl* }* **;**]
        [**returns** *type-id* **;**]
        [**{# c code #}**]
    endprocedure ;