Common Description Language

VERIMAG
Centre Equation, 2 Avenue de Vignate
38610 Gières France

June 23, 2003

1 Introduction

The common language proposed is an extension of the IF language developed at Verimag in order to model asychronous communicating real-time systems. Since its definition in 1998, the IF language was continously improved and becomes the *interchange format* between a set of validation tools dedicated to real-time systems i.e, the so-called IF validation environment. In this project, we extend the IF language, and more generally, the IF environment to be able to model and analyze systems that rely upon dynamic process creation and dynamic data types.

The document is organized as follows. Section 2 presents the extended automata model i.e, the formal basis of the common language. In section 3 we summarize the main design decisions. Section 4 gives an incremental presentation of the common language through several examples. Finally, in section 5 we discuss the missing features and the status of the tools. The annexes A, B, C and D give the complete language syntax in BNF and some more syntax-oriented explanations.

2 Extended Automata

The formal basis for our language is a dynamic version of extended automata.

We focus on systems composed from several components, hereafter called processes, running in parallel and interacting through point-to-point message-passing. The number of processes may change over time: they may be created and destroyed dynamically using specific actions, during the system execution.

Each process is an extended timed automaton. They have unique process identifier (*pid*) values and local memory consisting of variables (including clocks), control state and a queue of pending messages (received and not yet consummed). As usual, processes move from one control state to another by executing transitions. In a control state, transitions are trigerred by the presence of some message in the input queue and/or some (possibly timed) condition on variables.

In addition, transition bodies are *sequential* programs consisting of elementary actions such as variable assignments, clock settings, message sending, process creation/destruction, etc.

The semantics of the extended automata model is given by the graph of its executions. This graph is obtained by the *interleaved* execution of processes, that is, one process is executing a transition at a time, while the others are waiting. In other words, we consider process transitions to define *atomic*, non-interruptive, execution steps.

The semantics of time is as for timed automata: time may progress only in states (i.e, all running processes wait in some state before selecting and executing some transition) and transitions take 0 time to be executed. In order to control the time progress, or equivalently, the waiting time in states, we rely on transition urgencies – explicit deadlines attached to them defining when these transitions *must* be executed.

3 Design Decisions

The common language inherits from IF some of its main features, in particular:

- *structuring concepts:* systems consist of processes, running in parallel and communicating through message passing via communication buffers;
- *communication primitives:* more exactly, processes may communicate either through signal exchange (directly or via signalroutes) or through shared variables;
- *real-time primitives:* each process may use several clocks to measure time during the execution and, in addition, transitions may be guarded with time constraints (depending on clocks) and decorated with explicit (*eager, delayable, lazy*) deadlines;
- *open systems:* the language offers the concept of open communication channel, connected to the environment, and transporting messages between it and the system;
- *non-determinism:* processes may be non-deterministic i.e, more than one transition may be enabled at some control state, and all situations have to be considered at execution;
- *complex data types:* the language provide several *type constructors* such as enumeration, range, array, record, abstract as well as *predefined basic types* in order to simplify complex data description and manipulation;

In order to fit the new requirements, the common language improves the IF language in several directions, the most important being listed below:

- *parameterisation:* using integer constants it is now possible to parameterize data types (i.e, size of arrays), system configurations (i.e, number of instances), timing behavior (i.e, clock constraints)...;
- *dynamic features:* the language include dynamic creation and destruction of process and signalroute (channel) instances. This makes system configuration to be *dynamic*, that is, the number of components running (and in turn, the number of clocks ...) may change during execution;
- *structured control:* the common language integrates *hierachical states* (to structure automata) and,
- *composed transitions:* basic control statements such as *if-then-else* and *while-do* are provided to structure automata transitions;
- *external code integration:* the common language provide a simple an elegant way to abstract complex transformations on data through the integration of external code within *procedures*. The external code to be provided depend on tools used i.e, an executable implementation in order to simulate and model check, or a first-order axiomatic definition in order to use it inside a prover, etc.

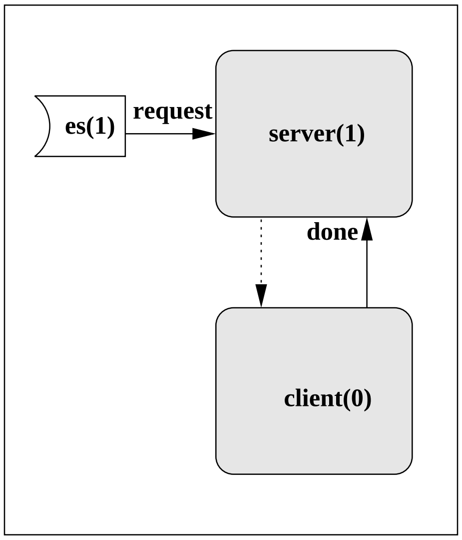
4 Examples

4.1 Parallel Server

This example describes a parallel server which can handle at maximum N requests simultaneously. Thus, when possible, for an incoming `request` message (received from the environment) a client process is created. The server keeps in a local variable the number of running clients. Client processes are quite simple: once created, they work for a while, and when finished they send a `done` message back to the server. The architecture of this application and the corresponding specification are presented in figure 1.

Basically, the specification contains several definitions. First of all, the constant `N`, which is a parameter for this specification, gives the maximal allowed number of clients running in parallel. Then, the signals `request` and `done`, used to communicate respectively between the server and the environment, and between the clients and the server, are defined. The signalroute `es` is an *open* (`from env`) communication channel transporting the requests. Finally, there are descriptions of respectively the `server` and `client` processes. The numbers after process and signalroute declarations indicate how many instances exist in the initital system configuration.

Let us focus now on the `server` process definition. It has one local integer variable `i`, counting the number of running clients. Its behavior is described by an one-state automaton with two transitions. The first transition models the reaction at the incoming requests: if the counter does not exceed the maximal allowed value, a new client is created and the counter is increased. The second




```
system server;
  const N = 2;
  signal request();
  signal done();
  signalroute es(1)
    from env to server
    with request;
  process server(1);
    ...
  endprocess;
  process client(0);
    ...
  endprocess;
endsystem;
```

Figure 1: Architecture specification of parallel server application.

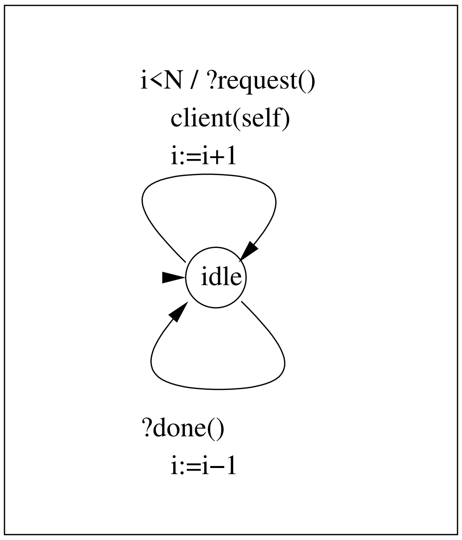
transition models the reaction at termination signals by decreasing the counter. The automaton and repectively the corresponding specification are presented in figure 2.

In the server specification, the *#start* construct is an *option* string which indicates that the `idle` state is the initial state of the automaton. Let us remark the use of the *self* construct, allowing to obtain the process identifier of the `server`. Here, this value is then given as parameter to each new client instance created (through the fork instruction).

Finally, the figure 3 gives the specification of `client` processes. Once created, they perform some informal `work` then send back to the parent the `done` signal. We recall here that the communication is asynchronous point-to-point. This means that, on one hand, in order to send some message, one has to know apriori the identity of the receiver process (hence, the need for the `parent` parameter which stores the identity of the server, who creates the client). On the other hand, asynchronous means that the sender is never blocked by the disponibility of the receiver, the message being imediately delivered to the receiver (i.e, stored in its input-queue of pending messages). Finally, the stop construct denotes the (auto) destruction of the process instance.

4.2 File System

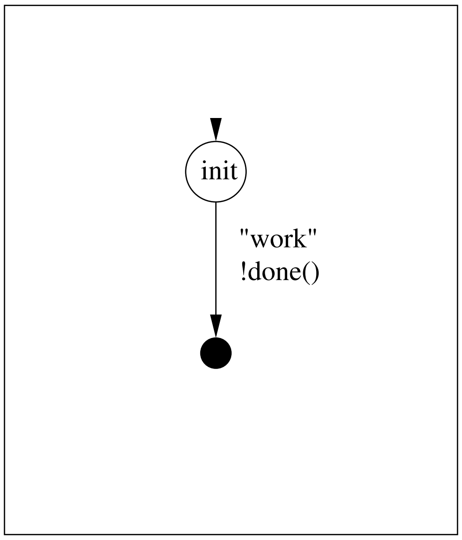
The second example describes a simple filesystem with several users communicating through a file server. The users address read/write requests to the server and wait for the corresponding accept/reject answers. The server policy is parameterized by two external procedures *FileAvailableForRead* and *FileAvailableForWrite*, described here by external C code. The global architecture is




```
process server(1);
  var i integer;

  state idle #start ;
    provided i < N;
    input request();
      fork client(self);
      task i := i + 1;
        nextstate idle;
    input done();
      task i := i - 1;
        nextstate idle;
  endstate;
endprocess;
```

Figure 2: Behavior specification of `server` processes.

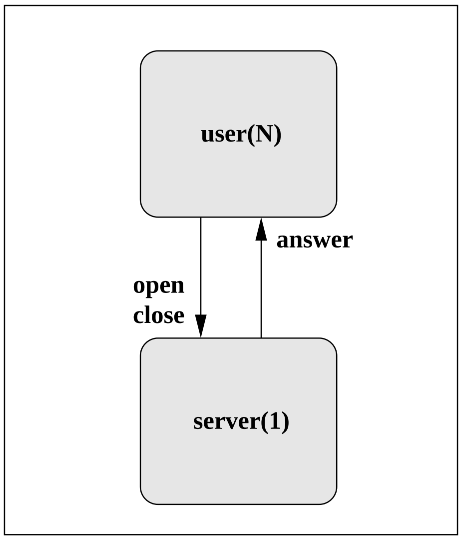



```
process client(0);
  fpar parent pid;

  state init #start ;
    informal "work";
    output done()
      to parent;
    stop;
  endstate;
endprocess;
```

Figure 3: Behaviour specification of `client` processes.

presented in figure 4.




```
system filesystem;
  const N = 3;
  const M = 4;
  type User = range 1 ..  N;
  type File = range 1 ..  M;
  type Mode =
    enum Read, Write endenum;
  type Result =
    enum Accept, Reject endenum;

  signal open(File, Mode, User);
  signal close(File, User);
  signal answer(Result);

  process user(N);
    ...
  endprocess;
  process server(1);
    ...
  endprocess;
endsystem;
```

Figure 4: Architecture specification of the file system application

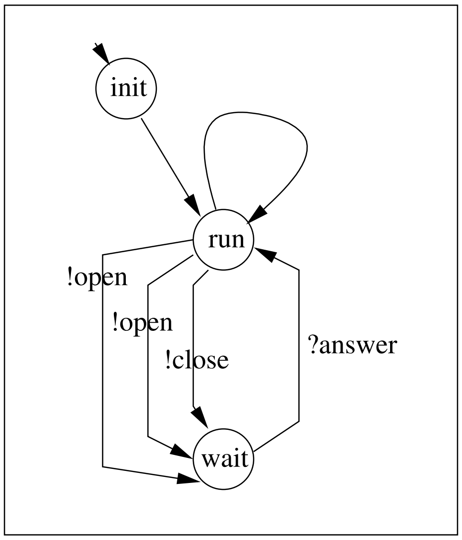
The specification is parameterised by two values: `N`, the number of users and `M`, the number of files in the filesystem. It contains the definition of several application specific data types, respectively, integer domains (`User` and `File`), and the enumerated lists (`Mode` and `Result`). The specification provides description of the communication signals `open`, `close` and `answer`. Finally, there is the description of the processes in the system, respectively `user` and `server`. At this point, let us notice the number of `user` process instances is defined to be `N`.

Figure 5 describe the `user` processes. Basically, after an initialisation phase they send non-deterministically `open` and `close` requests to the file server.

Let us focus on the initialisation transition. The first two assignements illustrate the conversion operators {-} allowing to convert (in this example) integer values to process identifiers and back.

In order to understand these operators, imagine that the process identifier values are couples of the form [process,integer]. For instance [user,7] is a valid process identifier for a `user` process instance. Now, the operator {integer} applied to a pid value will extract the integer component of it (for the example above, {integer} [user,7] = 7). Conversely, the operator {process} applied to an integer will construct a pid value (example, {user} 7 = [user,7]).

In the user example, the expression {`integer`} `self` extract the integer value corresponding to the `self` process identifier, hence, a number between 0 and `N-1`, depending on the instance which executes the transition. Conversely, the



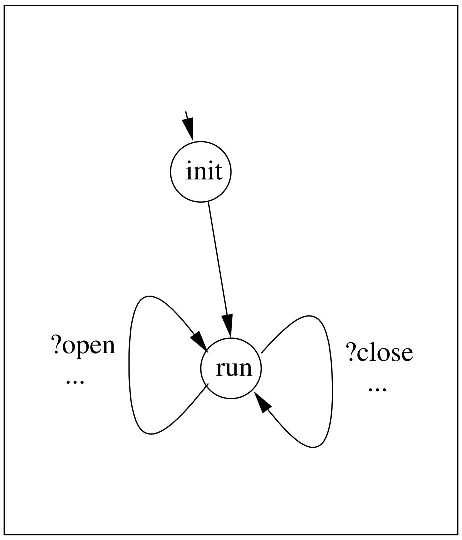

```
process user(N);
  var u User;
  var s pid;
  var f File;
  var r Result;
  state init #start;
    task u := {integer} self + 1;
    task s := {server} 0;
    task f := 1;
      nextstate run;
  endstate;
  state run;
    output open(f, Read, u) to s;
      nextstate wait;
    output open(f, Write, u) to s;
      nextstate wait;
    output close(f, u) to s;
      nextstate wait;
    task f := f % M + 1;
      nextstate -;
  endstate;
  state wait;
    input answer(r);
      nextstate idle;
  endstate;
endprocess;
```

Figure 5: Behavior specification of the user processes.

construction {`server`} `0` means the process identifier of the server instance.

Finally, remark also the construction `nextstate -;` which means remaining in the same state, where the transition begins.

The server process is sketched in the figure 6. In a few words, it stores the file system status in a two-dimensional array, which is modified accordingly opening and closing requests addressed by the users.



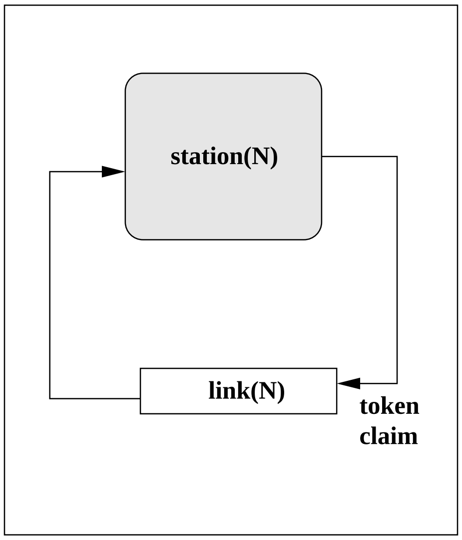

```
type FileStatus = enum
  Closed, Reading, Writing
endenum;
type FileControl =
  array [1 .. N] of FileStatus;
type FileSystem =
  array [1 .. M] of FileControl;
var fs FileSystem;
...
procedure FileAvailableForWrite
  fpar in f File,
    in u User,
    in fs FileSystem;
  returns boolean;
{#
  /* external C code */
  return ...;
#}
endprocedure;
...
state init #start ;
  task f := 1;
  while f <= M do
    task u := 1;
    while u <= N do
      task fs[f][u] := Closed;
      task u := u + 1;
    endwhile
    task f := f + 1;
  endwhile
  nextstate run;
endstate;
state run;
  ...
  input open(f, m, u);
    ...
    call FileAvailableForWrite(f,u,fs);
    ...
  input close(f, u);
    ...
endstate;
```

Figure 6: Behavior specification of the file server processes.

Let us notice the local definition of the `FileSystem` type, which is a two-dimensional array depending on both parameters `N` and `M`. Also, remark the definition of the external procedure *FileAvailableForWrite*, which implements the (possible complex) condition under that files are available for writing (e.g, *true if nobody is writing, except maybe the same user* `u`). Finally, let us observe the initialisation transition. It gives an example of using the `while` statement, here, in order to initialize the file system status.

4.3 Token Ring Protocol

The last example we consider is the leader election protocol in distributed ring networks. The architecture of the system is presented in figure 7.




```
system token;
  const N = 2;
  const D = 4;
  signal token();
  signal claim(integer, boolean);
  signalroute link(N)
    #lossy #delay[1,2]
    from station to station
    with token, claim;
  process station(N);
    ...
  endprocess;
endsystem;
```

Figure 7: Architecture specification of Token Ring Protocol.

The specification is parameterised with two values, respectively `N`, the number of stations in the ring and `D`, the average transit time for a message to make a complete tour of the ring. Two signal definitions are provided, respectively for `token` and `claim` signals. The later signal transports two values, an integer and a boolean one. The specification include the definition of `link` signalroutes, which model the communication paths between stations, and the definition of `station` processes.

For the sake of clarity, one can see (internal) signalroutes as specialised processes for the delivery of messages between *normal* processes. That is, messages transit through signalroute instances from one process instance to another process instances(s). The behaviour of signalroutes does not need to be explicitly defined by the user, it is implicitly defined by the following options:

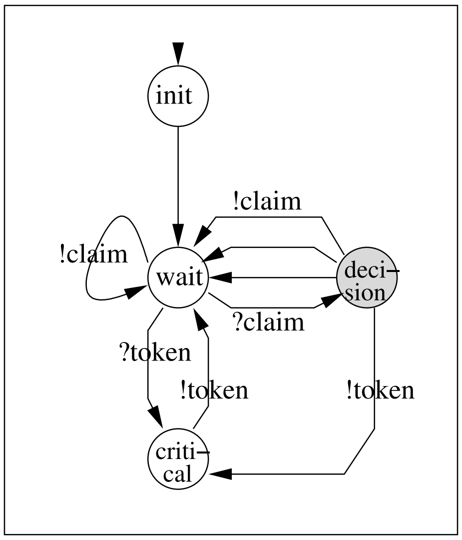
queueing policy denotes how the messages in transit are tackled by the signalroute. Two policies are available respectively, `#fifo`, order-preserving, uses a queue-based storage and `#multiset`, no order-preserving, uses a multiset storage.

delivering policy denotes how the messages are delivered. Three options are available here, respectively, `#peer`, which means delivery to the instance indicated in the output action using the `to` construct, `#unicast`, means delivery to one of the running instances and `#multicast`, means delivery to all running instances existing at the signalroute endpoint.

reliability could be either `#reliable`, which means that messages are not lost, or `#lossy`, which means that messages could be lost when transiting through the signalroute.

delaying policy denotes the delay associated with the signalroute and can be respectively, `#urgent`, means immediate (0-time) delivery, `#delay[a,b]`, means that any message entering the signalroute will eventually leave it after `a` and not later than `b` units of time, and `#rate[a,b]`, means that it takes between `a` and `b` units of time per message to be delivered by the signalroute.

In the token ring example, we considered that we have `N` signalroutes between stations, that all of them loose messages (#lossy) and delay messages between 1 and 2 units of time.




```
process station(N);
  var worried clock;
  var round boolean;
  var i integer;
  ...
  state init #start ;
      task i := {integer}self;
      task round := true;
      set worried := 0;
      nextstate wait;
  endstate;
  state wait;
    input token();
      reset worried;
      nextstate critical;
    when worried>=D;
      set worried := 0;
      output claim(i, round)
        via {link}i
        to {station} (i+1) % N;
      nextstate wait;
    input claim(k, t);
      nextstate decision;
  endstate;
  state critical;
      ...
      nextstate wait;
  endstate;
  state decision #unstable ;
    provided i < k;
      nextstate wait;
    ...
  endstate;
endprocess;
```

Figure 8: Behaviour specification of Token Ring stations.

Figure 8 present the station specification. Let us observe the definition of the `worried` clock, used later in specification to detect the token loss (i.e, if nothing happens during D time units, the timeout transition `provided worried` >= `D` will be executed).

Let us also observe the output action via `link` signalroutes. The intent is to send the `claim` message to next station, through the corresponding signalroute instance (from the `N` existing ones). This is done simply by using the `i`-th link instance, and defining the destination to be the next station `i+1 % N`, where `i` store the (integer) index of the running process. So, let us notice that the outputs encode the communication topology (here, the ring structure). That is, at the architectural level we have only indicated the number and the type of components (e.g, `N` stations and links) and the generic connection between them (e.g, stations communicates between them through links).

Finally, the last point we want to stress here is the `#unstable` option attached at the `decision` state. This makes the state to be *transient* (invisible) at execution. From an operational point of view, a process entering an unstable state must continue immediately by firing some transition at that state and so on, until a stable state will be reached. In other words, we will consider as atomic (uninterruptible) execution steps as sequences of transitions from one stable state to another stable state.

5 Discussion

Using the common language we were able to model the PGM (*Pretty Good Multicast*) protocol, one of the case studies considered in the ADVANCE project. The starting point is an SDL specification made by France Télécom R&D. We managed to manually *translate* this SDL specification into our common language and, to improve it in several respects. For example, we constructed an entirely parameterized specification with respect to the number of components involved in the protocol. Also, we simplified its control structure and also the data types definition.

Nevertheless, some key features are still missing in the common language definition. For example, there is a limited support to describe the system topology. For example, in the token ring protocol we specify that stations may communicate each other through link signalroutes but nothing is said about the *ring* structure. This information is *coded* inside station definition e.g, each one computing its successor based on its process identifier. We encounter the same problem in PGM, where the tree structure of the protocol is coded in the components definition (i.e, routers, receivers, etc.).

Another missing feature are quantifiers (existential, universal) in expressions. Quantifiers are really powerful operators allowing to write easily complex conditions, for example, *if there exists some process instance such that...*, or *if forall elements of an array holds...*, etc. At this time, we did not include them in the language because of pragmatic reasons, in particular, the difficulty of evaluation at runtime. Nevertheless, we plan to include them in the final version of the language.

From a practical point of view, we are currently upgrading the IF environment to the new description language. In particular, we build an *enumerative* *simulator*, implementing the semantics, and a *static analyzer*, implementing optimization and slicing techniques on the common language. We remember that both components exist for IF-1.0, however, they have to be completely re-designed and re-implemented for IF-2.0. On one hand, the simulator have to take into account dynamic process creation and destruction which implies, in turn, dynamic clock allocation, dynamic communication buffer allocation, dynamic pid allocation, etc. None of these points were adressed by the previous implementation which assumes an almost static state vector (except for buffer contents which may growth indefinitely). On the other hand, the static analyzer has to tackle the dynamic features too, the previous one relies on a fixed (static) communication scheme.