Modeling PGM with IF-2.0

VERIMAG
Centre Equation, 2 Avenue de Vignate
38610 Gières France

France Télécom R&D
Technopole Anticipa, 2 Avenue Pierre Marzin
22307 Lannion France

September 24, 2001

1 Overview

PGM stands for *Pragmatic General Multicast*. It is a telecommunication protocol defined for ensuring reliable multicast communications on top of another communication protocol like UDP. France Télécom R&D developed for its internal use an executable specification of this protocol written in the SDL language. Weaknesses of the simulation and validation tools around the SDL language (verification only possible with a finite-state space) sparked our interest in advanced verification techniques.

The goal of a multicast protocol is to ensure a data transfer from one application to many other applications. One outside application is linked to a sender PGM process. This sender PGM process communicates through network element PGM processes with many receiver PGM processes. Each of those receiver processes is connected to an outside application. For one communication session, the structure of processes is a tree. The problems of establishing a session, and creating the tree structure are beyond the scope of this study. But it is important to be able to describe a parametrized tree, for validating PGM in a maximum number of configurations. The PGM architecture is shown in figure 1.

The different processes exchange messages. The source outside application regularly sends data packets to the PGM sender. These data packets are to be transferred to the PGM receivers (and from there to the outside applications) as *Ordinary Data* (`ODATA`) messages, labelled with a packet number. The PGM protocol uses two complementary mechanisms for ensuring a reliable multicast.

First, the protocol uses a *negative acknowledgement* (`NAK`) mechanism. When a PGM receiver detects a loss of `ODATA` messages (for example because the packet numbers it has received are not consecutive), it sends a `NAK` message to the sender via the network elements with the number of the missing packet. When a network element process (or the sender process) receives a `NAK` message, it immediately sends back a *NAK ConFirmation* (NCF) message. If the receiver process does not receive this NCF message, it will repeat the NAK message after a waiting period. When the sender knows that a receiver needs a given packet number, it sends back a *Repair Data* (RDATA) message with the data packet.

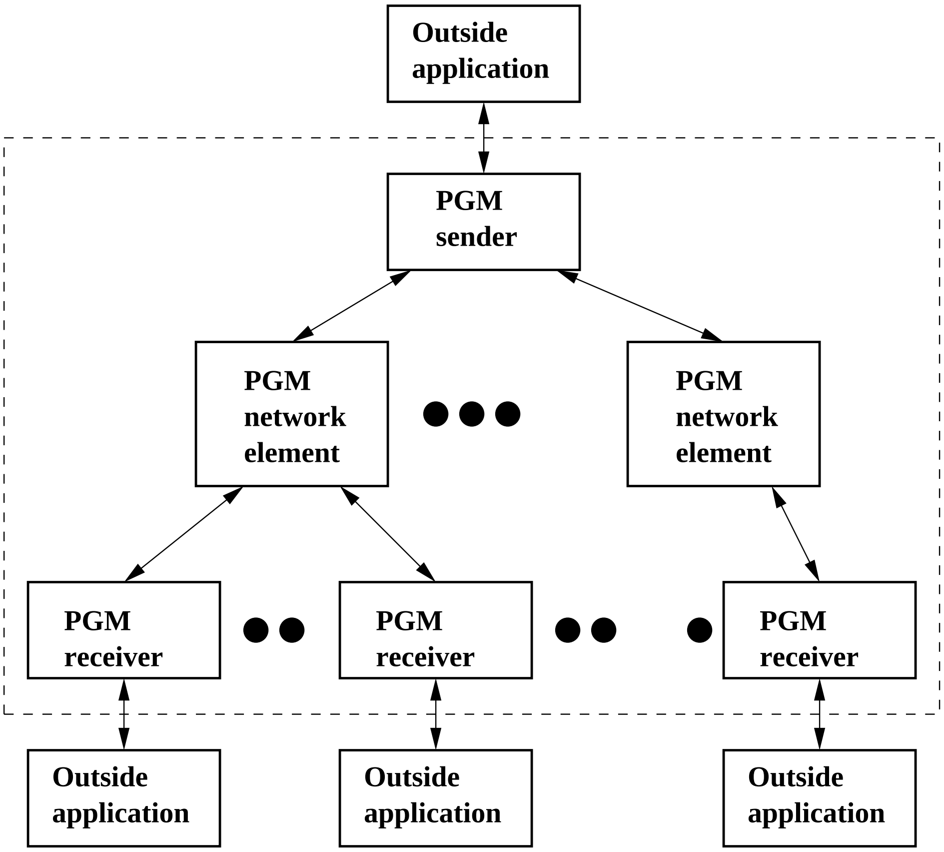

Figure 1: PGM Architecture.

The second mechanism is called a *sliding window*. The sender process cannot keep all the data packets in memory for re-transmission if needed. It keeps only a predefined number of those packets in memory, which constitutes a sliding window. Only the packets in this window can be requested. The window advances regularly, implying that some data packets will cease to be accessible. Hence, the sender process sends warning of the window advance to all receivers via network elements. These warning messages are contained in *Source Path Messages* (SPM) which are also used for establishing the session and setting up the tree structure. Those two mechanisms ensure that the numbers of acknowledgements will remain limited compared to the number of ODATA messages. For describing those mechanisms, the specification language needs to include elaborate datatypes, like lists, arrays and records for taking note of asked for rediffusions. The figure 2 illustrates the messages flowing downstream (ODATA, RDATA, NCF and SPM) and the NAK messages that flow upstream.

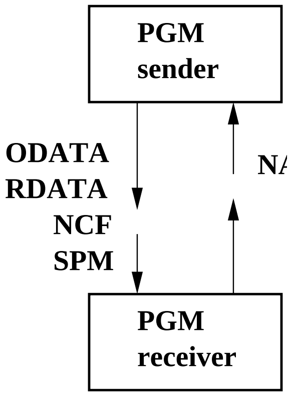

Figure 2: PGM Message Flow.

The PGM protocol also exploits *timing mechanisms* to enhance its reliability. Mainly, the receiver processes repeat the `NAK` messages regularly after waiting a given time. For ensuring this, dynamic processes can be created which are responsible for the management of `NAK` reemissions, following the firing of timers.

2 Modeling

The starting point for the modeling of PGM using the common language was the SDL description made by France Telecom. We tried to follow as much as possible the original description, however, we took advantage of the IF-2.0 features to improve it, when possible. The main modeling choices are discussed below.

2.1 The Architecture

France Telecom developers built a description of PGM by hard-coding the number of routers per source (at 3) and also the number of receivers per router (at 3, too). A plausible explanation of this could be the fact that specifying a generic PGM in SDL is rather complex. As a result,in the SDL specification, it is impossible to consider other PGM configurations except a source, 3 routers and 9 receivers .

With IF 2.0, it is easy to describe a parameterized tree with 1 sender process, `N` network processes and `N * M` receiver processes. Our specification contains three constants: `N` the number of routers, `M` the number of receivers for each router and, as a result, `P=N*M` the number of receiver instances running.

Although part of the system is created statically according to the values of `N` and `M` at the beginning of the program, the `router_nak` and `receiver_nak` processes are created and destroyed dynamiquely as needed.

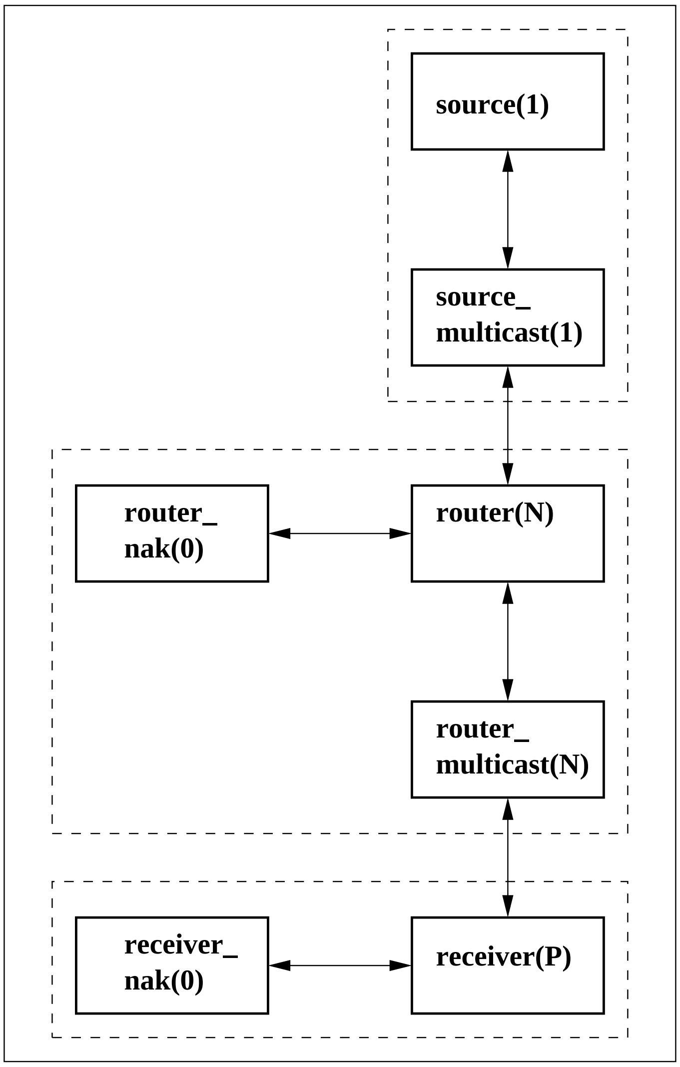

Figure 3: PGM Architecture.

```
system pgm;
    ...
  const N = 2; /* number of routeurs */
  const M = 2; /* number of receivers per router */
  const P = 4; /* number of receivers N * M */
    ...
  process source(1);
    ...
  process source_multicast(1);
    ...
  process router(N);
    ...
  process router_multicast(N);
    ...
  process router_nak(0);
    ...
  process receiver(P);
    ...
  process receiver_nak(0);
    ...
endsystem;
```

2.2 Data types

The SDL specification of PGM strongly relies on the use of SDL *strings* as the mean to implement dynamic lists. More concretely, the sender uses strings to store the content of the sliding window (ODATA list), the retransmission window (`RDATA` list) and the incoming window (fresh, incoming `ODATA` list). In addition, network elements and receiver instances use strings to represent data related to `nak` processes started dynamically by them.

Using IF-2.0, two solutions are available and will be explained below.

The first solution consists of modeling strings as abstract data types. The translation from SDL is straightforward, however, in order to (*semantically*) analyse the specification, an implementation of these types have to be provided. As an example, the string type `t_liste_odata` present in the SDL specification have to be translated in IF-2.0 as follows:

```
type t_liste_odata = abstract

  /* SDL specific operators */

  t_liste_odata empty_liste_odata();
  t_liste_odata mkstring(t_element_odata);
  t_liste_odata mkstring(integer);
  integer length(t_liste_odata);
  t_element_odata first(t_liste_odata);
  t_element_odata last(t_liste_odata);
  t_liste_odata concat(t_liste_odata,t_liste_odata);
  t_liste_odata substring(t_liste_odata,integer,integer);
```

```
  t_element_odata extract(t_liste_odata,integer);
  t_liste_odata modify(t_liste_odata,integer,t_element_odata);
  t_liste_odata remove(t_liste_odata, integer);

  /* user defined operators */

  integer get_index_odata(integer,t_liste_odata);
  t_element_odata get_element(integer,t_liste_odata);
  t_liste_odata del_data(integer,t_liste_odata);
  boolean doublon_obs(integer,t_liste_odata);

endabstract;
```

The second solution consists of modeling SDL strings as IF-2.0 strings. The advantage is that we have an operational implementation for free, however, the set of operators provided is fixed and cannot be extended by the user (as in SDL). Nevertheless, the set of operators provided for strings in IF-2.0 is richer than the SDL equivalent e.g, it contains searching, insertion and removing of elements. For example, the type `t_liste_odata` simply becomes:

```
type t_liste_odata = string[...]  of t_element_odata;
```

and one have to define only the `get_element` and `doublon_obs` operator (the two others, `get_index_odata` and `del_data` are predefined).

2.3 Process types

In general, the translation of SDL process types into IF processes was straightforward. Nevertheless, we found two major difficulties.

The first difficulty was the use of SDL procedures to structure the process behavior. Altough the intention was noble, the procedures were somehow badly defined and used. It is in general possible to expand them statically but, in this case, the IF code explodes (for example, the source use the `emission` procedure in at least 10 different contexts). The solution was to re-structure in some sense the behavior such that to reduce the explosion effect due to inlining. For example, the source process was entirely rewritten such that only one call to the `emission` procedure is made, while preserving completely its original behavior.

The second difficulty comes from parameterization. As we mention in the beginning, the original SDL description is partially generic: the numbers of routers and receivers being hard-coded. We completely redefine those parts, in order to obtain a fully generic specification. As an example, we include below the description of the `source_multicast` process, which deliver messages to all of the `N` routers, for any value of `N`.

```
process source_multicast(1);
    fpar parent pid;
```

```
  var sender pid;
  var offspring pid;

  var i_source pid;
  var i_seqnum integer;
  var i_spm_trail integer;
  var i_spm_inc integer;
  var i_txw_lead integer;
  var i_txw_trail integer;

  var i integer;

  state diffusion #start ;
    input nak(sender,i_seqnum,i_source);
      output nak(self,i_seqnum,i_source) to source0;
      nextstate -;
    input spm(sender,i_source,i_spm_trail,i_spm_inc,i_txw_lead);
      task i := 0;
      while i < N do
        output spm(self,i_source,i_spm_trail,i_spm_inc,i_txw_lead) to routeuri;
        task i := i + 1;
      endwhile
      nextstate -;
    input ncf(sender,i_seqnum,i_source);
      task i := 0;
      while i < N do
        output ncf(self,i_seqnum,i_source) to routeuri;
        task i := i + 1;
      endwhile
      nextstate -; input rdata(sender,i_seqnum,i_source,i_txw_trail);
      task i := 0;
      while i < N do
        output rdata(self,i_seqnum,i_source,i_txw_trail) to routeuri;
        task i := i + 1;
      endwhile
      nextstate -;
    input odata(sender,i_seqnum,i_txw_trail);
      task i := 0;
      while i < N do
        output odata(self,i_seqnum,i_txw_trail) to routeuri;
        task i := i + 1;
      endwhile
      nextstate -;
  endstate;

endprocess;
```

2.4 Timing

The reliability of PGM protocol is dependant on timing mechanisms. For example the source relies on several timers to multicast different packets with different priorities (ODATA, RDATA, SPM, etc...). Also, the receiver processes repeat the emission of NAK packets based on time. More precisely, for ensuring the correct functionning, nak processes are created and destroyed dynamically and each of them is responsible for NAK reemission according to the firing of several timers.

The usage of timers in PGM protocol, adheres well to the IF language. The notion of clock and the corresponding primitives existing in IF allows to simulate the behavior of SDL timers. Furthermore, the behavior of processes through time is controlled by explicit urgencies attached to transitions.

3 Experimentation

The first experiment was to simulate the SDL specification provided by FT R&D using ObjectGEODE. Although this tool has limited simulation capabilities, we were able to identify several problems at this level. For instance, we observed that internal lists (transmission window, retransmission window, etc.) always blow-up, for all reasonable scenarios. In fact, it is very difficult to find the relation between the window's sizes and the timing parameters controlling the transmission.

We translated the SDL specification into IF. As the IF model-checker is not fully operational,it cannot handle some of the features used inside the specification (i.e, dynamic strings). Nevertheless, we considered particular processes, such as `receiver_nak`, which were successfully analysed. For example, the `receiver_nak` has only few state variables but its behavior is very time-dependent: five different clocks are used to control the NAK re-emission. This process, in isolation, behaves according to the specifications.