

# Issues in Writing Large Models in VHDL

**Bernard Doray (doray@bnr.ca)**  
**Parviz Yousefpour (parviz@bnr.ca)**  
Bell-Northern Research Ltd  
P.O. Box 3511 Station C  
Ottawa, Ontario  
Canada, K1Y 4H7

## Abstract

*This paper describes an experience in modeling of a large telecommunication switching system for architecture exploration. It examines the modeling techniques and methodology used. It also explores the issues of: 1) model writing for ease of verification; 2) model testbench creation; 3) monitoring the progression of a simulation.*

*In this application an important aspect of the model was its flexibility for examining a variety of system configurations. The model developed supports significant reconfiguration without having to modify the VHDL model. An additional key aspect of the model is the ability of its testbench to “intelligently” extract relevant information out of the wealth of activity going on during simulation.*

## 1.0 Introduction

This paper describes the authors' experience in modeling of a large telecommunication system. It describes the modeling techniques and methodology adopted to make a VHDL model: easier to debug, easier to maintain, and easier to use. This methodology covers how to write the model, how to exercise the model, and how to extract meaningful information from simulation results.

### 1.1 The Switching System

The switching system modeled, called the Link Peripheral Processor (LPP), provides a high-speed, high-capacity platform for multiple, advanced data communication applications, such as Frame Relay, Integrated Packet Handler, and Common Channel Signaling No. 7 (CCS7). The model described here deals only with the Frame Relay application. Frame Relay Interface Units (FRIU) terminate either channelized or unchannelized DS-1s for this application. Each LPP can be provisioned with up to 36 of these FRIUs for a total of up to 36 DS-1 access lines. The LPPs can be networked together into a Metropolitan Area Service; the backbone network becomes a virtual network.

## 2.0 The Model

Writing a VHDL model of this complexity (40000 lines of VHDL), especially as a team effort, requires careful planning of how the model is going to be written. This maximizes the applicability of the model while minimizing the effort of developing that model: the goal is to write the right model and write it correctly.

### 2.1 Level of Abstraction

The selection of the level of abstraction for a model is an important step in the planning of a model. If the model is too detailed, simulation takes a long time to finish which limits the use of the model. In addition it makes it harder to maintain the more complex model. If the model is too abstracted, the conclusions that can be drawn from the model

are more limited; it can also be harder to include certain features of the system that are important. For example, if you model a microprocessor at the machine instruction level it is easy to model interrupt routines but if you model the microprocessor at the level of high-level language routines running on that processor, it becomes harder to model the interrupt mechanism.

Modeling the LPP at the gate level would have been beyond the capacity of most simulator/workstation combinations and even if it could have been simulated, the data generated would have been far too detailed for its targeted use. The goal was to see how various LPP configurations behaves for different traffic patterns so the level of abstraction chosen was queueing level simulation with abstract frames and fragments of frames being routed between the various sub-systems of the LPP. Even though frames are transmitted serially between some of these sub-systems in the LPP, the model deals with these transfers as larger aggregates of data but preserves the proper timing; this simplifies the processing of the data in the model and reduces the number of events (hence faster simulation).

## 2.2 Model Reconfigurability

For the Frame Relay Application, the LPP routes frames between end users connected to the system. These frames contain as part of their header a destination address; based on this address, the LPP forwards a frame to another DS1 user, or one of the other links connected to the LPP: one of the high-speed link used to network LPPs together or one of the slower links.

The intended use for the LPP model was to investigate the influence of various LPP configurations and traffic patterns on the behavior of the LPP. It was important to be able to evaluate various scenarios without having to modify the model because this would be an error-prone process. Each time the model is modified, some sanity checks would have to be performed to ensure that the LPP model still behaves properly; re-compilation can also take a significant time if many design units depend on the unit just recompiled.

The model has been written in such a way that it is easy to change the configuration of the LPP (number of each type of link interface boards). The number of boards instantiated is controlled with generate statements and top level generics; by changing the value of the generics it is possible to set-up the model for different configurations without having to modify the VHDL model. Some toolsets allow the setting of the top-level generics from the command line when starting the simulation but in some cases these values are propagated after elaboration; in these cases, the top-level entity/architecture pair has to be re-analyzed for each set of new generic values before running the simulation.

The model has also been written to use an ASCII file easily control the traffic pattern injected into the LPP; by editing this file it is possible to run a new simulation without re-analyzing the model. This makes it possible for people to use the model with minimum knowledge of how it fits together.

The value of various 'constant' parameters in the LPP (delays, size of buffers,...) were specified as deferred constants in a package. It makes it easier to modify the model during its validation because the constants are localized in a few packages; this also minimizes recompilation.

## 2.3 Methodology

When writing a large model a systematic approach to model writing and a set of guidelines for modeling style improve the quality of the final model. The benefits of this are:

- it favors the sharing of VHDL code between the model writers. (easier to write)
- it promotes the style consistency; this makes it easier for someone not familiar with the model to understand what it does. (easier to use)
- it maximizes the probability that the various parts of the model will work when they are connected together. (easier to debug)

To maximize sharing of VHDL code it is important to plan the design and partitioning of the model and adopt a library structure that favors the sharing of parts of the model between team members; re-use candidates must be put in packages where they can be directly accessed by other model writers. Making this VHDL code more flexible also increases the chances of re-use; for example in the LPP many different queues with different characteristics are used. In the model only one queue model was written with enough flexibility so it can be instantiated in all cases, each instance is tailored through the use of generics. The benefit is that only one queue model needs to be written and maintained.

The basic packages containing the definitions that are shared by all the subsystems (frame definition, basic resolution functions, print routines, etc.) were written first. The structure at the top level of the LPP was then defined. The work was then split by assigning the job of writing the architectures for these entities among the team members. The entities at the top-level became the contract between the team members to ensure that the various parts could be connected together. As more general packages were written by team members, they were made available to the other modelers.

The key to the coding guidelines was to agree on a consistent style for the model; this contributed to making the model more uniform and hence more understandable. The paper [1] describes one possible set of guidelines.

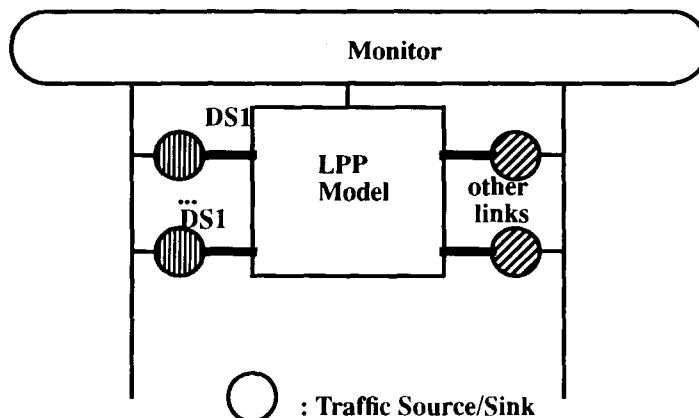
Some self-checking was included to make sure that some of the basic rules of communication between subsystems were followed; this allowed to catch some bugs that would have been difficult to find when the pieces of the model were assembled. The cards were individually tested with their own testbench before integrating them with the other sub-systems; where applicable they were also tested with their bus controller in a loop-back fashion. Print statements were included in the model to follow the propagation of a frame in the LPP; these can be turned on and off from a top-level generic for debugging purpose to pin-point possible bugs.

Good communication between team members was also important to maximize re-use and to keep track of changes. If the team is small and there is a lot of interaction between its members then informal discussions can be sufficient but for a larger team a structured approach with a code management system becomes necessary to track changes and avoid inconsistencies between model developers.

### 3.0 Testbench

Besides the model of the LPP itself, additional VHDL code was needed for simulation. The testbench serves two functions: it generates the frames to be injected into the LPP (Traffic Generator) and it monitors the state of the LPP (Monitor), checking the progression of simulation and reporting on the performance of the LPP. Figure 1 shows how these are tied with the LPP model. It should be noted that the effort to write the testbench for a large system can represent up to 40% ([2]) of the total modeling effort.

**Figure 1- LPP Model Top-Level**



### 3.1 Monitor

The monitor is a separate VHDL component that oversees the progression of the simulation of the LPP. The sub-systems in the LPP model report their changes in state to this monitor; the type of information sent is: frame received/sent/lost, current queue usage, or current bus usage. The monitor processes these messages to gather statistics about the state of the LPP and to do some checking on the progression of simulation; these checks ensure sanity of the model (e.g. a message cannot be received if it has not been sent!) and facilitate the debugging of the model. The data generated by the monitor is written to a file at the end of the simulation, it can be viewed using a graphics program (xgraph for example) or further processed by a separate program for analysis.

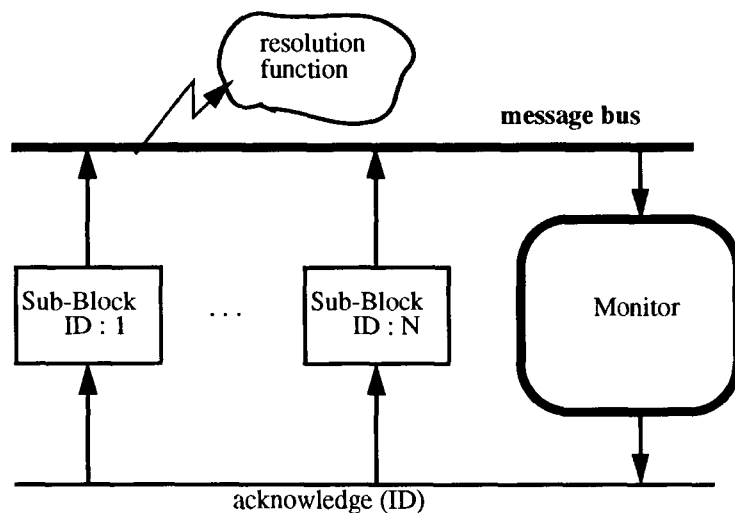
Having a centralized monitor has several advantages. It makes it easier to change the code that gathers statistics to generate a different report about the LPP (if enough information is passed from the sub-systems to the monitor). It is also possible to only enable the code that is required to gather the required statistics, this speeds up the simulation by minimizing the statistical processing performed. A special architecture can also be substituted to graphically display the progression of the simulation; this has been reported in [3].

Separating the monitor from the LPP model itself avoids cluttering the LPP model with the overhead code that gathers statistics. However, it has the disadvantage of slowing down simulation since the sub-systems have to send status messages to the monitor to gather statistics. The monitor had to be planned concurrently with the LPP model to ensure that the proper information is passed to the monitor by the model. This way of collecting statistics also influenced the LPP model since extra ports had to be added to propagate the messages to the testbench.

#### 3.1.1 Implementation

All the sub-systems in the LPP are connected to the monitor by a bus called the 'message bus' shown in figure 2; this bus has no equivalent in the real system, its use is to communicate state information to the monitor during simulation. When a sub-system wants to report information to the monitor, it builds a status message that contains the unique ID of this sub-system and the actual information that needs to be sent, it then calls the function `send_msg` to send this message. The details of the protocol used to send the information to the monitor are hidden in this function. The routine returns only when the message has been received by the monitor; it returns in zero time although some delta cycles elapse.

Figure 2: Connection to the Monitor



When the `send_msg` routine is called by a sub-system, it asserts the message for that sub-system onto the 'message bus' and waits for an acknowledge; more than one message can be asserted on the bus simultaneously since more

than one sub-system might send state information to the monitor at any given VHDL time step. The resolution function connected to the 'message bus' (shown in figure 3) lets through one of the messages asserted on the bus at a time. The monitor then receives this message filtered by the resolution function and process the information it contains. When this message is processed, the monitor acknowledges it by asserting the ID of the originating sub-system on the 'acknowledge signal'. When the routine that asserted the message sees its ID on the acknowledge line, it deasserts the message from the 'message bus' and returns control to the calling sub-system. The resolution function then propagates one of the other message asserted on the message bus to the monitor; this cycle continues until there are no more messages asserted for the current time step on the bus. The monitor processes the information in zero time so the VHDL time does not progress during statistical processing, but delta cycles do.

**Figure 3- Resolution Function for 'Message Bus'**

---

```

function stat_msg_resolver (
    stat : in stat_vector ) return stat_msg is
-- resolution function for status messages
-- sent to the test-bench
-- returns the first status message
-- with status not equal to none
begin
    if (stat'length = 0 ) then
        return null_stat_msg ;
    else
        stat_check_loop :
        for i in stat'Range loop
            if (stat(i).status /= NONE) then
                return stat(i) ;
            end if ;
        end loop stat_check_loop ;
        -- if all NONE then return a null_stat_msg
        return null_stat_msg ;
    end if;
end stat_msg_resolver ;

```

The monitor has no prior knowledge of how many sub-systems report to it for a given simulation run; the number of sub-systems for a given simulation depends on the configuration being simulated. The monitor has been written to adapt on the fly to the messages it receives; it creates the required data structures needed as it receives status messages from sub-systems. The data structures in the monitor are dynamic and rely on the use of VHDL pointers ([4]).

### 3.2 Traffic Generator

In the real system, the LPP connects to various systems (DS1 devices, other LPPs, etc.) that send frames to it and receive frames from it. For the purpose of simulation, traffic generators are connected to these links to mimic the working environment of the LPP. There are two modes of utilization for the traffic generator, each with different requirements:

- debug mode: in this mode, frames must be generated in a deterministic way for debugging of the model, so that the traffic can be easily traced through the system.
- experimentation mode: in this mode, the LPP is exercised with various realistic traffic patterns; it requires enough flexibility to describe, in a simple way, real traffic patterns.

The traffic generator is actually broken up into two parts since the links are bi-directional; the two parts are: the traffic source that generates frames for the LPP and the traffic sink that receives the frames from the LPP. For each link type, a separate traffic generator was written to accommodate for the differences in protocol but all the traffic generators shared the same code to generate the frame sent. Note that the minimum protocol characteristics were modeled in the LPP model and traffic sources, only the features that influence the way the frames are presented to the LPP were important; the focal point is how the LPP behaves under various traffic patterns, not the details of the interaction of the sources with the LPP.

The traffic generators are configured by generics and an ASCII file. This ASCII file characterizes the traffic to be generated; each field to be filled-in in a frame is specified by a probability distribution (Poisson, Normal, Constant, etc.) and the parameters required for the selected distribution. This allows a person running simulation to inject different traffic patterns with known statistical properties without having to modify the VHDL code for the model. The traffic generator could be modified to read a trace file to drive the traffic generator, this would make it possible to directly use real data traces to drive the model.

The traffic generator is flexible enough to be easily adapted and plugged in at various points in the model. Although this might have no real equivalent in the LPP system, it is useful for debugging part of the model when the model of the other sub-system are not yet available.

## 4.0 Tools

Finally, the last aspect that contributes to the quality of the VHDL model is the tools used. Some of the tools that were used or could have been used to simplify the writing of the VHDL model for the LPP are:

- **VHDL Sensitive Editor:** a language sensitive editor helped in using some of the VHDL guidelines for code consistency; by expanding tokens it is possible to obtain standard templates for comments and proper indentation for VHDL statements. This tool was developed at BNR as a VHDL mode running in EMACS.
- **Testbench Generator:** given an entity as an argument, it generates a testbench that instantiates that component under test; the user can then fill-in the VHDL code to exercise that component under test. This tool takes care of the mechanical part of writing a testbench.
- **Makefile Maker:** This tool generates makefiles that can be used to analyze all the design units in a model ([1]); having a makefile for each library simplifies the reanalysis of the model when changes are made.
- **Code Management System:** A code management system is an essential tool to keep track of changes in the VHDL code.
- **VHDL Browser:** This tool eases the task of understanding the model; its various query facilities allow to examine the model more easily.
- **Animation Tool:** Some form of support to include animation in the model is also a valuable aid in monitoring the progression of simulation.
- **Schematic Editor & Generator:** These tools can be of use to specify the structural part of a behavioral model and to view this structure; note that they are often of limited use with models using abstract data types.

## 5.0 Conclusion

The goals of this modeling effort was to write a model that is easier to debug, maintain, and use. This was achieved by providing a well defined VHDL methodology, a flexible test environment, and the tools to write/support the mod-

el. This resulted in a model that was within a few percents of the real system for the parameters measured by the monitor.

## 6.0 Acknowledgments

The authors would like to thank those who have in one way or the other helped in the project, especially the members of the LPP design team: Adrien Gobeil, Norm Lyon, Mike Adams, and Julian Cheesman for their technical contribution and direction. We also would like to thank Malcolm Coyne for his valuable comments.

## 7.0 References

- [1] Guidelines for Writing VHDL Models in a Team Environment; J. Bergeron; Proceedings of the Fall 1993 VIUF Conference.
- [2] Interactive Models and Testbenches in VHDL; H. Thaker and J. Bergeron; Proceedings of the Spring 1993 VIUF Conference.
- [3] Experiences in Real-Time Hardware-Software Co-Simulation; W. Loucks, B. Doray, and D. Agnew; Proceedings of the Spring 1993 VIUF Conference.
- [4] VHDL, System Design; Doug Perry; McGraw Hill 1990.