VHDL+/SpecC Comparisons – A Case Study

Andreas Gerstlauer
Shuqing Zhao
Daniel D. Gajski

Technical Report ICS-98-23
May 19, 1998

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-7063

gerstl@ics.uci.edu
szhao@ics.uci.edu
gajski@ics.uci.edu

Abstract

In this report, we make a comparison between two system level design languages—VHDL+ and SpecC—based on exploration of two worked examples modeled in both languages. We introduce the main features of the two language. Two examples, FIFO and FEC, are modeled with VHDL+ and SpecC. We conclude with a comparison between these two languages.

Contents

List of Figures

VHDL+/SpecC Comparisons – A Case Study

Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

In this report, we make a comparison between two system level design languages—VHDL+ and SpecC—based on exploration of two worked examples modeled in both languages. We introduce the main features of the two language. Two examples, FIFO and FEC, are modeled with VHDL+ and SpecC. We conclude with a comparison between these two languages.

1 Introduction

VHDL+ [1, 2] is a specification language proposed by ICL, Inc. to facilitate system level hardware design. VHDL+ is a superset of VHDL [4]. It provides a set of extensions for VHDL such as seperately defined interface specifications, timing abstraction using time ranges, behavioral abstraction using "choose" and "choice" statements, data abstraction with typeless messages, etc.

The examples in next two sections exhibit two VHDL+ key features: interface and activity. Designs modeled in VHDL+ are partitioned into design units and interfaces. A VHDL+ interface specification enables the user to specify a communication protocol, and to use that description to define the communication between design-unit specifications. The activity is a behavioral item which may be used to describe serial and parallel behavior. An important characteristic of an activity is that it is instanitiated and acts as a behavioral resource, e.g. an attempt to use an acitvity which is in use will cause queuing.

SpecC [5, 6, 7, 8, 9, 10] is a C based hardware/software co-design language proposed by UCI CADLAB. It is a superset of ANSI-C with added features like communication channel, hierarchy, concurrency, timing abstraction, etc. It is designed to be a single language that can be used in all hardware/software codesign process stages, supporting homogeneous approach in contrast to existing heterogeneous approaches. A SpecC program consists of a set of behavior, channel and interface declarations.

A behavior is a class consisting of a set of ports, a set of component instantiations, a set of private variables and functions, and a public main function. Through its ports, a behavior can be connected to other behaviors or channels in order to communicate. The functionality of a behavior is specified by its function declarations.

A channel is a class that encapsulates communication. It consists of a set of variables and functions, called methods, which define a communication protocol. A channel can be hierarchical, for example subchannels can be used to specify lower level communication.

An interface represents a flexible link between behaviors and channels. It consists of declarations of communication methods which will be defined in a channel.

It is hard to find the strength and weakness of a language without practical experiments. Therefore, in this report we take two worked examples in VHDL+, FIFO and FEC, from *Worked Examples* [3] and model them with SpecC to draw a comparison within the same context. In Section 2 and Section 3, we describe the FIFO and FEC example models in VHDL+ and SpecC. A conclusion of the comparison is drawn in Section 4.

2 FIFO Example

2.1 Example Description

The FIFO example shows in a client-server style communication enviornment how to model a FIFO buffer inside the server using VHDL+ and SpecC.

The client generates 5 bursts of 5 requests. Within a burst requests are seperatied by 20 ns. Each request is tagged with a sequence number which allows its progress to be monitored during the simulation.

The server receives the requests through its communication link to the client. It needs 20 ns to service each request. A FIFO is used to queue those requests not been serviced. If the FIFO is full the succeeding requests will be discarded.

2.2 VHDL+ model

An activity construct is used to model the FIFO buffer. The FIFO is then used in the simple client server model composed of a client entity, a server entity and a simple interface between these two entities as shown in Figure 1.

2.2.1 The Client Server Interface

The interface contains a simple message, called request, which has one data item called tag. The data item is used by a client to request a service from a server. The tag represents the data of a request and in this example it is used to show the sequence number of the requests during simulation.

2.2.2 The Client and the Server Entity

The client entity contains a single activity which generates requests in bursts. In fact, a VHDL process could be used here instead and produce the same simulation result because there is only one call to this activity.

The server entity has two processes and an activity. The activity is used to model the FIFO while the processes implement the queuing and servicing of requests.

2.2.3 The FIFO Activity

This activity is the basis of this example and is designed to illustrate two of the main properties of activities:

- Firstly, the variables of an activity can retain their values between calls like static variables in C.
- Secondly, only one instance of an activity can be active at a time. So when there are two different calls to the FIFO activity at the same time only one of them is allowed to proceed while the other is suspended until the activity is freed. This feature implicitly implements the mutual exclusiveness required to prevent the variables representing the FIFO from being updated simultaneously by the queuing process and the server process.

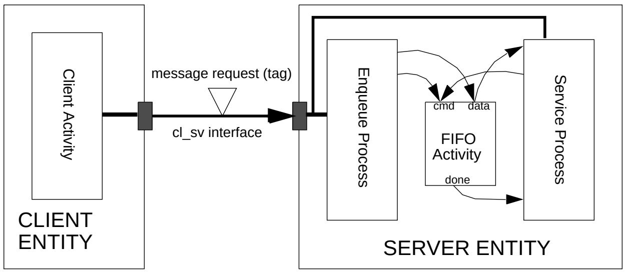

Figure 1: FIFO VHDL+ Model

2.2.4 Simulation Result Explanation

The simulation result is listed in Appendix A.2. Each line is divided into 4 columns. The first gives the current simulation time and the remaining columns contain output from the client activity, enqueue process and service process, respectively.

The first two lines (at 0 ns) show the enqueue process with an empty FIFO and the service process waiting for some service requests. On the third line the client process is seen to send a request which is then received by the queue process (line 4). Because there are no outstanding requests the service process acts on that request immediately (line 5) and the request is not queued. The simulation continues with the client making further requests and these are queued in the FIFO shown in the output from 5 ns to 150 ns. Eventually, because requests are produced at a higher rate than they are serviced, the FIFO fills up. At 155 ns, an error message "FIFO OVERRUN" is printed and the request is not queued. When the client stops making requests the service process continues to process the requests until the queue is emptied. Notice that requests 20, 23 and 24 have been lost due to FIFO overrun.

2.3 FIFO Example SpecC Model Description

As shown in Figure 2, the translation of the FIFO VHDL+ model to a corresponding SpecC model is straightforward. The CLIENT entity and CLIENT activity are mapped to one client "behavior" that is a unified construct used by SpecC to model structural and behavioral view of the design. The cl_sv interface is mapped to a channel to model the abstract communication protocol between client behavior and server behavior. The SERVER entity is mapped to a server composite behavior which is composed of an *enqueue* behavior and a *service* behavior and a *fifo* channel between them.

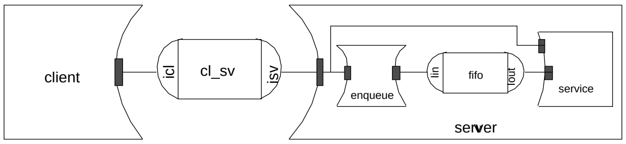

Figure 2: FIFO SpecC Model

2.3.1 Simulation Result Explanation

The simulation results are listed in Appendix B.2 and Appendix B.3. The results in Appendix B.3 were generated by exchanging the ordering of enqueue behavior (line 239) and service behavior (line 240). Because SpecC does not specify the order of execution of events happening at the same point in time, the simulation result will depend on the scheduler. In the former run, the enqueue behavior was executed first. It put one data item into the fifo and when the service behavior checks the status of the fifo by *getStatus()* (line 210) it got a non-empty flag. In the latter run, the service behavior was executed first and it got an empty flag. This kind of non-determinism introduced by SpecC may be helpful for modeling concurrent software but it remains a question for the determinism needed in modelling hardware as for example provieded by the delta time notion in VHDL or non-blocking assignment in Verilog.

3 Fec Example

3.1 Example Description

The Forward Error Correction (FEC) example basically models a sender and a receiver connected through a noisy bit channel. In order to account for the bit errors on the transmission medium the sender and receiver employ an error detection and correction scheme.

The sender is generating 4 bits of data (range 0 to 15) which are encoded into a 7 bit word by adding 3 error correction bits. This code word is then transmitted serially over the channel. The channel models the noise by randomly flipping bits with a probability of 0.1. On the receiver side, the words are decoded, corrected and finally compared with the expected value.

In addition, to show the effect of error correction, each data value is transmitted with and without FEC.

3.2 VHDL+ Model

The purpose of the VHDL+ FEC example is mainly to show the communication abstraction capabilities of VHDL+ via so called *interfaces* mentioned before. In VHDL+ an interface contains a set of routines (called *messages*) which can be called by the entities connecting to the interface. This way, the interface defines the communication whereas the computation is described separately in the entities connecting to the interface.

Most importantly, the example demonstrates how interfaces can provide multiple routines for communication at different abstraction levels. Internally, the interface is responsible for mapping between high-level and low-level communication. The idea is that in the course of moving from specification to implementation the entities get replaced by lower level descriptions and calls to high-level interface routines then get replaced successively by calls to the interface's lower-level routines.

3.2.1 Overview

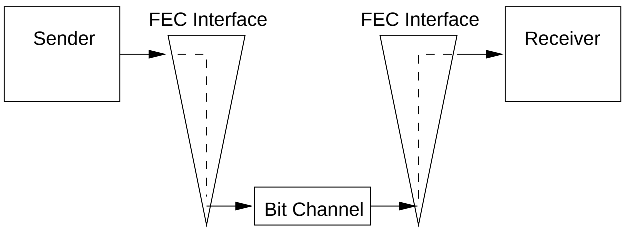

Figure 3: VHDL+ model.

The VHDL+ model of the FEC example is shown in Figure 3. The VHDL+ source code of the example can be found in Appendix C on page 22.

The VHDL+ FEC example consists of three entities connected by two instances of the FEC interface. On the sender side, the sender entity (lines 172-188) creates sample data values and sends them via a high-level call to the sender interface. The sender interface encodes the data and converts it into a serial bit stream. The bit channel entity (lines 194-250) receives the bit stream of the sender interface, and feeds the modified bit stream into the receiver interface. On the receiver side, the interface decodes the bit stream and performs error correction. Finally, the receiver entity (lines 256-308) receives the reconstructed high-level values.

3.2.2 Interface

The core of the VHDL+ model is the interface *FEC* (lines 87-166). The interface implements error correction, encoding/decoding and parallel to serial conversion. To move from higher to lower levels of abstraction higher-level routines recursively call lower-level message routines, i.e. higher-level messages are composed of lower-level messages. Note that the actual conversion performed at each level is specified as a set of functions in a separate package (lines 5-85). The example interface contains two of these message hierarchies: an error correcting stack (lines 95-135) and a non-error correcting stack (lines 137-166).

An important property of a VHDL+ interface is that only the decomposition of higher-level routines into lower-level ones has to specified. The reverse conversion from lower to higher levels is performed implicitly. For each conversion routine used in the composition an inverse routine has to be provided which gets called automatically by the VHDL+ compiler when needed.

A VHDL+ interface is always specified between two abstract ends (line 93) and the interface routines specify the direction in which the message is sent. Entities connect to a certain interface end and send or receive messages. The interface performs the transformation between messages at different levels.

3.3 SpecC Model

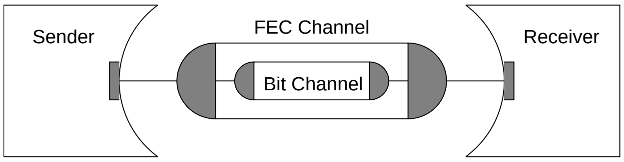

Figure 4: Basic SpecC model.

Figure 4 shows a straightforward transformation of the VHDL+ model of the FEC example into a corresponding SpecC version. The abstract communication of the VHDL+ interfaces is modeled by a SpecC channel. The hierarchical channel encapsulates encoding/decoding, error correction, parallel/serial conversion, and it contains a sub-channel which models the serial transmission of single bits over a noisy medium. The FEC channel provides interfaces to send and receive abstract data values using error correcting communication. The sender and receiver behaviors connect to the channel interface on both ends.

In order to exercise some more features of SpecC two variations of this basic model have been explored and actually implemented. The idea was to increase the flexibility in exploring different architectures by manually inlining the error correction and encoding/decoding parts of the channel into the sender and receiver behaviors.

3.3.1 Simple Model

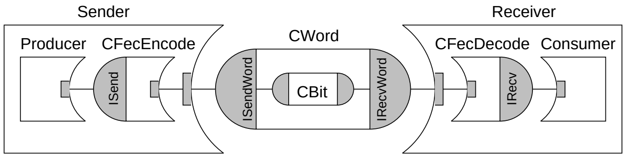

Figure 5: Simple SpecC model.

In a first SpecC realization of the FEC example the model of Figure 5 has been implemented (see Appendix D for the source code). The error correction and encoding/decoding part of the communication is inlined into the sender and receiver behaviors. Note, however, that only parts of the communication have been inlined. The channel *CWord* still provides some abstraction from the bit level in the sense that its interfaces (*ISendWord* and *IRecvWord*) send and receive whole bit words in parallel. This configuration will later allow (see Section 3.3.2) to easily replace the sender and receiver behaviors without having to modify the channel.

The channel source code which is common to both SpecC models can be found in Appendix D.1 on page 28. The channel *CWord* implements the parallel/serial conversion part of communication and interfaces to the bit channel contained in it. The bit channel *CBit* in turn implements interfaces *ISendBit* and *IRecvBit* to transmit single bits. It synchronizes sender and receiver and it models random bit noise on the transmission medium.

The *Sender* and *Receiver* behaviors connecting to the channel (Appendix D.2) each contain two parts:

- Behaviors *Producer* and *Consumer* which create/send and receive/compare abstract data values, respectively.
- Inlined error correction and encoding/decoding

communication parts in channels *CFecEncode* and *CFecDecode*.

3.3.2 Pipelined Model

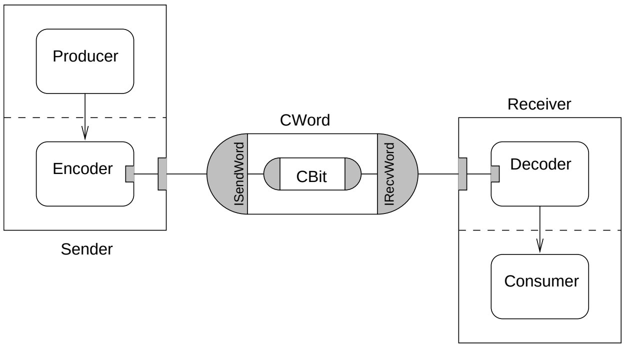

Figure 6: Pipelined SpecC model.

As a second version a pipelined design of the FEC example has been implemented in SpecC (Figure 6). The decoding/encoding and the error correction has been transformed into a behavior which executes overlapping with the producer and consumer behaviors in the sender and receiver pipelines, respectively. The source code for this example can be found in Appendix D.3.

This version shows how easily different architectures/designs can be explored. A lot of other design alternatives can be thought of. It is especially easy to modify this version such that the two pipelined behaviors in the sender and receiver run sequentially instead. This would effectively lead to a design which is functionally equivalent to the first alternative of Section 3.3.1. Note that all the changes only affect the sender and receiver behaviors. The channel itself is not modified.

3.3.3 Testbench

Finally, the testbench (Appendix D.4) instantiates *Sender* and *Receiver* behaviors and connects them via an instance of the channel. Since all versions of the sender and receiver behaviors conform to the same interfaces they can be exchanged easily and it is even possible to mix two different sender and receiver design styles. In fact, using the SpecC import mechanism as shown in the first lines of the testbench source file it is only necessary to import a different SpecC module in order to plug in a different version of a behavior.

	VHDL	VHDL+	SpecC
Scope Target	Hardware Simulation	Systems Simulation	Systems Synthesis
Behavioral Hierarchy	No	Partial	Yes
Structural Hierarchy	Yes	Yes	Yes
Concurrency	Yes	Yes	Yes
Communication Abstraction	No	Partial (message passing)	Yes
Synchronization	Yes	Yes	Yes
Exception Handling	No	No	Yes
Timing	Partial	Yes	Yes
State Machines	No	No	Yes

Table 1: Language comparison.

4 Conclusions

Using two example cases, this report has shown that a translation of VHDL+ models into a SpecC specification is straightforward and can be done easily. All VHDL+ concepts are supported by SpecC and can be mapped directly to corresponding SpecC descriptions.

Both languages are aimed at system modeling and design, providing the basis for executable system specifications at early stages of the design cycle. In comparison to traditional hardware description languages like VHDL [4], the requirements for system specification languages are higher [7].

Table 1 lists some of the required features along with the amount in which they are supported by VHDL, VHDL+ and SpecC. VHDL+ is a superset of VHDL and is targeted to throwaway prototyping for simulation and validation of a system level design concept early in the design process [1]. A VHDL+ description is translated by a compiler into a simulatable VHDL description. SpecC, on the other hand, is based on ANSI-C and builds the core of a complete co-design system [9]. In addition to simulation, it therefore explicitly supports automated synthesis of the specifications into hardware (e.g. synthesizable VHDL) or software (e.g. C code).

Basically, VHDL+ extends VHDL by the following three concepts:

- *Interfaces* which support abstract communication and separation of computation from communication.

- *Activities* which allow to explicitly specify serial and parallel behavioral hierarchy.

- Timing abstractions in the sense that timing constraints are supported.

The main VHDL+ contribution are certainly interfaces. As the previous sections have shown, the VHDL+ interface descriptions and their communication abstractions can be mapped directly to SpecC channels. However, VHDL+ interfaces are based on implicit message passing semantics. In addition, in VHDL+ interfaces only the decomposition of messages is specified by the user. The actual generation of a communication protocol is left to the compiler. On the one hand, VHDL+ interfaces facilitate easy modeling of many abstract communication relations that rely on these implicit semantics. On the other hand, interfaces are limited in their flexibility to model general communication behavior. In addition, interface semantics are on top of other language concepts and not integrated into VHDL. Due to the complexity of the translation from interface specification to implementation as can be seen on the code generated by the VHDL+ compiler it is hard to move from VHDL+ descriptions to implementations or synthesis.

In contrast, SpecC integrates channels (and interfaces) orthogonally and allows for the full language flexibility in modeling abstract and hierarchical communication behavior. For example, as seen in the FEC case, the message hierarchies of VHDL+ interfaces can be mapped to SpecC channels in exactly the same manner using a corresponding method hierarchy or hierarchical channels. Furthermore, as the design gets refined when moving towards implementation the channels get successively inlined. Moving parts of the communication protocol into the design behaviors automatically exposes communication at lower levels. In VHDL+, when moving from specification to implementation, high-level communication protocols have to be manually translated and inlined into the entities.

As mentioned above, in addition to interfaces VHDL+ supports behavioral hierarchy through *activities* and timing abstractions. However, the support for behavioral hierarchy is only partial. For example, no support for pipelining is included. On the other hand, SpecC allowed to easily model and explore different design styles by simply changing a couple of lines of code. Furthermore, other SpecC features which are not supported by VHDL+ are asynchronous exception handling (e.g. reset or interrupts) and state transitions for explicit modeling of finite state machines.

In Summary, VHDL+ provides some generally useful features. However, the new concepts are on top of the vast amount of standard VHDL features, increasing the complexity even further. VHDL+ is limited to simulation and not synthesizable. SpecC, on the other hand, supports the full set of features needed for systems design in a orthogonal and clean way.

5 References

[1] ICL Inc. *System Specification and Design Methodology.* SuperVISE Version 3.0, November 1997.

[2] ICL Inc. *Extensions to VHDL for System Specification.* VHDL+ Version 3.0, November 1997.

[3] ICL Inc. *Worked Examples.* SuperVISE Version 3.0, November 1997.

[4] IEEE Inc., N.Y. *IEEE Standard VHDL Language Reference Manual.* 1998.

[5] D. Gajski, J. Zhu, R. Dömer. *The SpecC+ Language.* University of California, Irvine, Technical Report ICS-TR-97-15, April 15, 1997.

[6] J. Zhu, R. Dömer, D. Gajski. *Syntax and Semantics of the SpecC+ Language.* University of California, Irvine, Technical Report ICS-TR-97-16, April 1997.

[7] D. Gajski, J. Zhu, R. Dömer. *Essential Issues in Codesign.* University of California, Irvine, Technical Report ICS-TR-97-26, June 1997.

[8] J. Zhu, R. Dömer, D. Gajski. *Syntax and Semantics of the SpecC Language.* Proceedings of the Synthesis and System Integration of Mixed Technologies 1997, Osaka, Japan, December 1997.

[9] D. Gajski, G. Aggarwal, E.-S. Chang, R. Dömer, T. Ishii, J. Kleinsmith, J. Zhu. *Methodology for Design of Embedded Systems.* University of California, Irvine, Technical Report ICS-TR-98-07, March 1998.

[10] R. Dömer, J. Zhu, D. Gajski. *The SpecC Language Reference Manual.* University of California, Irvine, Technical Report ICS-TR-98-13, March 1998.

A FIFO VHDL+ Source Listing

This appendix contains the VHDL+ source code, generated VHDL code and simulation result for the VHDL+ model of the FIFO example.

A.1 VHDL+ source code

```
--------------------------------------------------------------------------------
--                              FIFO Example                                  --
--------------------------------------------------------------------------------


---------------
-- INTERFACE --
---------------
interface cl_sv is
between CLIENT, SERVER;

   message request(tag : integer) is
   from CLIENT to SERVER;
   end;

end interface cl_sv;

------------
-- CLIENT----------------------------------------------------------------------
------------

entity client is
        interface port (i1 : CLIENT of cl_sv);
end client;

use std.textio.all;
architecture l1 of client is
   procedure write_string(L : inout LINE; VALUE : in string) is
   begin
      write(l, value);
   end;

   activity client_requester is
      variable l   : line;
      variable tag : integer := 0;
   serial
      ---------------------------------------------------
      -- send 5 bursts of requests separtated by 20 ns --
      ---------------------------------------------------
      for i in 1 to 5 loop
         ------------------------------------------------
         -- send a burst of requests separated by 5 ns --
         ------------------------------------------------
         for j in 1 to 5 loop
            tag := tag + 1;

            ------------------------------------------
            -- send a request. The data is just the --
            -- sequence number of the request       --
            ------------------------------------------
            send i1.request(tag);

            write(l, now, right, 7);
            write_string(l, " : service ");
            write(l, tag);
            write_string(l, " requested"); writeline(output, l);
            wait for 5 ns;
         end loop;
         wait for 20 ns;
```

```
      end loop;
      wait;
   end;
begin
   client_requester;
end l1;

------------
-- SERVER------------------------------------------------------------
------------

entity server is
      interface port (i1 : SERVER of cl_sv);
end server;

use std.textio.all;
architecture l1 of server is
   type q_cmd is (put,get);
   type int_array is array(integer range <>) of integer;

   constant FIFO_SIZE : integer := 11;

   procedure write_string(L : inout LINE; VALUE : in string) is
   begin
      write(l,value);
   end;

   procedure inc(i : inout integer) is
   begin
      i := (i + 1) mod FIFO_SIZE;
   end;

   ---------------------------------------------------------------------
   -- FIFO activity                                                 --
   ---------------------------------------------------------------------

   activity fifo(cmd : in q_cmd;
                 data : inout integer;
                 done : out boolean) is
      variable q_data      : int_array(0 to FIFO_SIZE-1);
      variable full        : boolean := false;
      variable empty       : boolean := true;
      variable write_to    : integer := 0;
      variable read_from   : integer := 0;
      variable transparent : boolean := false;
      variable l           : line;
   serial
      case cmd is
         when put =>
            if full then
               write(l,now,right,7); write_string(l," :                          ");
               write_string(l,"FIFO OVERRUN");
               writeline(output,l);
               write(l,now,right,7); write_string(l," :                          request ");
               write(l,data);
               write_string(l," NOT queued"); writeline(output,l);
               done := false;
            elsif transparent then
               transparent := false;
               done := true;
            else
               write(l,now,right,7); write_string(l," :                          request ");
               write(l,data);
               write_string(l," queued"); writeline(output,l);
               q_data(write_to) := data;
               inc(write_to);
```

```
            if write_to = read_from then
               full := true;
            end if;
            empty := false;
            done := true;
         end if;
      when get =>
         if empty then
            write(l,now,right,7); write_string(l," :                          ");
            write_string(l,"fifo empty");
            writeline(output,l);
            done := false;
            transparent := true;
         else
            data := q_data(read_from);
            inc(read_from);
            if read_from = write_to then
               empty := true;
            end if;
            done := true;
            full := false;
         end if;
   end case;
end;

begin
   enqueue : process
      variable tag  : integer;
      variable done : boolean;
      variable l    : line;
   begin
      ---------------------------
      -- wait for the next request --
      ---------------------------
      receive i1.request(tag);

      write(l,now,right,7); write_string(l," :                         request ");
      write(l,tag);
      write_string(l," received"); writeline(output,l);

      -------------------------------
      -- put the request into the FIFO. --
      -------------------------------
      fifo(put,tag,done);
   end process;

   service : process
      variable tag  : integer;
      variable done : boolean;
      variable l    : line;
   begin
      -------------------------------
      -- get a request from the FIFO. --
      -------------------------------
      fifo(get,tag,done);
      if not done then
         ---------------------------------------
         -- done is false when the FIFO is empty. --
         -- FIFO will now be transparent.         --
         ---------------------------------------
         write(l,now,right,7); write_string(l," :                                              ");
         write_string(l,"service queue empty"); writeline(output,l);

         ----------------------------------------------
         -- FIFO transparent so wait for the next request --
         -- directly, it will not be put in the FIFO.     --
```

```
      -----------------------------------------------------------
      receive i1.request(tag);
    end if;

    write(l,now,right,7); write_string(l,":                                                       service ");
    write(l,tag);
    write_string(l," started"); writeline(output,l);

    ------------------------
    -- service request --
    ------------------------
    --                  --
    wait for 20 ns;
    --                  --
    ------------------------

    write(l,now,right,7); write_string(l,":                                                       service ");
    write(l,tag);
    write_string(l," finished"); writeline(output,l);
  end process;
end l1;

----------------
-- TESTBENCH--------------------------------------------------------------
----------------

entity test_cl_sv is
end test_cl_sv;

architecture test of test_cl_sv is

component client
    interface port (i1 : CLIENT of cl_sv);
end component;

component server
    interface port (i1 : SERVER of cl_sv);
end component;

interface signal cl_sv_1 : cl_sv;

begin
    c : client
        interface map( i1 => cl_sv_1 );

    s : server
        interface map( i1 => cl_sv_1 );
end test;
```

A.2 Simulation Result

```
   0 NS :                         fifo empty
   0 NS :                                                   service queue empty
   0 NS : service 1 requested
   0 NS :                                                   service 1 started
   0 NS :                         request 1 received
   5 NS : service 2 requested
   5 NS :                         request 2 received
   5 NS :                         request 2 queued
  10 NS : service 3 requested
  10 NS :                         request 3 received
  10 NS :                         request 3 queued
  15 NS : service 4 requested
  15 NS :                         request 4 received
  15 NS :                         request 4 queued
  20 NS :                                                   service 1 finished
  20 NS :                                                   service 2 started
  20 NS : service 5 requested
  20 NS :                         request 5 received
  20 NS :                         request 5 queued
  40 NS :                                                   service 2 finished
  40 NS :                                                   service 3 started
  45 NS : service 6 requested
  45 NS :                         request 6 received
  45 NS :                         request 6 queued
  50 NS : service 7 requested
  50 NS :                         request 7 received
  50 NS :                         request 7 queued
  55 NS : service 8 requested
  55 NS :                         request 8 received
  55 NS :                         request 8 queued
  60 NS :                                                   service 3 finished
  60 NS :                                                   service 4 started
  60 NS : service 9 requested
  60 NS :                         request 9 received
  60 NS :                         request 9 queued
  65 NS : service 10 requested
  65 NS :                         request 10 received
  65 NS :                         request 10 queued
  80 NS :                                                   service 4 finished
  80 NS :                                                   service 5 started
  90 NS : service 11 requested
  90 NS :                         request 11 received
  90 NS :                         request 11 queued
  95 NS : service 12 requested
  95 NS :                         request 12 received
  95 NS :                         request 12 queued
 100 NS :                                                   service 5 finished
 100 NS :                                                   service 6 started
 100 NS : service 13 requested
 100 NS :                         request 13 received
 100 NS :                         request 13 queued
 105 NS : service 14 requested
 105 NS :                         request 14 received
 105 NS :                         request 14 queued
 110 NS : service 15 requested
 110 NS :                         request 15 received
 110 NS :                         request 15 queued
 120 NS :                                                   service 6 finished
 120 NS :                                                   service 7 started
 135 NS : service 16 requested
 135 NS :                         request 16 received
 135 NS :                         request 16 queued
 140 NS :                                                   service 7 finished
 140 NS :                                                   service 8 started
```

```
140 NS : service 17 requested
140 NS :                              request 17 received
140 NS :                              request 17 queued
145 NS : service 18 requested
145 NS :                              request 18 received
145 NS :                              request 18 queued
150 NS : service 19 requested
150 NS :                              request 19 received
150 NS :                              request 19 queued
155 NS : service 20 requested
155 NS :                              request 20 received
155 NS :                              FIFO OVERRUN
155 NS :                              request 20 NOT queued
160 NS :                                                          service 8 finished
160 NS :                                                          service 9 started
180 NS :                                                          service 9 finished
180 NS :                                                          service 10 started
180 NS : service 21 requested
180 NS :                              request 21 received
180 NS :                              request 21 queued
185 NS : service 22 requested
185 NS :                              request 22 received
185 NS :                              request 22 queued
190 NS : service 23 requested
190 NS :                              request 23 received
190 NS :                              FIFO OVERRUN
190 NS :                              request 23 NOT queued
195 NS : service 24 requested
195 NS :                              request 24 received
195 NS :                              FIFO OVERRUN
195 NS :                              request 24 NOT queued
200 NS :                                                          service 10 finished
200 NS :                                                          service 11 started
200 NS : service 25 requested
200 NS :                              request 25 received
200 NS :                              request 25 queued
220 NS :                                                          service 11 finished
220 NS :                                                          service 12 started
240 NS :                                                          service 12 finished
240 NS :                                                          service 13 started
260 NS :                                                          service 13 finished
260 NS :                                                          service 14 started
280 NS :                                                          service 14 finished
280 NS :                                                          service 15 started
300 NS :                                                          service 15 finished
300 NS :                                                          service 16 started
320 NS :                                                          service 16 finished
320 NS :                                                          service 17 started
340 NS :                                                          service 17 finished
340 NS :                                                          service 18 started
360 NS :                                                          service 18 finished
360 NS :                                                          service 19 started
380 NS :                                                          service 19 finished
380 NS :                                                          service 21 started
400 NS :                                                          service 21 finished
400 NS :                                                          service 22 started
420 NS :                                                          service 22 finished
420 NS :                                                          service 25 started
440 NS :                                                          service 25 finished
440 NS :                              fifo empty
440 NS :                                                          service queue empty
```

B FIFO SpecC Source Listings

This appendix contains the source listings for the SpecC model of the FIFO example.

B.1 SpecC source code

```
/*************************************************
* @title:        FIFO Example
* @author:       Shuqing Zhao
* @date:         3/19/98
**************************************************
** This example models a simple client-server system.
** It demonstrates the use of channel. Actually, there
** are two channels in this example. One is between
** the client and the server, the other is between the
** enqueue behavior and the service behavior.
*************************************************/

#include <stdio.h>
#include "clsv.h"

/*************************************************
**       channel declaration, between client & server
*************************************************/

/************************
** two ends of the channel
************************/
interface ICL {
        void send(int val);
};

interface ISV {
        int receive(void);
};

/************************
** channel implementation
************************/
channel cl_sv(void)
        implements ICL, ISV
{
        int message;
        bool valid;
        event activate;

        void send(int val){
                message = val;
                valid = true;
                notify(activate);
        }

        int receive(void){
                if(!valid)
                        wait(activate);
                valid = false;
                return message;
        }
};

/*************************************************
**       client behavior
*************************************************/
behavior client(ICL i1)
{
```

```
	void main()
	{
		int i,j, tag=0;

		/* send 5 bursts of requests separated by 20 ns */
		for(i=1; i<=5; i++){
			/* send a burst of requests separated by 5 ns */
			for(j=1; j<=5; j++){
				tag++;

				/* send a request. The data is just
				 the sequence number of the request. */
				i1.send(tag);
				printf("%3d ns: service %d requested\n",
				    now, tag);
				waitfor(5);
			};
			waitfor(20);
		};
	}
};

/************************************************
**	channel "fifo" inside server,
**	between enqueue & service
************************************************/

/************************
** two ends of the channel
************************/
interface Iinfifo{
	void put(int data);
};

interface Ioutfifo{
	int get(void);
	int getStatus(void);
};

/************************
** channel implementation
************************/
channel fifo
	implements Iinfifo, Ioutfifo
{

	int q_data[FIFO_SIZE];
	bool full=false;
	bool empty=true;
	int write_to=0;
	int read_from=0;
	bool transparent=false;
	bool done=false;
	bool occupied=false;
	event activate;

	void put(int data){
		if(occupied) wait(activate);
		occupied = true;

		if (full) {
			printf("%3d ns: \tFIFO OVERRUN\n", now);
			printf("%3d ns: \trequest %d NOT queued\n",
				now, data);
			done=false;
```

```
                }
                else if (transparent) {
                        transparent=false;
                        done=true;
                }
                else {
                        printf("%3d ns: \trequest %d queued\n", now, data);
                        q_data[write_to]=data;
                        inc(&write_to);
                        if(write_to==read_from)
                                full=true;
                        empty=false;
                        done=true;
                };

                occupied=false;
                notify(activate);
        }

        int get(void)
        {
                int data;

                if(occupied) wait(activate);
                occupied = true;

                if(empty){
                        printf("%3d ns: \tfifo empty\n", now);
                        done=false;
                        transparent=true;
                }
                else{
                        data=q_data[read_from];
                        inc(&read_from);
                        if(read_from==write_to)
                                empty=true;
                        done=true;
                        full=false;
                };

                occupied=false;
                notify(activate);

                return data;
        }

        int getStatus(){
                if(occupied) wait(activate);

                return done;
        }

};

/***************************************************
**      enqueue behavior
***************************************************/
behavior enqueue(ISV i1, Iinfifo i2)
{
        int tag;

        void main()
        {
                while(true){
                        tag=i1.receive();
```

```
                        printf("%3d ns: \trequest %d received\n", now, tag);
                        i2.put(tag);
                };
        }
};

/*****************************************************
**      service behavior
*****************************************************/
behavior service(ISV i1, Ioutfifo i2)
{
        int tag;
        int done;

        void main()
        {
                while(true){
                        tag=i2.get();
                        done=i2.getStatus();
                        if (!done) {
                                printf("%3d ns: \t\tservice queue empty\n",
                                        now);
                                tag=i1.receive();
                        };

                        printf("%3d ns: \t\tservice %d started\n", now, tag);

                        waitfor(20);

                        printf("%3d ns: \t\tservice %d finished\n", now, tag);
                };
        }
};

/*****************************************************
**      server behavior which uses enqueue & service
**      and connect them with fifo channel
*****************************************************/
behavior server(ISV i1)
{

        fifo f1;
        enqueue q1(i1, f1);
        service s1(i1, f1);

        void main(){
                par{
                        q1.main();
                        s1.main();
                }
        }
};

/*****************************************************
**      Test Bench
*****************************************************/
behavior test_cl_sv(void)
{
        cl_sv channel1;

        client c(channel1);
        server s(channel1);

        void main(void){
                par {
                        c.main();
```

```
                    s.main();
                }
            }
    };
```

B.2 Simulation Result 1

```
  0 ns: service 1 requested
  0 ns:                      fifo empty
  0 ns:                                               service queue empty
  0 ns:                                               service 1 started
  5 ns: service 2 requested
  5 ns:                      request 2 received
 10 ns: service 3 requested
 10 ns:                      request 3 received
 10 ns:                      request 3 queued
 15 ns: service 4 requested
 15 ns:                      request 4 received
 15 ns:                      request 4 queued
 20 ns: service 5 requested
 20 ns:                                               service 1 finished
 20 ns:                                               service 3 started
 20 ns:                      request 5 received
 20 ns:                      request 5 queued
 40 ns:                                               service 3 finished
 40 ns:                                               service 4 started
 45 ns: service 6 requested
 45 ns:                      request 6 received
 45 ns:                      request 6 queued
 50 ns: service 7 requested
 50 ns:                      request 7 received
 50 ns:                      request 7 queued
 55 ns: service 8 requested
 55 ns:                      request 8 received
 55 ns:                      request 8 queued
 60 ns: service 9 requested
 60 ns:                                               service 4 finished
 60 ns:                                               service 5 started
 60 ns:                      request 9 received
 60 ns:                      request 9 queued
 65 ns: service 10 requested
 65 ns:                      request 10 received
 65 ns:                      request 10 queued
 80 ns:                                               service 5 finished
 80 ns:                                               service 6 started
 90 ns: service 11 requested
 90 ns:                      request 11 received
 90 ns:                      request 11 queued
 95 ns: service 12 requested
 95 ns:                      request 12 received
 95 ns:                      request 12 queued
100 ns: service 13 requested
100 ns:                                               service 6 finished
100 ns:                                               service 7 started
100 ns:                      request 13 received
100 ns:                      request 13 queued
105 ns: service 14 requested
105 ns:                      request 14 received
105 ns:                      request 14 queued
110 ns: service 15 requested
110 ns:                      request 15 received
110 ns:                      request 15 queued
120 ns:                                               service 7 finished
120 ns:                                               service 8 started
135 ns: service 16 requested
135 ns:                      request 16 received
135 ns:                      request 16 queued
140 ns: service 17 requested
140 ns:                                               service 8 finished
140 ns:                                               service 9 started
140 ns:                      request 17 received
```

```
140 ns:                       request 17 queued
145 ns: service 18 requested
145 ns:                       request 18 received
145 ns:                       request 18 queued
150 ns: service 19 requested
150 ns:                       request 19 received
150 ns:                       request 19 queued
155 ns: service 20 requested
155 ns:                       request 20 received
155 ns:                       request 20 queued
160 ns:                                                 service 9 finished
160 ns:                                                 service 10 started
180 ns:                                                 service 10 finished
180 ns:                                                 service 11 started
180 ns: service 21 requested
180 ns:                       request 21 received
180 ns:                       request 21 queued
185 ns: service 22 requested
185 ns:                       request 22 received
185 ns:                       request 22 queued
190 ns: service 23 requested
190 ns:                       request 23 received
190 ns:                       FIFO OVERRUN
190 ns:                       request 23 NOT queued
195 ns: service 24 requested
195 ns:                       request 24 received
195 ns:                       FIFO OVERRUN
195 ns:                       request 24 NOT queued
200 ns: service 25 requested
200 ns:                                                 service 11 finished
200 ns:                                                 service 12 started
200 ns:                       request 25 received
200 ns:                       request 25 queued
220 ns:                                                 service 12 finished
220 ns:                                                 service 13 started
240 ns:                                                 service 13 finished
240 ns:                                                 service 14 started
260 ns:                                                 service 14 finished
260 ns:                                                 service 15 started
280 ns:                                                 service 15 finished
280 ns:                                                 service 16 started
300 ns:                                                 service 16 finished
300 ns:                                                 service 17 started
320 ns:                                                 service 17 finished
320 ns:                                                 service 18 started
340 ns:                                                 service 18 finished
340 ns:                                                 service 19 started
360 ns:                                                 service 19 finished
360 ns:                                                 service 20 started
380 ns:                                                 service 20 finished
380 ns:                                                 service 21 started
400 ns:                                                 service 21 finished
400 ns:                                                 service 22 started
420 ns:                                                 service 22 finished
420 ns:                                                 service 25 started
440 ns:                                                 service 25 finished
440 ns:                       fifo empty
440 ns:                                                 service queue empty
```

B.3 Simulation Result 2

```
   0 ns: service 1 requested
   0 ns:                     request 1 received
   0 ns:                     request 1 queued
   0 ns:                                          service 1 started
   5 ns: service 2 requested
   5 ns:                     request 2 received
   5 ns:                     request 2 queued
  10 ns: service 3 requested
  10 ns:                     request 3 received
  10 ns:                     request 3 queued
  15 ns: service 4 requested
  15 ns:                     request 4 received
  15 ns:                     request 4 queued
  20 ns: service 5 requested
  20 ns:                                          service 1 finished
  20 ns:                                          service 2 started
  20 ns:                     request 5 received
  20 ns:                     request 5 queued
  40 ns:                                          service 2 finished
  40 ns:                                          service 3 started
  45 ns: service 6 requested
  45 ns:                     request 6 received
  45 ns:                     request 6 queued
  50 ns: service 7 requested
  50 ns:                     request 7 received
  50 ns:                     request 7 queued
  55 ns: service 8 requested
  55 ns:                     request 8 received
  55 ns:                     request 8 queued
  60 ns: service 9 requested
  60 ns:                                          service 3 finished
  60 ns:                                          service 4 started
  60 ns:                     request 9 received
  60 ns:                     request 9 queued
  65 ns: service 10 requested
  65 ns:                     request 10 received
  65 ns:                     request 10 queued
  80 ns:                                          service 4 finished
  80 ns:                                          service 5 started
  90 ns: service 11 requested
  90 ns:                     request 11 received
  90 ns:                     request 11 queued
  95 ns: service 12 requested
  95 ns:                     request 12 received
  95 ns:                     request 12 queued
 100 ns: service 13 requested
 100 ns:                                          service 5 finished
 100 ns:                                          service 6 started
 100 ns:                     request 13 received
 100 ns:                     request 13 queued
 105 ns: service 14 requested
 105 ns:                     request 14 received
 105 ns:                     request 14 queued
 110 ns: service 15 requested
 110 ns:                     request 15 received
 110 ns:                     request 15 queued
 120 ns:                                          service 6 finished
 120 ns:                                          service 7 started
 135 ns: service 16 requested
 135 ns:                     request 16 received
 135 ns:                     request 16 queued
 140 ns: service 17 requested
 140 ns:                                          service 7 finished
 140 ns:                                          service 8 started
```

```
140 ns:                          request 17 received
140 ns:                          request 17 queued
145 ns: service 18 requested
145 ns:                          request 18 received
145 ns:                          request 18 queued
150 ns: service 19 requested
150 ns:                          request 19 received
150 ns:                          request 19 queued
155 ns: service 20 requested
155 ns:                          request 20 received
155 ns:                          FIFO OVERRUN
155 ns:                          request 20 NOT queued
160 ns:                                                      service 8 finished
160 ns:                                                      service 9 started
180 ns:                                                      service 9 finished
180 ns:                                                      service 10 started
180 ns: service 21 requested
180 ns:                          request 21 received
180 ns:                          request 21 queued
185 ns: service 22 requested
185 ns:                          request 22 received
185 ns:                          request 22 queued
190 ns: service 23 requested
190 ns:                          request 23 received
190 ns:                          FIFO OVERRUN
190 ns:                          request 23 NOT queued
195 ns: service 24 requested
195 ns:                          request 24 received
195 ns:                          FIFO OVERRUN
195 ns:                          request 24 NOT queued
200 ns: service 25 requested
200 ns:                                                      service 10 finished
200 ns:                                                      service 11 started
200 ns:                          request 25 received
200 ns:                          request 25 queued
220 ns:                                                      service 11 finished
220 ns:                                                      service 12 started
240 ns:                                                      service 12 finished
240 ns:                                                      service 13 started
260 ns:                                                      service 13 finished
260 ns:                                                      service 14 started
280 ns:                                                      service 14 finished
280 ns:                                                      service 15 started
300 ns:                                                      service 15 finished
300 ns:                                                      service 16 started
320 ns:                                                      service 16 finished
320 ns:                                                      service 17 started
340 ns:                                                      service 17 finished
340 ns:                                                      service 18 started
360 ns:                                                      service 18 finished
360 ns:                                                      service 19 started
380 ns:                                                      service 19 finished
380 ns:                                                      service 21 started
400 ns:                                                      service 21 finished
400 ns:                                                      service 22 started
420 ns:                                                      service 22 finished
420 ns:                                                      service 25 started
440 ns:                                                      service 25 finished
440 ns:                          fifo empty
440 ns:                                                      service queue empty
```

C FEC VHDL+ Source Listing

This appendix contains the source code for the VHDL+ model of the Forward Error Correction (FEC) example.

```
-----------------------------------------------------------------------------
-- PACKAGE nibble_types -----------------------------------------------------
-----------------------------------------------------------------------------

package fec_types is
   subtype small_int is natural range 0 to 15;
   subtype nibble is bit_vector(3 downto 0);
   subtype code_word is bit_vector(6 downto 0);

   function nibbler(arg : small_int) return nibble;
   function inverse_nibbler(arg : nibble) return small_int;

   function coder(arg : nibble) return code_word;
   function inverse_coder(arg : code_word) return nibble;
end fec_types;

package body fec_types is
   function nibbler(arg : small_int) return nibble is
      variable result : nibble;
      variable temp : small_int;
   begin
      temp := arg;
      for i in 0 to 3 loop
         if (temp mod 2) = 1 then
            result(i) := '1';
         else
            result(i) := '0';
         end if;

         if temp > 0 then
            temp := temp / 2;
         else
            temp := (temp - 1) / 2;
         end if;
      end loop;
      return result;
   end;

   function inverse_nibbler(arg : nibble) return small_int is
      variable result : small_int;
   begin
      result := 0;
      for i in 3 downto 0 loop
         result := result*2;
         if arg(i) = '1' then
           result := result + 1;
         end if;
      end loop;
      return result;
   end;

   function coder(arg : nibble) return code_word is
      variable result : code_word := "0000000";
   begin
      result(3 downto 0) := arg;
      result(4) := arg(2) xor arg(1) xor arg(0);
      result(5) := arg(3) xor arg(2) xor arg(1);
      result(6) := arg(3) xor arg(1) xor arg(0);
      return result;
   end;

   function inverse_coder(arg : code_word) return nibble is
```

```
      variable result : nibble;
      variable s : bit_vector(0 to 2);
      variable v : code_word;
   begin
      v := arg;
      s(0) := arg(6) xor arg(3) xor arg(1) xor arg(0);
      s(1) := arg(5) xor arg(3) xor arg(2) xor arg(1);
      s(2) := arg(4) xor arg(2) xor arg(1) xor arg(0);
      case s is
         when "000" => v := arg xor "0000000"; -- e(0);
         when "100" => v := arg xor "1000000"; -- e(1);
         when "010" => v := arg xor "0100000"; -- e(2);
         when "001" => v := arg xor "0010000"; -- e(3);
         when "110" => v := arg xor "0001000"; -- e(4);
         when "011" => v := arg xor "0000100"; -- e(5);
         when "111" => v := arg xor "0000010"; -- e(6);
         when "101" => v := arg xor "0000001"; -- e(7);
      end case;
      result := v(3 downto 0);
      return result;
   end;

end fec_types;

--------------------
-- INTERFACE FEC------------------------------------------------------------------
--------------------
use work.fec_types.all;

interface FEC is
between SENDER, RECEIVER;

   ------------------------------------------
   -- forward error correcting message stack --
   ------------------------------------------

   message fec_mess(data : small_int) is
      from SENDER to RECEIVER;
   composition
   serial
      nibble_mess(nibbler(data));
   end;

   message nibble_mess(data : nibble) is
      from SENDER to RECEIVER;
   composition
   serial
      code_mess(coder(data));
   end;

   message code_mess(data : code_word) is
      from SENDER to RECEIVER;
   composition
   serial
      bit_mess(data(0));
      pause for 1 ns;
      bit_mess(data(1));
      pause for 1 ns;
      bit_mess(data(2));
      pause for 1 ns;
      bit_mess(data(3));
      pause for 1 ns;
      bit_mess(data(4));
      pause for 1 ns;
      bit_mess(data(5));
      pause for 1 ns;
```

```
      bit_mess(data(6));
      pause for 1 ns;
   end;

   message bit_mess(data : bit) is
      from SENDER to RECEIVER;
   end;

   -----------------------------------------
   -- non error correcting message stack --
   -----------------------------------------

   message nec_mess(data : small_int) is
      from SENDER to RECEIVER;
   composition
   serial
      nibble1_mess(nibbler(data));
   end;

   message nibble1_mess(data : nibble) is
      from SENDER to RECEIVER;
   composition
   serial
      bit1_mess(data(0));
      pause for 1 ns;
      bit1_mess(data(1));
      pause for 1 ns;
      bit1_mess(data(2));
      pause for 1 ns;
      bit1_mess(data(3));
      pause for 1 ns;
   end;

   message bit1_mess(data : bit) is
      from SENDER to RECEIVER;
   end;

end interface FEC;

-------------
-- SENDER----------------------------------------------------------------
-------------

entity sender is
        interface port (o1 : SENDER of FEC);
end sender;

use std.textio.all;

architecture l1 of sender is
begin
   process
   begin
      for i in 0 to 15 loop
         wait for 10 ns;
         send o1.fec_mess(i);
         send o1.nec_mess(i);
      end loop;
   end process;
end l1;

--------------
-- CHANNEL---------------------------------------------------------------
--------------

entity channel is
```

```
        interface port (i1 : RECEIVER of FEC;
                        o1 : SENDER of FEC);
end channel;

use std.textio.all;

architecture l1 of channel is
   procedure write_string(L : inout LINE; VALUE : in string) is
   begin
      write(l, value);
   end;

   activity bit_noise is
      variable data : bit;
      variable error_bit : bit;
   serial
      receive i1.bit_mess(data);
      choose
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '1';
      end choose;
      send o1.bit_mess(data xor error_bit);
   end activity;

   activity bit1_noise is
      variable data : bit;
      variable error_bit : bit;
   serial
      receive i1.bit1_mess(data);
      choose
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '0';
         => error_bit := '1';
      end choose;
      send o1.bit1_mess(data xor error_bit);
   end activity;

begin
   bit_noise;
   bit1_noise;
end l1;

-------------------------------------------------------------
-- RECEIVER------------------------------------------------------------------------------------------------
-------------------------------------------------------------

entity receiver is
        interface port (i1 : RECEIVER of FEC);
end receiver;

use std.textio.all;
```

```
architecture l1 of receiver is

   procedure write_string(L : inout LINE; VALUE : in string) is
   begin
      write(l, value);
   end;

   activity fec_receive is
      variable num_messages : integer := 0;
      variable ri : small_int;
      variable l  : line;
   serial
      for i in 0 to 15 loop
         receive i1.fec_mess(ri);
         num_messages := num_messages + 1;
         if ri /= i then
            write(l,num_messages,right,5);
            write_string(l,"␣error␣in␣fec_mess␣:␣received␣");
            write(l,ri,right,2);
            write_string(l,"␣-␣expected␣"); write(l,i,right,2);
            writeline(output,l);
         end if;
      end loop;
   end activity;

   activity nec_receive is
      variable num_messages : integer := 0;
      variable ri : small_int;
      variable l  : line;
   serial
      for i in 0 to 15 loop
         receive i1.nec_mess(ri);
         num_messages := num_messages + 1;
         if ri /= i then
            write(l,num_messages,right,5);
            write_string(l,"␣error␣in␣nec_mess␣:␣received␣");
            write(l,ri,right,2);
            write_string(l,"␣-␣expected␣"); write(l,i,right,2);
            writeline(output,l);
         end if;
      end loop;
   end activity;

begin
   fec_receive;
   nec_receive;
end l1;

----------------
-- TESTBENCH--------------------------------------------------------------
----------------

entity test_fec is
end test_fec;

architecture test of test_fec is

component sender
     interface port (o1 : SENDER of FEC);
end component;

component channel
     interface port (i1 : RECEIVER of FEC;
                     o1 : SENDER of FEC);
end component;
```

```
component receiver
    interface port (i1 : RECEIVER of FEC);
end component;

interface signal fec_1 : FEC;
interface signal fec_2 : FEC;

begin
    s : sender
        interface map( o1 => fec_1 );

    c : channel
        interface map( i1 => fec_1,
                       o1 => fec_2 );

    r : receiver
        interface map( i1 => fec_2 );
end test;
```

D FEC SpecC Source Listings

This appendix contains the source listings for the different SpecC models of the FEC example.

D.1 Channel

The following two sections list the interfaces and the implementation of the channels *CWord* and *CBit*. Channel *CWord* is the common channel connecting the different sender and receiver behaviors at the word level. Internally, it performs parallel/serial conversion and it conatains the bit level channel *CBit* which in turn transfers single bits over a noisy medium.

D.1.1 Interfaces - `IBit.sc`

```
/*****************************************************/
/* Interfaces at bit level                           */
/*****************************************************/

// 3/16/98, A. Gerstlauer

// Bit vector types
typedef unsigned bit[3:0] nibble;
typedef unsigned bit[6:0] word;


// --- Transmit bit vectors

interface IRecvWord
{
  word    recvWord();
  nibble  recvNibble();
};

interface ISendWord
{
  void sendWord(word data);
  void sendNibble(nibble data);
};


// --- Transmit bits

interface IRecvBit
{
  bool recvBit();
};


interface ISendBit
{
  void sendBit(bool data);
};
```

D.1.2 Implementation - `CBit.sc`

```
/*****************************************************/
/* Bit-level channels                                */
/*****************************************************/

// 3/16/98, A. Gerstlauer


import "IBit";
```

```
#include <stdlib.h>



// Single-bit, serial channel with noise
channel CBit
  implements ISendBit, IRecvBit
{
  bool  bitReady;
  bool  bitValue;
  event bitWakeup;

  // Generate noise
  bool noise()
  {
    return ((nrand48() % 10) == 9);
  }

  void sendBit(bool data)
  {
    bitValue = data ^ noise();
    bitReady = true;
    notify(bitWakeup);
  }

  bool recvBit()
  {
    if (!bitReady) {
      wait(bitWakeup);
    }
    bitReady = false;
    return bitValue;
  }

};


// Channel for transmitting whole bit vectors
channel CWord
  implements ISendWord, IRecvWord
{
  CBit  bitChannel;

  void sendWord(word data)
  {
    int i;

    for (i = 0; i < 7; i++) {
      bitChannel.sendBit(data[i]);
      waitfor(10);
    }
  }

  word recvWord()
  {
    int  i;
    word result;

    for (i = 0; i < 7; i++)
      result[i] = bitChannel.recvBit();

    return result;
  }

  void sendNibble(nibble data)
  {
```

```
    int i;

    for (i = 0; i < 4; i++) {
      bitChannel.sendBit(data[i]);
      waitfor(10);
    }
  }

  nibble recvNibble()
  {
    int    i;
    nibble result;

    for (i = 0; i < 4; i++)
      result[i] = bitChannel.recvBit();

    return result;
  }
};
```

D.2 Simple Sender & Receiver

In this section the first, sequential version of *Sender* and *Receiver* behaviors is listed. Both behaviors each contain a sub-behavior to produce/consume data values and an encoder/decoder channel to perform error correcting communication.

D.2.1 Interface - `ISend.sc`

```
/*******************************************************/
/* High-level sender interface                         */
/*******************************************************/

// 3/16/98, A. Gerstlauer

interface ISend
{
  // Send with error correction
  void send(int data);

  // ... and without ...
  void sendRaw(int data);
};
```

D.2.2 Interface - `IRecv.sc`

```
/*******************************************************/
/* High-level receiver interface                       */
/*******************************************************/

// 3/16/98, A. Gerstlauer

interface IRecv
{
  // Receive with error correction/decoding
  int recv();

  // ... and without ...
  int recvRaw();
};
```

D.2.3 Error Correcting Channel - `CFec.sc`

```
/*****************************************************/
/* Error correction, encoding & decoding channels   */
/*****************************************************/

// 3/16/98, A. Gerstlauer


import "ISend";
import "IRecv";
import "IBit";



// Encoding channel
channel CFecEncode (ISendWord bitChannel)
  implements ISend
{
  word encoder(nibble data)
  {
    word result;

    result    = data;
    result[4] = data[2] ^ data[1] ^ data[0];
    result[5] = data[3] ^ data[2] ^ data[1];
    result[6] = data[3] ^ data[1] ^ data[0];

    return result;
  }

  void send(int data)
  {
    bitChannel.sendWord(encoder(data));
  }

  void sendRaw(int data)
  {
    bitChannel.sendNibble(data);
  }
};



// Decoding channel
channel CFecDecode (IRecvWord bitChannel)
  implements IRecv
{
  nibble decoder(word data)
  {
    unsigned bit[2:0] s;

    s[0] = data[6] ^ data[3] ^ data[1] ^ data[0];
    s[1] = data[5] ^ data[3] ^ data[2] ^ data[1];
    s[2] = data[4] ^ data[2] ^ data[1] ^ data[0];

    switch(s)
    {
      case 0: data = data ^ 0000000ub; break;
      case 4: data = data ^ 1000000ub; break;
      case 2: data = data ^ 0100000ub; break;
      case 1: data = data ^ 0010000ub; break;
      case 6: data = data ^ 0001000ub; break;
      case 3: data = data ^ 0000100ub; break;
      case 7: data = data ^ 0000010ub; break;
      case 5: data = data ^ 0000001ub; break;
```

```
    }

    return data[3:0];
  }

  int recv()
  {
    return decoder(bitChannel.recvWord());
  }

  int recvRaw()
  {
    return bitChannel.recvNibble();
  }
};

```

D.2.4 Sender - `Sender.sc`

```
/****************************************************/
/* Sender behaviors                                 */
/****************************************************/

// 3/16/98, A. Gerstlauer

import "ISend";
import "IBit";
import "CFec";

// Actual producer
behavior Producer (ISend oport)
{
  void main(void)
  {
    int i;

    i = 0;
    while (true) {
      waitfor(100);
      oport.send(i);
      oport.sendRaw(i);
      i = (i + 1) % 16;
    }
  }
};

// Encapsulating sender
behavior Sender (ISendWord ch)
{
  // Instantiate encoder channel
  CFecEncode encoder(ch);

  // Instantiate producer
  Producer producer(encoder);


  void main(void)
  {
    producer.main();
  }
};
```

D.2.5 Receiver - Receiver.sc

```
/*****************************************************/
/* Recevier behaviors                               */
/*****************************************************/

// 3/16/98, A. Gerstlauer

import "IRecv";
import "IBit";
import "CFec";

#include <stdio.h>


behavior Consumer (IRecv iport)
{
  void dump(int data, int dataRaw, int nr)
  {
    printf("%d: Received: %d, Raw: %d\n", nr, data, dataRaw);
  }

  void main(void)
  {
    int i;
    int data;
    int dataRaw;

    i = 0;
    while (true) {
      data = iport.recv();
      dataRaw = iport.recvRaw();
      dump(data, dataRaw, i);
      i = (i + 1) % 16;
    }
  }
};


behavior Receiver (IRecvWord ch)
{
  // Instantiate decoder channel
  CFecDecode decoder(ch);

  // Instantiate consumer
  Consumer consumer(decoder);


  void main(void)
  {
    consumer.main();
  }
};
```

D.3 Pipelined Sender & Receiver

This part contains the pipelined versions of the *Sender* and *Receiver* behaviors with overlapping execution of production/consumption and encoding/decoding sub-behaviors.

D.3.1 Sender - `Sender.sc`

```
/*****************************************************/
/* Sender behaviors                                  */
/*****************************************************/

// 3/16/98, A. Gerstlauer


import "IBit";


// Producer
behavior Producer (inout int data)
{
  void main(void)
  {
    waitfor(100);
    data = (data + 1) % 16;
  }
};


// Encoder
behavior Encoder (in int data, ISendWord bitChannel)
{
  word encoder(nibble data)
  {
    word result;

    result    = data;
    result[4] = data[2] ^ data[1] ^ data[0];
    result[5] = data[3] ^ data[2] ^ data[1];
    result[6] = data[3] ^ data[1] ^ data[0];

    return result;
  }

  void main(void)
  {
    bitChannel.sendWord(encoder(data));
    bitChannel.sendNibble(data);
  }
};


// Encapsulating sender
behavior Sender (ISendWord ch)
{
  piped int data;

  // Instantiate producer
  Producer producer(data);

  // Instantiate encoder
  Encoder  encoder(data);

  void main(void)
  {
    pipe {
```

```
      producer.main();
      encoder.main();
    }
  }
};
```

D.3.2 Receiver - `Receiver.sc`

```
/*****************************************************/
/* Recevier behaviors                                */
/*****************************************************/

// 3/16/98, A. Gerstlauer

import "IBit";

#include <stdio.h>


// Decoder
behavior Decoder (IRecvWord bitChannel, out int data, out int dataRaw)
{
  nibble decoder(word data)
  {
    unsigned bit[2:0] s;

    s[0] = data[6] ^ data[3] ^ data[1] ^ data[0];
    s[1] = data[5] ^ data[3] ^ data[2] ^ data[1];
    s[2] = data[4] ^ data[2] ^ data[1] ^ data[0];

    switch(s)
    {
      case 0: data = data ^ 0000000ub; break;
      case 4: data = data ^ 1000000ub; break;
      case 2: data = data ^ 0100000ub; break;
      case 1: data = data ^ 0010000ub; break;
      case 6: data = data ^ 0001000ub; break;
      case 3: data = data ^ 0000100ub; break;
      case 7: data = data ^ 0000010ub; break;
      case 5: data = data ^ 0000001ub; break;
    }

    return data[3:0];
  }

  void main(void)
  {
    data = decoder(bitChannel.recvWord());
    dataRaw = bitChannel.recvNibble();
  }
};



behavior Consumer (in int data, in int dataRaw)
{
  void dump(int data, int nr)
  {
    printf("Received: %d, Raw: %d\n", data, dataRaw);
  }

  void main(void)
  {
    dump(data);
    dump(dataRaw);
  }
```

```
};


behavior Receiver (IRecvWord ch)
{
  piped int data;
  piped int dataRaw;


  // Instantiate decoder
  Decoder   decoder(ch, data, dataRaw);

  // Instantiate consumer
  Consumer consumer(data, dataRaw);


  void main(void)
  {
    pipe {
      decoder.main();
      consumer.main();
    }
  }
};

```

D.4 Testbench - `fec.sc`

The testbench instantiates sender and receiver behaviors, connecting them via an instance of the word-level channel. The testbench allows to easily plug in different sender and receiver designs by importing different SpecC modules.

```
/*****************************************************/
/*                                                   */
/* Forward Error Correction (FEC) example            */
/*                                                   */
/*****************************************************/

/*****************************************************/
/* Top-level behavior, testbench                     */
/*****************************************************/

// 3/16/98, A. Gerstlauer


// Plug & Play: exchangable senders and receivers
import "v1/Sender";
//import "v2/Sender";
import "v1/Receiver";
//import "v2/Receiver";

import "bit/CBit";



// Test bench: connects sender and receiver via channel
behavior Main ()
{
  CWord   ch;

  Sender    sender(ch);
  Receiver  receiver(ch);
```

```
    void main(void)
    {
        par {
            sender.main();
            receiver.main();
        }
    }
};
```