CS-Note* 87

Bill J. Ellis and Andrew Ireland
December 14, 2005

Environment Simulator for Teaching SPARK

Contents

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

SPARK is often used to build a core high integrity component of a larger system. The Informed Approach underlines this observation, strongly encouraging designers to focus on the SPARK boundary - where SPARK ends and the other system begins.

In the real world, there are space ships, trains, nuclear reactors, aeroplanes ; all ideal systems with which to incorporate a SPARK component. Unfortunately[1] these systems are not available to students learning SPARK. An alternative is required.

1.1 The SPARK course

The SPARK course at Praxis-HIS, includes a medium size central heating control example. Students are provided with a specification and a bunch of boundary packages that simulate the central heating system. The students are then asked to build the control system in SPARK, interfacing with the central heating system via the boundary packages.

This is an effective example, sufficiently small to grasp in a short session, while still containing some surprising complexity. Unfortunately, there is no means to compile and execute the SPARK code. The boundary packages are just stubs.

1.2 Andrew's High Integrity Computing course

Andrew's High Integrity Computing course is illustrated via the SPARK Approach. In particular, the practical exercise for the course involves writing SPARK. Andrew designed an Automated Train Protection (ATP) simulation system example for this purpose.

The ATP example mirrors the style of the SPARK central heating example. However, it supports the compilation and execution of the SPARK code. The student would write SPARK code, exploiting provided boundary packages. These packages would connect with a top level system (also provided) allowing the student to execute their code. The top level system took the form:

```
procedure Test_ATP is
begin
   Env.Open_File;
   Log.Open_File;
   loop
      exit when Env.At_End;
      Env.Update;
      Log.Update;
      ATP.Control;
      Log.Update;
   end loop;
   Env.Close_File;
   Log.Close_File;
end Test_ATP;
```

The students were responsible for writing the code in `ATP.Control`. This system assists student understanding by letting them execute and visualise their code. Coincidently, it also eases the marking process, as the system log generated during program execution can be quickly checked against the same log generated by the correct system.

However, this system does not faithfully reflect a true SPARK system. The students code, and the environment is executed in an unrealistic turn by turn basis. Worse, the student's code must exit to allow the environment to continue. Thus the student's code must set global variables to capture any system state before their code exits.

1.3 Andrew's Idea

Andrew recognised the inherent weaknesses in the ATP system, and conjectured a student dissertation project to overcome them. The idea is to have a clear visually simulated environment[2], that provides an interface to key features of the environment. The student would then write a control program in SPARK that interfaces with this environment. The environment and the control program would be separate processes. Thus this would faithfully reflect a true embedded SPARK application.

[1] Or, perhaps, fortunately. . .

[2] With proper graphics. For example, via OpenGL or Java Drawing

2 Code Monkey

Andrew's idea is essentially a client/server model. The server is the environment, the client the control. These could communicate using sockets. And some tasking (separate process threads) might be required to hold the communication together. A little demonstration code was requested to see if this is possible, and how well it might work.

2.1 Architecture

The architecture adopted is below. Note that the client does not have any separate tasks, as it does not need to respond, and hence wait, for unknown communication.

```
-------------------------------------------------------
|                        SERVER                       |
|                                                     |
| ---------------  query/modify     ----------------- |
| |     TASK    |  the Environment  |     TASK      | |
| | Environment |<----------------->| Communication | |
| ---------------                   ----------*------ |
|                                             |       |
----------------------------------------------|--------
                                              |
                                              | Socket
                                              |
  |-------------------------------------------|--------
  |                      CLIENT               |       |
  |                                           |       |
  | ---------------                 ----------*------ |
  | | Students    |  call boundary  | Boundary      | |
  | | Control     |  packages       | Packages With | |
  | | Code        |<--------------->| Communication | |
  | ---------------                 | Backend       | |
  |                                 ----------------- |
  |                                                   |
  -----------------------------------------------------
```

2.2 Concrete Architecture

```
SERVER

server.adb-|(theworld: ads, adb)
           |(serversocket: ads, adb)

CLIENT

client.adb-|(interface: ads, adb)-|(clientsocket: ads, adb)
```

2.3 Weaknesses of Implementation

- **Exiting -** The server halts after so many loop iterations. When the server closes, the socket closes. Thus the client crashes on the next communication request! The client should probably have a separate task listening for the server closing down, and tidy up accordingly.
- **Separation -** The client communication interface is explicitly initialised in the SPARK code. This is a bit untidy.
- **Sockets -** The sockets operate over a local machine and on a hard coded channel number. Really, the client should broadcast to the server, and negotiate a channel number. There is no reason to suspect this can not be done. Helpful code exists inside the include file `g-socket.ads`.
- **Communication -** The communication is very unsophisticated, being restricted to numbers only. The code includes a very messy translation of `Boolean` to `Integer` and back again.

3 Example

Our example involves a reactor and its temperature. The control system should try to keep the temperature between 20 and 25 units by turning the reactor on and off accordingly. A server was created that presents a primitive display of the temperature (T) and the reactor (R) status. Two different clients were created. The first behaves correctly:

```
*                                 (    T: 0 R:FALSE)
0^1       ^10       ^20  ^25  ^!
*                                 (    T: 0 R:TRUE)
0^1       ^10       ^20  ^25  ^!
**                                (    T: 1 R:TRUE)
0^1       ^10       ^20  ^25  ^!
...
*********************             (    T: 20 R:TRUE)
0^1       ^10       ^20  ^25  ^!
**********************            (    T: 21 R:TRUE)
0^1       ^10       ^20  ^25  ^!
...
**************************        (    T: 25 R:TRUE)
0^1       ^10       ^20  ^25  ^!
***************************       (    T: 26 R:FALSE)
0^1       ^10       ^20  ^25  ^!
**************************        (    T: 25 R:FALSE)
0^1       ^10       ^20  ^25  ^!
...
```

The second client has a small bug, switching the reactor on (rather than off) at the peak temperature:

```
*                                 (    T: 0 R:FALSE)
0^1       ^10       ^20  ^25  ^!
*                                 (    T: 0 R:TRUE)
0^1       ^10       ^20  ^25  ^!
**                                (    T: 1 R:TRUE)
0^1       ^10       ^20  ^25  ^!
...
******************************    (    T: 29 R:TRUE)
0^1       ^10       ^20  ^25  ^!
*******************************   (    T: 30 R:TRUE)
0^1       ^10       ^20  ^25  ^!
*********
Meltdown!
*********
============
Server ended
============
```

3.1 Sparking the Example

The client is a complete SPARK application, which makes calls to an interface that is hidden from SPARK. By not giving the Examiner the details of the interface, the Examiner will function as desired without any 'hidden package' warnings[3]. For the example, the Examiner is called with `spark interface.ads client.adb`, producing the clean and correct:

```
          *******************************************************
     SPARK95 Examiner with VC and RTC Generator Release 7.1d01 / 01.04
                           Demonstration Version
          *******************************************************


                       DATE : 20-OCT-2005 15:17:30.29


           Examining the specification of package Interface ...

           Generating listing file interface.lst ...

           Examining main program Client ...

+++        Flow analysis of subprogram Client performed: no
           errors found.

           Generating listing file client.lst ...

           Generating report file ...


-----------End of SPARK Examination--------------------------------
```

4 Conclusions

The code presented here is a poor implementation of a good idea. However, it does serve to explain the nature of the task in more detail. Further, it provides some evidence that this task is possible.

[3] Hidden package warnings are quite acceptable. However it is felt that the more pure the SPARK client the better.

A Server

A.1 server.adb

```
with GNAT.Sockets;
with Ada.Text_IO;
with Ada.Exceptions;
use GNAT.Sockets;
use Ada.Exceptions;

with TheWorld;
with ServerSocket;

procedure Server is

    TimeToExit: Boolean := false;

    --Server has two threads (tasks):
    --
    --ONE: Simulate the world.
    --TWO: Provide read and write access to the world.
    --
    --These must be separate tasks. Otherwise, the world would
    --have to stop, every so often, to allow a burst of
    --communication. Not only would this be inconvenient (mixing
    --communication and simulation code) but it would interfere
    --with the simulation model, producing a more turn based style
    --(like the turn-based ATP model).

    --Ada tasks are quite confusing. This seems to work, but
    --i can't convincingly explain why.

   task Simulation is
      entry Start;
      entry Stop;
   end Simulation;

   task body Simulation is
   begin

      --The code for Start.
      accept Start;

      --Ada.Text_IO.Put_Line ("Simulation task started");

      TheWorld.Simulate;

      --The code for Stop.
      accept Stop;
      --Ada.Text_IO.Put_Line ("Simulation task ended");
      TimeToExit:= true;
   end Simulation;

   task Communication is
      entry Start;
      entry Stop;
   end Communication;

   task body Communication is
       Value: Integer;
   begin

      --The code for Start.
      accept Start;

      --Ada.Text_IO.Put_Line ("Communication task started");

      while not(TimeToExit) loop
          ServerSocket.ServerGetNumber(Value);

          --1 = tell me the Temperature.
          if (Value=1) then
              ServerSocket.ServerPutNumber(TheWorld.Temperature);
              --Ada.Text_IO.Put_Line ("Reporting the temperature to client.");
```

```
         --2 = tell me the Reactor status. (0=false ; 1=true)
         elsif (Value=2) then
             if TheWorld.Reactor then
                 ServerSocket.ServerPutNumber(1);
             else
                 ServerSocket.ServerPutNumber(0);
             end if;
             --Ada.Text_IO.Put_Line ("Reporting the reactor to client.");

         --3 = Turn off the Reactor.
         elsif  (Value=3) then
             TheWorld.Reactor:=false;
             --Ada.Text_IO.Put_Line ("At request of client, turning the reactor off.");

         --4 = Turn on the Reactor.
         elsif (Value=4) then
             TheWorld.Reactor:=true;
             --Ada.Text_IO.Put_Line ("At request of client, turning the reactor on.");

         else
             Ada.Text_IO.Put("Server received unknown command number:");
             Ada.Text_IO.Put(Integer'Image (Value));
             Ada.Text_IO.Put_Line("");
         end if;
      end loop;

      --The code for Stop.
      accept Stop;

      --Ada.Text_IO.Put_Line ("Communication task ended");
   end Communication;

begin
   Ada.Text_IO.Put_Line ("==============");
   Ada.Text_IO.Put_Line ("Server started");
   Ada.Text_IO.Put_Line ("==============");

   --Initialise.
   ServerSocket.ServerInitialise;
   TheWorld.Initialise;

   --Execute tasks.

   --  "Indicate whether the thread library provides process
   --  blocking IO. Basically, if you are not using FSU threads
   --  the default is ok."
   -- Hu?
   Initialize (Process_Blocking_IO => False);

   Simulation.Start;
   Communication.Start;
   Simulation.Stop;
   Communication.Stop;
   Finalize;

   --Close.
   ServerSocket.ServerClose;

   Ada.Text_IO.Put_Line ("============");
   Ada.Text_IO.Put_Line ("Server ended");
   Ada.Text_IO.Put_Line ("============");
end Server;
```

A.2 theworld.ads

```
with GNAT.Sockets;
with Ada.Text_IO;
with Ada.Exceptions;

use Ada.Exceptions;
use GNAT.Sockets;

package TheWorld is
```

```
    Temperature: Integer;
    Reactor: Boolean;

    procedure Initialise;
    procedure Simulate;

end TheWorld;
```

A.3 theworld.adb

```
package body TheWorld is

    procedure Initialise
    is
    begin
        Temperature:=0;
        Reactor:=false;
    end Initialise;

    procedure Simulate
    is
    begin

    --Run the world for 1000 steps.
    for I in 1 .. 1000 loop

        --Display the world.
        for I in 0 .. 30 loop
            if I <= Temperature then
                Ada.Text_IO.Put("*");
            else
                Ada.Text_IO.Put(" ");

            end if;
        end loop;
        Ada.Text_IO.Put("   (   ");
        Ada.Text_IO.Put("T:");
        Ada.Text_IO.Put (Integer'Image (Temperature));
        Ada.Text_IO.Put(" R:");
        Ada.Text_IO.Put (Boolean'Image (Reactor));
        Ada.Text_IO.Put_Line(")");
        Ada.Text_IO.Put_Line("0^1       ^10       ^20  ^25  ^!");

        --Test for meltdown!
        if Temperature >= 30 then
            Ada.Text_IO.Put_Line("*********");
            Ada.Text_IO.Put_Line("Meltdown!");
            Ada.Text_IO.Put_Line("*********");
            exit;
        end if;

        if Reactor = true then
            --Heating up.
            Temperature:=Temperature+1;
        else
            if Temperature>0 then
                --Cooling down.
                Temperature:=Temperature-1;
            end if;
        end if;

        delay 0.2;
    end loop;
    end Simulate;

end TheWorld;
```

A.4 serversocket.ads

```
with GNAT.Sockets;
with Ada.Text_IO;
with Ada.Exceptions;
```

```
use GNAT.Sockets;
use Ada.Exceptions;

package ServerSocket is

    FixedPortNumber: constant Port_Type := 5432;
    Address: Sock_Addr_Type;
    Channel: Stream_Access;

    ListenSocket: Socket_Type;
    TalkSocket: Socket_Type;

    procedure ServerInitialise;
    procedure ServerClose;
    procedure ServerGetNumber(Value: out Integer);
    procedure ServerPutNumber(Value: in Integer);

end ServerSocket;
```

A.5 serversocket.adb

```
package body ServerSocket is

    procedure ServerInitialise
    is
    begin
        -- Retrieve the (first reported) address of this machine.
        Address.Addr := Addresses (Get_Host_By_Name (Host_Name), 1);

        -- Get a socket address at a hard coded value.
        -- (Ideally, this value should be dynamically allocated. In this
        --  case the client would need to negotiate with the server to
        --  determine what the value is)
        Address.Port := FixedPortNumber;

        -- Create a socket.
        Create_Socket (ListenSocket);

        -- Allow reuse of local addresses.
        -- (This (probably) allows the server to re-allocate the hard
        --  coded port on any future invocation.
        --  This would allow this code to be run multiple times without
        --  restarting the computer or chancing the hard coded value)
        Set_Socket_Option
          (ListenSocket,
           Socket_Level,
           (Reuse_Address, True));

        -- Bind the socket to the address.
        Bind_Socket (ListenSocket, Address);

        -- Mark the socket as willing to receive connections.
        Listen_Socket (ListenSocket);

        -- Say something.
        Ada.Text_IO.Put_Line ("----------------------------------------");
        Ada.Text_IO.Put_Line ("Server:");
        Ada.Text_IO.Put_Line ("I am waiting for a client to say hello.");
        Ada.Text_IO.Put_Line ("----------------------------------------");

        -- Listen for a client.
        -- When a client connects, Socket reflects the client socket.
        -- The server socket is still available for other clients.
        -- (However, here we assume one client)
        Accept_Socket (ListenSocket, TalkSocket, Address);

        -- Say something.
        Ada.Text_IO.Put_Line ("----------------------------------------");
        Ada.Text_IO.Put_Line ("Server:");
        Ada.Text_IO.Put_Line ("Ah, hello client!");
        Ada.Text_IO.Put_Line ("----------------------------------------");

        -- Return a stream associated to the connected socket.
        -- These streams appear to be bi-directional. Cool!
```

```
        Channel := Stream (TalkSocket);
    end ServerInitialise;

    procedure ServerClose
    is
    begin
        Close_Socket (ListenSocket);
        Close_Socket (TalkSocket);
    end ServerClose;

    procedure ServerGetNumber(Value: out Integer)
    is
    begin
         --Wait for and return a number from the client.
         Value := Integer'Input (Channel);
    end ServerGetNumber;

    procedure ServerPutNumber(Value: in Integer)
    is
    begin
         Integer'Output (Channel, Value);
    end ServerPutNumber;

end ServerSocket;
```

B Good Client

B.1 client.adb

```
with Interface;
--# inherit Interface;

--# main_program;
procedure Client
--# global in Interface.Read_Reactor;
--#        in Interface.Read_Temperature;
--#        out Interface.Write_Reactor;
--# derives Interface.Write_Reactor from
--#         Interface.Read_Reactor, Interface.Read_Temperature;
is
        Temperature: Integer;
        Reactor: Boolean;
begin
    -- This initialise is bit unfortunate.
    -- A better design might hide this outside the core spark..
    Interface.Initialise;

    loop
        Interface.GetReactor(Reactor);
        Interface.GetTemperature(Temperature);

        if (Temperature <= 20) then
            --0..20
            if not(Reactor) then
            Interface.TurnReactorOn;
            end if;
        else
            if (Temperature <= 25) then
            --21..25
            --The ideal. Do nothing.
                null;
            else
                --26..??
                if Reactor then
                Interface.TurnReactorOff;
                end if;
            end if;
        end if;
    end loop;
end Client;
```

C Bad Client

C.1 badclient.adb

```
with Interface;
--# inherit Interface;

--# main_program;
procedure BadClient
--# global in Interface.Read_Reactor;
--#        in Interface.Read_Temperature;
--#        out Interface.Write_Reactor;
--# derives Interface.Write_Reactor from
--#         Interface.Read_Reactor, Interface.Read_Temperature;
is
        Temperature: Integer;
        Reactor: Boolean;
begin
    -- This initialise is bit unfortunate.
    -- A better design might hide this outside the core spark..
    Interface.Initialise;

    loop
        Interface.GetReactor(Reactor);
        Interface.GetTemperature(Temperature);

        if (Temperature <= 20) then
            --0..20
            if not(Reactor) then
            Interface.TurnReactorOn;
            end if;
        else
            if (Temperature <= 25) then
            --21..25
            --The ideal. Do nothing.
                null;
            else
                --26..??
                if Reactor then
                --*ERROR: should be: Interface.TurnReactorOff;
                Interface.TurnReactorOn;
                end if;
            end if;
        end if;
    end loop;
end BadClient;
```

D General client support

D.1 interface.ads

```
package Interface
    --#own in Read_Reactor;
    --#    in Read_Temperature;
    --#    out Write_Reactor;
is
    procedure Initialise;
    --# derives ;

    procedure GetTemperature(Temperature: out Integer);
    --# global in Read_Temperature;
    --# derives Temperature from Read_Temperature;

    procedure GetReactor(Reactor: out Boolean);
    --# global in Read_Reactor;
    --# derives Reactor from Read_Reactor;

    procedure TurnReactorOff;
    --# global out Write_Reactor;
    --# derives Write_Reactor from ;

    procedure TurnReactorOn;
    --# global out Write_Reactor;
```

```
    --# derives Write_Reactor from ;

end Interface;
```

D.2 interface.adb

```
with ClientSocket;

package body Interface is
    --#hide Interface;

    procedure Initialise
    is
    begin
        ClientSocket.ClientInitialise;
        --Give the server a bit of time to get itself sorted.
        delay 0.1;
    end Initialise;

    procedure GetTemperature(Temperature: out Integer)
    is
    begin
        ClientSocket.ClientPutNumber(1);
        ClientSocket.ClientGetNumber(Temperature);
    end GetTemperature;

    procedure GetReactor(Reactor: out Boolean)
    is
        Value: Integer;
    begin
        ClientSocket.ClientPutNumber(2);
        ClientSocket.ClientGetNumber(Value);
        if Value=0 then
            Reactor:=false;
        else
            Reactor:=true;
        end if;
    end GetReactor;

    procedure TurnReactorOff
    is
    begin
        ClientSocket.ClientPutNumber(3);
    end TurnReactorOff;

    procedure TurnReactorOn
    is
    begin
        ClientSocket.ClientPutNumber(4);
    end TurnReactorOn;

end Interface;
```

D.3 clientsocket.ads

```
with GNAT.Sockets;
with Ada.Text_IO;
with Ada.Exceptions;
use GNAT.Sockets;
use Ada.Exceptions;

package ClientSocket is

    FixedPortNumber: constant Port_Type := 5432;
    Address: Sock_Addr_Type;
    Channel: Stream_Access;
    TalkSocket: Socket_Type;

    procedure ClientInitialise;
    procedure ClientClose;
    procedure ClientGetNumber(Value: out Integer);
    procedure ClientPutNumber(Value: in Integer);
end ClientSocket;
```

D.4 clientsocket.adb

```
package body ClientSocket is

    procedure ClientInitialise
    is
    begin
        -- Retrieve the (first reported) address of this machine.
        Address.Addr := Addresses (Get_Host_By_Name (Host_Name), 1);

        -- Get a socket address at a hard coded value.
        -- (Ideally, this value should be dynamically allocated. In this
        --  case the client would need to negotiate with the server to
        --  determine what the value is)
        Address.Port := FixedPortNumber;

        -- Create a socket.
        Create_Socket (TalkSocket);

        -- Allow reuse of local addresses.
        -- (This (probably) allows the server to re-allocate the hard
        --  coded port on any future invocation.
        --  This would allow this code to be run multiple times without
        --  restarting the computer or chancing the hard coded value)
        Set_Socket_Option
          (TalkSocket,
           Socket_Level,
           (Reuse_Address, True));

        -- Connect to an existing socket.
        Connect_Socket (TalkSocket, Address);

        -- Return a stream associated to the connected socket.
        -- These streams appear to be bi-directional. Cool!
        Channel := Stream (TalkSocket);
    end ClientInitialise;

    procedure ClientClose
    is
    begin
        Close_Socket (TalkSocket);
    end ClientClose;

    procedure ClientGetNumber(Value: out Integer)
    is
    begin
         Value := Integer'Input (Channel);
    end ClientGetNumber;

    procedure ClientPutNumber(Value: in Integer)
    is
    begin
         Integer'Output (Channel, Value);
    end ClientPutNumber;
end ClientSocket;
```