Problem 1 [15 points]:
Compare the exception domains of the following apparently identical pieces of code (the variable Initial is of type integer):

```vhdl
procedure Do_Something is
    subtype Small_Int is Integer range -16..15;
    A; Small_Int := Initial;
begin
    ...
end Do_Something;
```

```vhdl
procedure Do_Something2 is
    subtype Small_Int is Integer range -16..15;
    A; Small_Int;
begin
    A := Initial;
    ...
end Do_Something2;
```

```vhdl
procedure Do_Something3 is
    subtype Small_Int is Integer range -16..15;
    A; Small_Int := Initial;
begin
    begin
        A := Initial;
        ...
    end;
end Do_Something3;
```

Solution:

In the first code fragment, the exception can not be handled by the Do_Something procedure since the domain is the calling code.

In the second code fragment, the exception can be handled by the Do_Something procedure since its domain is the procedure itself.

In the final case, the exception can be handled within the inner block declared within the procedure since the domain is this block.
Problem 2 [15 points]:
In a process control application, gas is heated in an enclosed chamber. The chamber is surrounded by a coolant which reduces the temperature of the gas by conduction. There is also a valve which when open releases the gas into the atmosphere. The operation of the process is controlled by an Ada package whose specification is given below. For safety reasons, the package recognizes several error conditions; these are brought to the notice of the user of the package by the raising of exceptions. The exception Heather_Stuck_On is raised by the procedure Heater_Off when it is unable to turn the heater off. Similarly other two exceptions are listed below.

package Temperature_Control is
    Heater_Stuck_On, Temperature_Still_Rising, Valve_Stuck: exception;

procedure Heater_On;
    -- turn on heater
procedure Heater_Off;
    -- turn off heater
    -- raises Heater_Stuck_On
procedure Increase_Coolant;
    -- causes the flow of coolant which surrounds the chamber to increase until temperature reaches a safe level
    -- raises Temperature_Still_Rising alarm
procedure Open_Valve;
    -- opens a valve to release some of the gas there by avoiding an explosion
    -- raises Valve_Stuck

procedure Panic;
    -- sounds an alarm and calls the fire services

end Temperature_Control;

Write an Ada (or C, C++ if you like) procedure which when called will attempt to turn off the heater in the gas chamber. If the heater is stuck on then the flow of coolant surrounding the chamber should be increased. If the temperature is still rising then the escape valve should be opened. If all fails, then the alarm must be sounded.
Solution:

```plaintext
procedure Reliable_Heater_Off is
type Stage is (First, Second, Third, Fourth);
begin

  for I in Stage loop
  begin
    case I is
    when First =>
      Heater_Off;
      exit;
    when Second =>
      Increase_Coolant;
      exit;
    when Third =>
      Open_Valve;
      exit;
    when Fourth =>
      Panic;
      exit;
    end case;
    exception
    when Heater_Stuck_On |
      Temperature_Still_Rising |
      Valve_Stuck =>
      null;
    end;
  end loop;
end Reliable_Heater_Off;
```

Problem 3 [10, 10 points]: Time and Value Domain Errors
Are timing and value failures orthogonal? You can answer this question by explaining the following two questions: (a) explain how a system can be transformed so that all timing failures manifest themselves as value failures; (b) explain if the converse of (a) can be achieved? Is this universally true, or explain the circumstances under which the converse is true.

Solution:

*A timing failure is defined to be the delivery of a service outside its defined delivery interval—typically beyond some defined deadline. Often the service is delivered late because of the time needed to construct the correct value for the service. If the system was designed to always deliver a value in the correct interval then the “lack of time” would be manifest as an incorrect value (delivered on time). Hence timing and value failures can not be considered orthogonal.*
The converse to the above can also be true. A service that has failed because the value it delivers is incorrect may be able to deliver a correct value if it is given more (CPU) time; hence a correct value may be delivered but too late. However, this is not universally true; a value failure (or error) can be due to many reasons (e.g., software error, hardware error) other than insufficient allocation of processor cycles.

Problem 4 [10 points]: Providing delay through blocking

Using your favorite language, implement a monitor interface to a simple (coarse) delay mechanism consisting to two procedures tick() and delay(). A caller to delay(int d) wishes to be suspended for d ticks. The procedure tick() is called by some clock routine. Whenever it is called each process blocked on delay() wakes up, decrements some counter (which is initialized to d) and exits the monitor if the counter has reached zero – otherwise it reblocks.

Show how the body of the monitor can be implemented.

Hint: You can use condition variables that have wait and signal operations defined on them.

Solution:

Use a blocking semantics for signal, i.e., a signal that wakes a process is blocking. So the process that wakes another up waits until that process has exited (or become suspended) before continuing. A cascade wake up can thus be used for delay.

For instance:

```
monitor body time is
  CV : condition;

  procedure tick is
    begin
      signal (CV);
    end;

  procedure delay (D: natural) is
    count_down: natural:=D;
    begin
      while count_down > 0 loop
        wait(CV);
        signal(CV); -- will block if this wakes another process
        count_down := count_down – 1;
      end loop;
    end delay;
end time;
```