Tommy Ingulfsen
July 30, 2002

Pushing Tin Project - Fourth Design Draft

1 Introduction

This design document is a refinement of Design Draft 3. This paper contains a more detailed design for the SPARK ACDS subsystem (see section 3.6), the Generator and a little more on the Scheduler, as well as refined figures. Apart from these changes the contents are the same as for Design Draft 3.

The INFORMED design method has been used for the ACDS, as explained in the paper *INFORMED Design Method for SPARK* by Praxis Critical Systems. The rest of the program has not been subjected to any particular methodology, but is the result of careful thought and code prototyping.

While this paper lays out a design that should be ready to implement, it may well be that changes will be made during coding and testing as needed. When the coding is finished, an updated design document will be written. That report will describe the actual design.

2 System Architecture

Figure 1 gives an overview of the system. Figure 2 gives a more detailed picture of the ACDS part.

When complete, the application will consist of four different main parts. The Generator that updates the radar situation, the Scheduler that runs everything and keeps track of time, the ACDS that finds out if any aircraft are in danger of colliding and the Display that provides output to the user.

The World is a collection of global data structures that holds the radar situation and the results from the ACDS.

It is intended to keep these four subsystems as separate as possible. This will ensure that any changes to the program are as localised as possible. Ideally, the Scheduler will be the only part that knows of the other components. The intention is that the Scheduler will only call each of the Generator, the ACDS and the Display once during any program iteration.

The design of the Display is incomplete. To begin with any output to the user will be simple text displayed on the command line. Later it will hopefully be in a GUI. Since the Display has great opportunities for enhancement, it will be designed separately as coding progresses.

3 ACDS Design

As mentioned, the INFORMED method of Praxis has been used for this part. The steps denoted below correspond to the 6 steps of this design methodology.

3.1 Identification of the system boundary, inputs and outputs

This step is concerned with the entire system, not just the safety-critical core that will be written in SPARK.

The boundary of the ENTIRE system is the display (outputs). There is no specific boundary for inputs, since these are embedded within the Generator part. The outputs of the system and the random generation are not considered to be an important part of this design, since we are only concerned with the SPARK core.

*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).

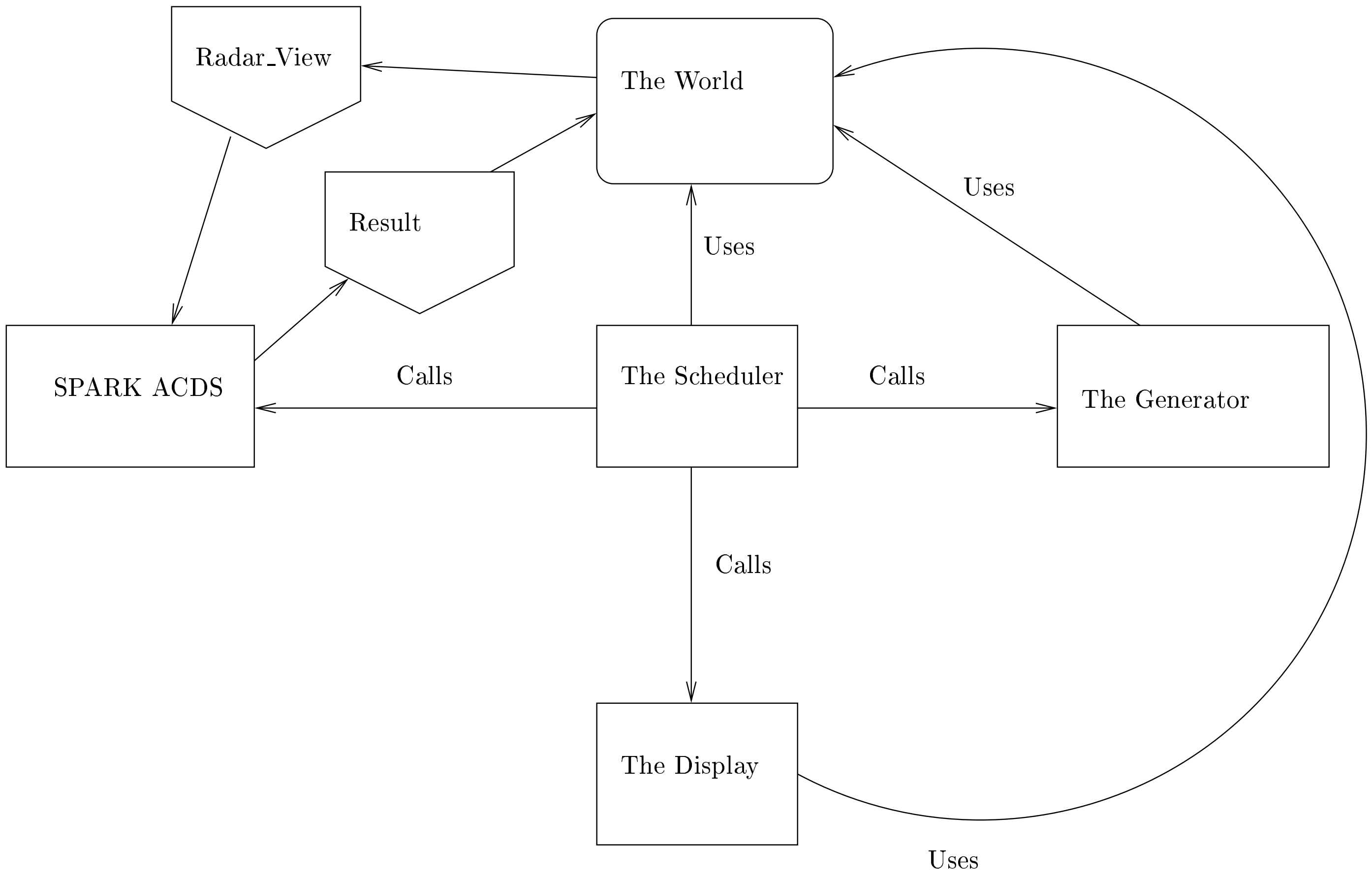

Figure 1: System Architecture

3.2 Identification of the SPARK boundary

The boundary consists of the data input to the SPARK subsystem and the results (outputs) that are obtained.

The input makes up the "radar view" of the current situation. The radar view does not include additional information on aircraft such as their callsign, but just contains the X, Y, Z co-ordinates of each aircraft. So to the safety-critical SPARK subprograms, the current state of the system is just a set of points in the 3-D space.

The boundary could be implemented as an array of the current flight positions. A straightforward way to deal with this is to make a global package Radar_View that is updated by the Scheduler and read by the ACDS:

```
package Radar_View
is
– in order to read the updated array
– for use by SPARK sys.
procedure Poll;

– in order to update the array and the size
– for use by "Scheduler"
procedure Update;

– SPARK sys. check this to get size of array
function Get_Size return Integer;
end Radar_View;
```

Clearly, some mechanism for controlling the size of the array is needed, this is provided in the Get_Size function. Get_Size can be called by the SPARK module to find out how many elements are present in the array.

As for the outputs from the safety-critical core, this could be implemented in a similar way. We might use a globally available package Results like this:

```
package Results
is
– SPARK updates array
procedure Update;

– Scheduler reads results
procedure Read;
end Results
```

There should not be any need to add boundary abstraction layers for either of the boundaries.

3.3 Identification and localisation of system state

The essential state of the system is made up of

- The current positions of all aircraft within radar range
- Their future positions
- A combination of the above 2 for use in the computations
- The result of the Short Term Collision Avoidance (STCA) calculations

The result and the current positions are already described as boundary variables.

In order to predict the future positions of aircraft, the previous positions must be stored. Given the state at the previous snapshot, linear interpolation can be used to make a well-qualified guess on the routes that aircraft will take within the next time instant.

The prior situation should be stored within the SPARK boundary. An alternative, slightly easier to implement, is to have it as part of the Radar_View package, although coupling/cohesion might suffer as a result.

Because of the algorithm being used to detect potentially hazardous situations, the simplest way to store information on aircraft at an intermediate stage is as a line. A line consists of the "present" (X, Y, Z) co-ordinates and a direction vector. See the file "Line_Utils.ads".

As a summary, the following data structures will exist within/be used by the SPARK STCA module:

- Radar_View (boundary variable package)
- Results (boundary variable package)
- Line_Utils (type package, utilities for working with vectors)
- Radar_State (a mechanism to extract the next position of an aircraft from the current/prior positions)

3.4 Handling initialisation of state

The inputs can be regarded as initialised by the Scheduler. The Results are initialised by the STCA program.

During the second iteration of program execution, the STCA will initialise the previous positions.

3.5 Handling secondary requirements

In this context secondary requirements denote additional needs that are not strictly necessary for how the system itself will work, but have arisen due to some other consideration - e.g testing.

In this project, no secondary requirements have been identified yet.

3.6 Implementing the internal behaviour of components

Some packages, like one to find the square root and Line_Utils, have been implemented. The others are lined out here.

The ACDS will be broken down into several packages. The top level package will be called ACDS. A private child package Radar will contain the main program of the STCA system and hold the data structure, Radar_State required to store the previous and current state of the radar image. Associated with Radar_State will be some procedures to initialise and update or read its contents.

A private child package ACDS.Lines will hold the procedures that derives aircraft paths from the Radar_View. Another private child package, ACDS.Proximity will hold the procedures that checks the distance between aircraft.

ACDS.Proximity has been partly implemented already (see "proximity.ads" and "proximity.adb"). Additions to this package includes a procedure that drives the minimum distance checking by iterating through an array of aircraft, checking every aircrafts path against every other's. ACDS.Lines has not yet been started.

The "main" collision detection subprogram will be rather short. It will consist of calls to update the Radar_State using the Radar_View boundary package, derive the plane paths and execute the proximity checks, hence utilising other procedures to carry out its tasks. After the proximity checks have been carried out, the ACDS.Proximity procedures will update the Result boundary package.

As mentioned, the packages that make up the SPARK ACDS will most likely be organised in a hierarchy with ACDS on top and the others as private child packages. The advantage of this is that it becomes possible to have the code in several smaller files rather than a single one. But it is not quite certain that this is the way the software will be arranged since I have no experience with package hierarchies. If using private child packages does not work, public packages or one big file will be used instead.

In summary, a typical execution of the SPARK Automatic Collision Detection System will take the following form:

1. The Scheduler updates The World to hold the new picture of the radar.
2. The Scheduler updates Radar_View with the new data.
3. The Scheduler calls The SPARK ACDS.
4. The SPARK ACDS reads Radar_View.
5. The SPARK ACDS uses data from the Radar_View to update its Radar_State.
6. The SPARK ACDS derives the aircraft paths using Radar_State.
7. The SPARK ACDS performs a proximity check to find out if any flight paths are hazardous.
8. The SPARK ACDS updates the Result.
9. The Scheduler reads Result and calls the Display to give feedback to the software user.

Figure 2 gives a picture of the SPARK subsystem, showing the inside of the SPARK box of figure 1.

4 Generator Design

The Generator produces and updates the current situation by using random number generation. There will be 2 air routes forming an X:

Using 2 routes provides simplicity, while the X-shape ensures a sufficient danger of collision in the region where they meet.

Aircraft will fly in along a linear route from one end of a route to the other. The ends are denoted "gates". The gates are given fixed co-ordinates according to the system given in figure 4 below.

4.1 Air routes specification

The X-direction gives the horizontal position, Y denotes altitude and Z denotes "depth". A move from X = 1 to X = 2 is a movement 1 meter to the right.

Z range:

- Gates 1 and 2: 0
- Gates 3 and 4: 66667

Y ranges from 500 to 5500 at all points along any route.

X range:

- Gate 1: 0 .. 4000
- Gate 2: 20000 .. 24000
- Gate 3: 20000 .. 24000
- Gate 4: 0 .. 4000

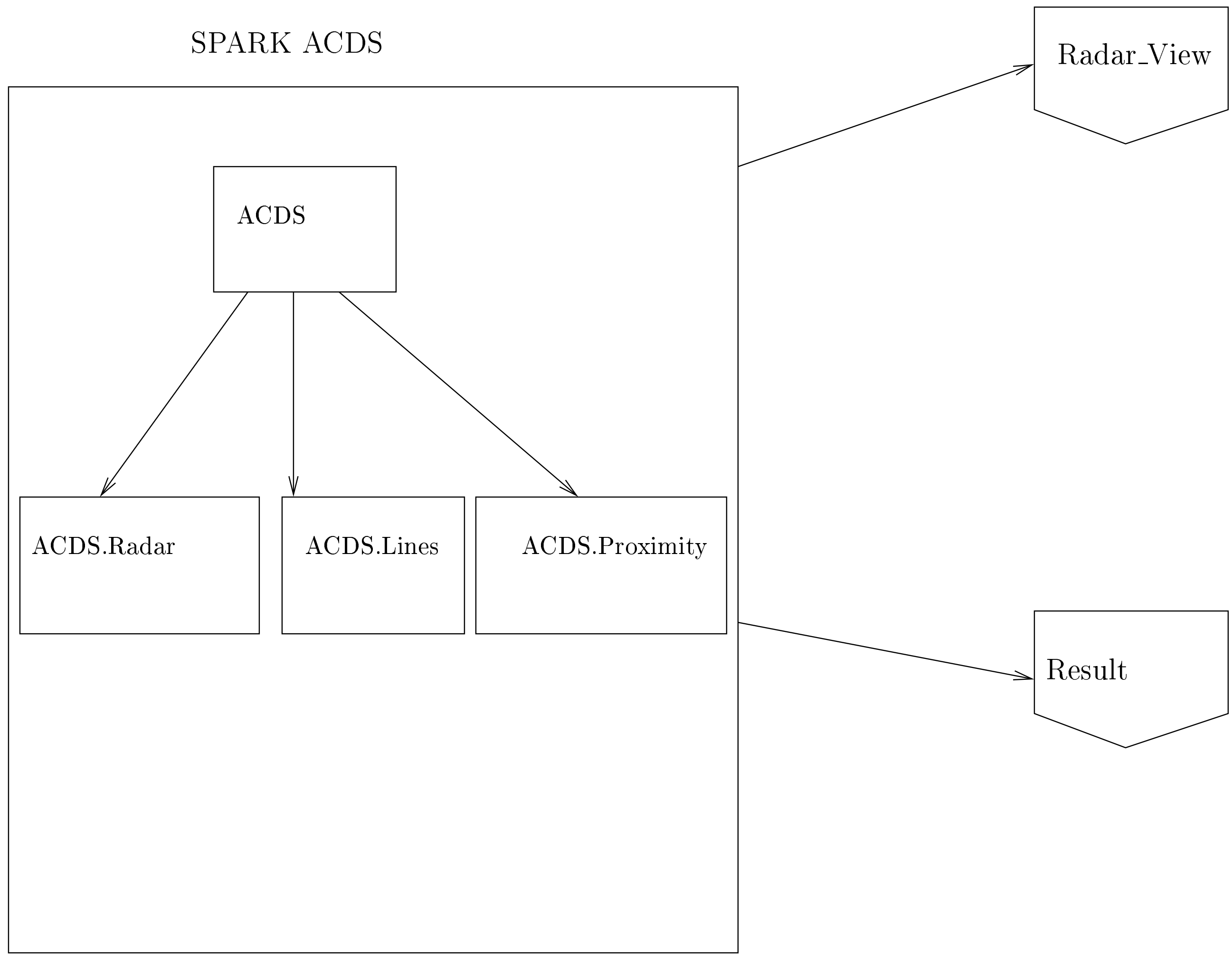

Figure 2: SPARK ACDS internally

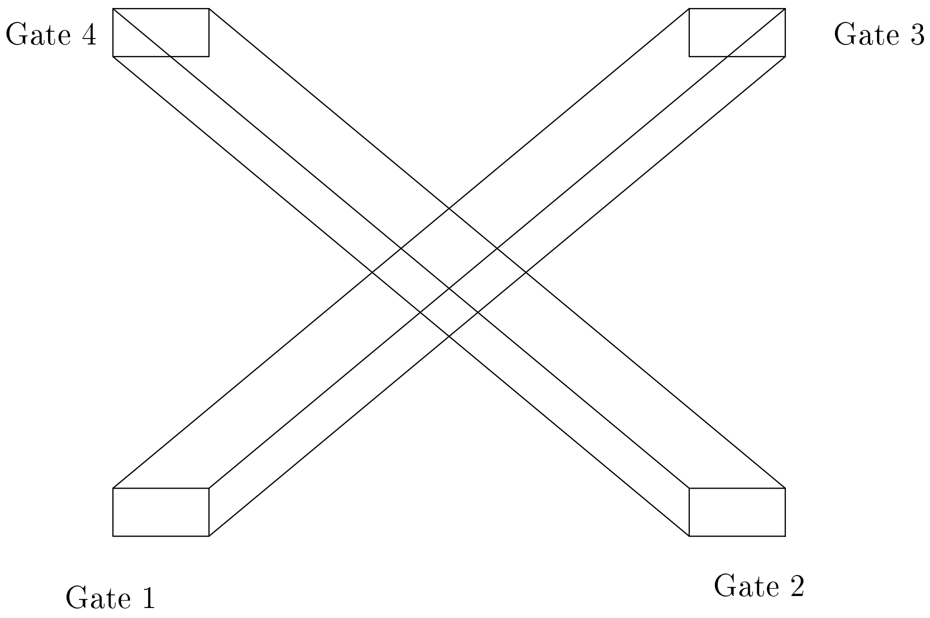

Figure 3: Airways

These numbers have been chosen to give as realistic a system as possible. They must be seen in conjunction with the speed of the aircraft as well as the total duration of a single simulation. The speed of an aircraft will be about 800 km/hour, circa 500 mph. If we let the program run for 5 minutes before it halts automatically, then an aircraft will have time to complete a single journey along its route.

Clearly, these constants will be represented within some package. They can (and probably will) be altered at any point in the implementation. The reason for this is that the application should not be too "boring", that is, it should not take too long before a hazardous situation arises. On the other hand, there should not be too many such cases either, and so the constants described above will be tampered with until the performance is deemed satisfactory.

In implementation terms, the ranges for the gates will be used to provide start and end points for aircraft, using random number generation.

As a comment to the previous design document, changing altitude "with probability P_a" means that the Z co-ordinate of the end point corresponding to the aircraft will change, causing a climb or descent. (This highlights a spelling mistake in the Second Design Draft - in section 3.1 it should say "With probability P_n introduce a new aircraft into the system.")

In addition to a package representing constants and ranges, there will be a package for holding data on aircraft in the system. That package is going to store the current co-ordinates of aircraft, an aircraft's direction vector, its callsign in an array of planes. This will be part of "The World", but is only updated by the Generator. See "aircrafts.ads". The Scheduler will read information from this before passing on the co-ordinates to the ACDS.

As for the actual processing in the Generator, there will be one package for updating the system. This package will have methods to

- Introduce new aircraft into the system
- Remove aircraft that have disappeared past the depth limits (Z $<$ 0 or Z $>$ 66667)
- Change the altitude of an aircraft
- Move all aircraft forward one step (updating upon a new iteration of the main program loop)

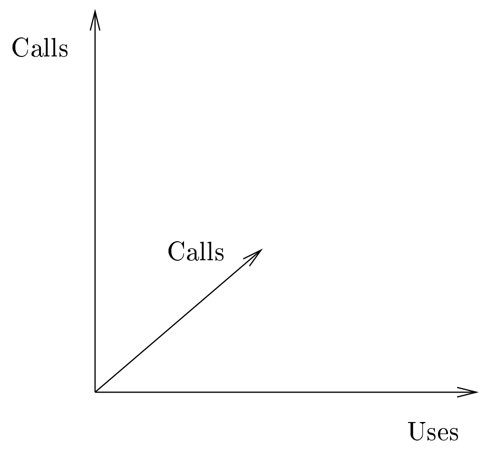

Figure 4: Co-ordinate System

- A "main" Generator procedure that is called by the Scheduler and distributes control as needed to execute radar system updates.

5 Scheduler Design

The Scheduler will be the "main" subprogram within the application, and is going to be kept as small as possible. The Scheduler is the "glue" that holds the other parts together.

As explained in Design Draft 2, the Scheduler will be structured as a loop. Before the loop, initialisation of "The World" and any other data structures (e.g. the "Result" boundary of the SPARK subsystem) will be conducted.

In the loop there will be calls to the Generator to update the situation, calls to the ACDS to evaluate risk and calls to the Display to give the user some feedback. To simulate the passage of time there will be a few seconds delay before the succeeding iteration of the loop.

The Scheduler is intended to be kept so small that it will fit into a single package.

6 The World

"The World" is a collection of globally accessible data structures. It encompasses the following:

- Result (Boundary variable package for ACDS results)
- Radar_View (Boundary variable package for ACDS inputs)
- Aircrafts (Type package for holding aircrafts with callsigns)
- New_Planes (Type package holding info on new aircraft entering the system - see below)

7 Display Design

To begin with, this component will be basic. It will negotiate "The World" to find out if any potentially dangerous situations have arisen, and probably notify the user of any aircraft leaving or entering the system.

Over time, however, this is the part that has the greatest possibilities for improvement. When the Generator, the Scheduler, the ACDS and the simple initial Display are coded and tested it is likely that a Graphical User Interface (GUI) will be written into the Display. Opportunities exist for letting the user interact using buttons and visualisation using graphics.

Because little is known at present about how to create GUIs using GtkAda (a GUI package for Ada), design of the Display is left until later.

8 Implementation

When the design is agreed upon, work will proceed in parallel on the SPARK and the ordinary Ada parts. Documentation will also be written while coding still goes on. Unless unforeseen difficulties arise, there should be plenty of time to improve the system. Some suggestions for improvement are:

- A GUI (likely to be implemented)
- Adding graphics to the GUI
- Implementing a second algorithm for the ACDS proximity calculations, for comparison
- Letting the user interact with the application by changing the flight paths of the aircraft that are in danger of collision

The choice of what to do here depends on how much time is left when the core system (encompassing fully working versions of the Generator, Scheduler and the ACDS) has been coded and at least some testing carried out.

A line_utils.ads - code

```
-- File "line_utils.ads" - holds various useful data structure specifications.
-- Especially the representation of a line.

-- Author: Tommy Ingulfsen, July 2002

-- A line segment X_1 is represented as X_1 = a + td where
-- a = the starting points of the line
-- d = the direction of the line
-- t = time (between 0 and 1)
-- In 3-D of course.

with Aircrafts;
--# inherit Aircrafts;

package Line_Utils
is
    -- Minimum separation between aircraft, in meters.
    Min_Separation: constant Integer := 1610;

    -- Data type for a vector representation of a line.
    type Line is
       record
           X_Start: Integer;
           Y_Start: Integer;
           Z_Start: Integer;
           X_Direction: Integer;   -- line direction in X
           Y_Direction: Integer;   -- line direction in Y
           Z_Direction: Integer;   -- line direction in Z
       end record;

    -- We also need a data type for an array of lines
    type Line_Array is array(Aircrafts.System_Size) of Line;

    -- We are only concerned with a single second.
    subtype Time_Interval is Float range 0.0 .. 1.0;
end Line_Utils;
```

B acds_aircrafts.ads - code

```
-- File "acds_aircrafts.ads" - data structure for holding information on
-- an aircraft. This is for use within the Automatic Collision Detection
-- System only. It therefore holds considerably less information than
-- the full Aircrafts package (See "aircrafts.ads").

-- Author: Tommy Ingulfsen, July 2002.

with Aircrafts;
with Line_Utils;
--# inherit Aircrafts, Line_Utils;

package ACDS_Aircrafts is

    -- An aircraft is simply represented as a line within the ACDS
    -- We will work with an array of aircraft repr. system
    type Aircraft is array(Aircrafts.System_Size) of Line_Utils.Line;

    -- Gather array of aircrafts and its ACTUAL size in one package
    type ACDS_Planes is
       record
           Planes: Aircraft;
           Planes_Size: Integer;
       end record;

    -- Holds the other aircrafts that may collide with "this"
    -- An array of an array: Dimension 1 = All planes
    --                       Dimension 2 = All planes that "this" plane can collide with
    -- INVALID = empty entry (see Aircrafts.ads)
    type Collision_Array is array(Aircrafts.System_Size) of Aircraft;

    -- Gather array of collisions and its ACTUAL size in one package
    type Collisions is
       record
           Hazards: Collision_Array;
           Hazards_Size: Integer;    -- size of first dimension
       end record;



end ACDS_Aircrafts;
```

C proximity.ads - code

```
-- File "proximity.ads" - checks the minimum distance between 2 lines
-- (flight paths) in the next time interval.

-- Author: Tommy Ingulfsen, July 2002



with Line_Utils;
with Square_Package;
with Spark_IO;
--# inherit Square_Package, Line_Utils, Spark_IO;
-- inherit Line_Utils;  CANNOT HAVE MORE THAN --# inherit!!
package Proximity
is
    -- Finds the minimum distance between 2 planes on a line within a time interval,
    -- when the planes are moving at the same, constant speed and they keep moving.
    procedure Min_Distance(Line_1: in Line_Utils.Line;
                           Line_2: in Line_Utils.Line;
   Result: out Boolean);
    --# global in out Spark_IO.Outputs;
    --# derives Result from Line_1, Line_2 &
    --#         Spark_IO.Outputs from Spark_IO.Outputs;



    --Converts integer co-ordinates to floating point for the float calculations.
    procedure Convert(I1x: in Integer; I1y: in Integer; I1z: in Integer;
                      I2x: in Integer; I2y: in Integer; I2z: in Integer;
                      I1dx: in Integer; I1dy: in Integer; I1dz: in Integer;
                      I2dx: in Integer; I2dy: in Integer; I2dz: in Integer;
                      I1xf: out Float; I1yf: out Float; I1zf: out Float;
                      I2xf: out Float; I2yf: out Float; I2zf: out Float;
                      I1dxf: out Float; I1dyf: out Float; I1dzf: out Float;
                      I2dxf: out Float; I2dyf: out Float; I2dzf: out Float);
    --# derives I1xf from I1x &
    --#         I1yf from I1y &
    --#         I1zf from I1z &
    --#         I2xf from I2x &
    --#         I2yf from I2y &
    --#         I2zf from I2z &
    --#         I1dxf from I1dx &
    --#         I1dyf from I1dy &
    --#         I1dzf from I1dz &
    --#         I2dxf from I2dx &
    --#         I2dyf from I2dy &
    --#         I2dzf from I2dz;

    -- Finds part of the square root when calculating the actual min distance,
    -- avoiding too much repetitive code in body of procedure Min_Distance.
    procedure Part_Calc(Part_Root: out Float; -- result of calculation
                        Line1ns: in Float;    -- start co-ord for dimension n, line 1
                        Line1nd: in Float;    -- end co-ord for dimension n, line 1
```

```
                        Line2ns: in Float;    -- start co-ord for dimension n, line 2
                        Line2nd: in Float;    -- end co-ord for dimension n, line 2
                        Time: in Float);      -- some time parameter
    --# global in out Spark_IO.Outputs;
    --# derives Spark_IO.Outputs from Spark_IO.Outputs &
    --#         Part_Root from Line1ns, Line1nd, Line2ns, Line2nd, Time;


    -- Finds the distance between two points on two lines, given the vector and the
    -- time parameters.
    procedure Actual_Distance(Line_1: in Line_Utils.Line;
                              Line_2: in Line_Utils.Line;
                              Time: in Float;
                              Smallest_Distance: out Float);
    --# derives Smallest_Distance from Line_1, Line_2, Time;


end Proximity;
```

D proximity.adb - code

```
-- File "proximity.adb" - package body for "proximity.ads". Contains the
-- implementation of the function Min_Distance, which calculates the
-- minimum distance between 2 lines.

-- ADD CHECK FOR IDENTICAL LINES, NEGATIVE/ZERO VALUES IN DENOMINATOR!
-- WHAT HAPPENS TO t WHEN THE LINES INTERSECT OR ARE PARALLELL? TEST!!!
-- ADD CHECK FOR 0/0!

-- WHAT ABOUT LIMITING THE SEPARATIONS BEHIND THE CURRENT POSITION AND IN FRONT OF IT?
-- ADD TEST TO SEE IF AIRCRAFT ARE TO FAR AWAY TO EVER GET CONTACT DURING TIME INTERVAL


-- CHECK THAT SQUARE ROOT OF <=0 IS NOT ATTEMPTED!!!!!


-- Author: Tommy Ingulfsen, July 2002.

package body Proximity
is

    procedure Convert(I1x: in Integer; I1y: in Integer; I1z: in Integer;
                      I2x: in Integer; I2y: in Integer; I2z: in Integer;
                      I1dx: in Integer; I1dy: in Integer; I1dz: in Integer;
                      I2dx: in Integer; I2dy: in Integer; I2dz: in Integer;
                      I1xf: out Float; I1yf: out Float; I1zf: out Float;
                      I2xf: out Float; I2yf: out Float; I2zf: out Float;
                      I1dxf: out Float; I1dyf: out Float; I1dzf: out Float;
                      I2dxf: out Float; I2dyf: out Float; I2dzf: out Float)
    is
    begin

        I1xf := Float (I1x);
        I1yf := Float (I1y);
        I1zf := Float (I1z);
        I2xf := Float (I2x);
        I2yf := Float (I2y);
        I2zf := Float (I2z);
        I1dxf := Float (I1dx);
        I1dyf := Float (I1dy);
        I1dzf := Float (I1dz);
        I2dxf := Float (I2dx);
        I2dyf := Float (I2dy);
        I2dzf := Float (I2dz);
    end Convert;


    procedure Part_Calc(Part_Root: out Float;
                        Line1ns: in Float;
                        Line1nd: in Float;
```

```
                    Line2ns: in Float;
                    Line2nd: in Float;
                    Time: in Float)
is
  Root1, Root2, Root3, Root4, Root5: Float;
begin
    -- The calculations here are the repetitive part of the entity under the
    -- square root in the formula for the actual distance between two points on
    -- two lines at time Time.

    Root1 := Line1ns * Line1ns + 2.0 * Line1ns * Time * Line1nd;
    Root2 := 2.0 * Line1ns * Line2ns - 2.0 * Line1ns * Time * Line2nd;
    Root3 := Time * Time * Line1nd * Line1nd - 2.0 * Line2ns * Time * Line1nd;
    Root4 := -2.0 * Time * Time * Line1nd * Line2nd + Line2ns * Line2ns;
    Root5 := Time * Time * Line2nd * Line2nd;

    Part_Root := Root1 + Root2 + Root3 + Root4 + Root5;

    Spark_IO.Put_String(Spark_IO.Standard_Output, "Part_Root = ", 0);
    Spark_IO.Put_Float(Spark_IO.Standard_Output, Part_Root, 0, 8, 0);
end Part_Calc;


-- Finds the distance between two points on two lines, given the vector and the
-- time parameters.
procedure Actual_Distance(Line_1: in Line_Utils.Line;
                          Line_2: in Line_Utils.Line;
                          Time: in Float;
                          Smallest_Distance: out Float)
is
    X1, X2, X3: Float;
    Y1, Y2, Y3: Float;
    Adder1, Adder2, Adder3: Float;
    Under: Float;
begin
    X1 := Float (Line_1.X_Start) + Time * Float (Line_1.X_Direction);
    X2 := Float (Line_1.Y_Start) + Time * Float (Line_1.Y_Direction);
    X3 := Float (Line_1.Z_Start) + Time * Float (Line_1.Z_Direction);
    Y1 := Float (Line_2.X_Start) + Time * Float (Line_2.X_Direction);
    Y2 := Float (Line_2.Y_Start) + Time * Float (Line_2.Y_Direction);
    Y3 := Float (Line_2.Z_Start) + Time * Float (Line_2.Z_Direction);

    Adder1 := (X1 - Y1) * (X1 - Y1);
    Adder2 := (X2 - Y2) * (X2 - Y2);
    Adder3 := (X3 - Y3) * (X3 - Y3);

    Under := Adder1 + Adder2 + Adder3;

    Smallest_Distance := Square_Package.Square_Root(Under);
end Actual_Distance;
```

```
 procedure Min_Distance(Line_1: in Line_Utils.Line;
                        Line_2: in Line_Utils.Line;
Result: out Boolean)
 is
     -- Finds the distance between two points on two lines, given the vector and the
     -- time parameters.
     procedure Actual_Distance(Line_1: in Line_Utils.Line;
                               Line_2: in Line_Utils.Line;
                               Time: in Float;
                               Smallest_Distance: out Float)
     is
         X1, X2, X3: Float;
         Y1, Y2, Y3: Float;
         Adder1, Adder2, Adder3: Float;
         Under: Float;
     begin
         X1 := Float (Line_1.X_Start) + Time * Float (Line_1.X_Direction);
         X2 := Float (Line_1.Y_Start) + Time * Float (Line_1.Y_Direction);
         X3 := Float (Line_1.Z_Start) + Time * Float (Line_1.Z_Direction);
         Y1 := Float (Line_2.X_Start) + Time * Float (Line_2.X_Direction);
         Y2 := Float (Line_2.Y_Start) + Time * Float (Line_2.Y_Direction);
         Y3 := Float (Line_2.Z_Start) + Time * Float (Line_2.Z_Direction);

         Adder1 := (X1 - Y1) * (X1 - Y1);
         Adder2 := (X2 - Y2) * (X2 - Y2);
         Adder3 := (X3 - Y3) * (X3 - Y3);

         Under := Adder1 + Adder2 + Adder3;

         Smallest_Distance := Square_Package.Square_Root(Under);
     end Actual_Distance;


     Numerator: Integer;
     Denominator: Integer;  -- must never be 0!
     Time, Distance: Float;
     Top1: Integer;
     Top2: Integer;
     Top3, Top4, Top5: Integer;
     Bottom1: Integer;
     Bottom2: Integer;
     Bottom3: Integer;
     Bottom4: Integer;
     Bottom5: Integer;
     --Root1, Root2, Root3: Float;
     --Total_Argument: Float;
     Current1, Current2, Current3: Integer;
     Current_Distance: Integer;
     Result_Set: Boolean;
```

```
    --Line1X_Start: Float;
    --Line1Y_Start, Line1Z_Start: Float;
    --Line1X_Direction, Line1Y_Direction, Line1Z_Direction: Float;
    --Line2X_Start, Line2Y_Start, Line2Z_Start: Float;
    --Line2X_Direction, Line2Y_Direction, Line2Z_Direction: Float;


begin

    -- Initialise result
    Result := False;
    Result_Set := False;

    -- First, check if the current distance between two aircraft is too small.
    Current1 := Line_1.X_Start - Line_2.X_Start;
    Current1 := Current1 * Current1;
    Current2 := Line_1.Y_Start - Line_2.Y_Start;
    Current2 := Current2 * Current2;
    Current3 := Line_1.Z_Start - Line_2.Z_Start;
    Current3 := Current3 * Current3;

    Current_Distance := Current1 + Current2 + Current3;
    Distance := Square_Package.Square_Root(Float (Current_Distance));
    if Integer (Distance) <= Line_Utils.Min_Separation then
        --Result := True;
        --Result_Set := True;
        Spark_IO.Put_String(Spark_IO.Standard_Output, "Already too close!", 0);
        Spark_IO.Put_String(Spark_IO.Standard_Output, "Distance = ", 0);
        Spark_IO.Put_Float(Spark_IO.Standard_Output, Distance, 0, 8, 0);
    end if;

    -- Check for paralell flight paths. If paralell, then the
    -- distance will be determined by the current distance between the two
    -- flights. So result will already have been set (if too close)
    -- by the above statements.
    if Line_1.X_Direction = Line_2.X_Direction and
       Line_1.Y_Direction = Line_2.Y_Direction and
       Line_1.Z_Direction = Line_2.Z_Direction then
        Result_Set := True;
        Spark_IO.Put_String(Spark_IO.Standard_Output, "Paralell", 0);
    end if;

    if Result_Set = False then

        Top1 := Line_1.X_Start * Line_1.X_Direction -
        Line_1.X_Start * Line_2.X_Direction
                - Line_2.X_Start * Line_1.X_Direction;
        Top2 := Line_2.X_Start * Line_2.X_Direction +
        Line_1.Y_Start * Line_1.Y_Direction
- Line_1.Y_Start * Line_2.Y_Direction;
        Top3 := -Line_2.Y_Start * Line_1.Y_Direction +
```

```
       Line_2.Y_Start * Line_2.Y_Direction;
       Top4 := Line_1.Z_Start * Line_1.Z_Direction -
       Line_1.Z_Start * Line_2.Z_Direction -
Line_2.Z_Start * Line_1.Z_Direction;
       Top5 := Line_2.Z_Start * Line_2.Z_Direction;


       Numerator := Top1 + Top2 + Top3 + Top4 + Top5;
       Numerator := - Numerator;

       Bottom1 := Line_1.X_Direction * Line_1.X_Direction - 2 *
          Line_1.X_Direction
    * Line_2.X_Direction;
       Bottom2 := Line_2.X_Direction * Line_2.X_Direction +
          Line_1.Y_Direction *
  Line_1.Y_Direction;
       Bottom3 := -2 * Line_1.Y_Direction * Line_2.Y_Direction +
          Line_2.Y_Direction
  * Line_2.Y_Direction;
       Bottom4 := Line_1.Z_Direction * Line_1.Z_Direction - 2 *
          Line_1.Z_Direction
   * Line_2.Z_Direction;
       Bottom5 := Line_2.Z_Direction * Line_2.Z_Direction;

       Denominator := Bottom1 + Bottom2 + Bottom3 + Bottom4 + Bottom5;

       Time := Float (Numerator) / Float (Denominator);

       Spark_IO.Put_String(Spark_IO.Standard_Output, "Numerator = ", 0);
       Spark_IO.Put_Integer(Spark_IO.Standard_Output, Numerator, 0, 10);
       Spark_IO.New_Line(Spark_IO.Standard_Output, 1);
       Spark_IO.Put_String(Spark_IO.Standard_Output, "Denominator = ", 0);
       Spark_IO.Put_Integer(Spark_IO.Standard_Output, Denominator, 0, 10);
       Spark_IO.New_Line(Spark_IO.Standard_Output, 1);


       Spark_IO.Put_String(Spark_IO.Standard_Output, "Time = ", 0);
       Spark_IO.Put_Float(Spark_IO.Standard_Output, Time, 0, 8, 0);
       Spark_IO.New_Line(Spark_IO.Standard_Output, 1);

       if Time in Line_Utils.Time_Interval
       then

           -- calculate actual minimum distance between the aircraft

           Actual_Distance(Line_1, Line_2, Time, Distance);
           Spark_IO.Put_String(Spark_IO.Standard_Output, "Distance = ", 0);
           Spark_IO.Put_Float(Spark_IO.Standard_Output, Distance, 0, 8, 0);

           if Integer (Distance) <= Line_Utils.Min_Separation then
               Result := True;
           else Result := False;
```

```
                end if;

            else Result := False;
            end if;
        end if;
    end Min_Distance;
end Proximity;
```

E generate.ads - code for Generator

```
-- File "generate.ads" - specification of the Generator. The Generator
-- creates and deletes aircraft and controls their course and position.

-- Author: Tommy Ingulfsen, July 2002

with Aircrafts;
use Aircrafts;

package Generate
is
    type Which_Gate is (G_1, G_2, G_3, G_4);

    Z_Max: constant Integer := 66667;
    Z_Min: constant Integer := 0;
    subtype Y_Range is Integer range 500..5500;  -- same everywhere in radar system
    subtype X14_Range is Integer range 0..4000;
    subtype X23_Range is Integer range 20000..24000;

    -- 4 entry/exit points for aircraft, defined as an X. Gate_1
    -- is the lower left endpoint and they are numbered anti-clockwise
    type Gate_1 is
       record
           X_First: Integer := X14_Range'First;
           X_Last: Integer := X14_Range'Last;
           Z: Integer := Z_Min;
       end record;

    type Gate_2 is
       record
           X_First: Integer := X23_Range'First;
           X_Last: Integer := X23_Range'Last;
           Z: Integer := Z_Min;
       end record;

    type Gate_3 is
       record
           X_First: Integer := X23_Range'First;
           X_Last: Integer := X23_Range'Last;
           Z: Integer := Z_Max;
       end record;

    type Gate_4 is
```

```
        record
            X_First: Integer := X14_Range'First;
            X_Last: Integer := X14_Range'Last;
            Z: Integer := Z_Max;
        end record;


    -- probability constants
    P_New_Aircraft: constant Integer := 7;    -- with P=0.142857 introduce new aircraft
    P_Change_Altitude: constant Integer:= 10; -- with P=0.1 change altitude of aircraft

    -- subprograms
    -- introduces a new aircraft into the system
    procedure New_Aircraft(Transponder_View: in out Aircraft);

    -- changes altitude of an aircraft
    procedure Change_Altitude(Transponder_View: in out One_Aircraft);

    -- removes an aircraft that's no longer within radar range
    --procedure Remove_Aircraft(Transponder_View: in out Aircraft);

    -- moves all aircraft forward one step
    --procedure Update_System(Transponder_View: in out Aircraft);

    -- "main" program for Generator
    --procedure Generate(Transponder_View: in out Aircraft);

private
    -- utility for New_Aircraft
    procedure Get_Y(Flying: out One_Aircraft);

    -- utility for New_Aircraft
    procedure Get_Callsign(Flying: out One_Aircraft);

end Generate;
```

F generate.adb - code for Generator

```
-- USE FUNCTIONS/PROCEDURES TO SHORTEN THIS DOWN!!
-- ADD UPDATE OF New_Planes to New_Aircraft!
-- DOCUMENT USE OF Ada.Numerics.Float_Random IN Get_Callsign!!!



-- File "generate.adb" - implementation of the Generator part
-- of the system.

-- Author: Tommy Ingulfsen, July 2002

with Ada.Numerics.Float_Random;
```

```
with Ada.Numerics.Discrete_Random;
with Ada.Text_IO;
with Ada.Integer_Text_IO;
with Ada.Strings.Unbounded;
use Ada.Strings.Unbounded;
use Ada.Text_IO;
use Ada.Integer_Text_IO;
with Aircrafts;



package body Generate
is

    procedure Get_Y(Flying: out One_Aircraft)
    is
        Temp_Y: Integer := -1;
        G: Ada.Numerics.Float_Random.Generator;

    begin
        Ada.Numerics.Float_Random.Reset(G);

        while Flying.Y < Generate.Y_Range'First loop
            Flying.Y := Integer (Ada.Numerics.Float_Random.Random(G)
                      * Float (Generate.Y_Range'Last));
        end loop;

        while Temp_Y < Generate.Y_Range'First loop
            Temp_Y := Integer (Ada.Numerics.Float_Random.Random(G)
                      * Float (Generate.Y_Range'Last));
        end loop;
        Flying.Delta_Y := Temp_Y - Flying.Y;
    end Get_Y;


    procedure Get_Callsign(Flying: out One_Aircraft)
    is
        function Get_Capital return Character
        is

            -- in ASCII table, capital letters are in 65..90
            subtype Capital_Range is Integer range 65..90;
            package Random_Cap is new Ada.Numerics.Discrete_Random(Capital_Range);
            use Random_Cap;

            G: Generator;
            GF: Ada.Numerics.Float_Random.Generator;
            Seed: Integer;
            Result: Character;
        begin
            -- get random float number to initialise the
            -- discrete random generator
```

```
        Ada.Numerics.Float_Random.Reset(GF);
        Seed := Integer (Ada.Numerics.Float_Random.Random(GF) * 999.0);
        New_Line;
        Put(Seed);
        -- initialise discrete random no generator and get
        -- a capital letter
        Reset(G, Seed);
        Result := Character'Val(Random(G));
        return Result;
    end Get_Capital;

    subtype Int_Range is Integer range  0..9;
    package Random_Char is new Ada.Numerics.Discrete_Random(Int_Range);
    use Random_Char;

    G: Generator;
    Temp: Character;
    New_Callsign: Unbounded_String;
begin

    Reset(G);

    -- get the first two capital letters
    for I in 1..2 loop
        Temp := Get_Capital;
        Put(Temp);
        New_Line;
        Append(New_Callsign, Temp);
    end loop;


    -- get the 4 digits at the end of a callsign
    for I in 3..6 loop

        -- char values for decimal digits are in range 48-59 inclusive
        Temp := Character'Val( (Random(G) + 48) );
        Append(New_Callsign, Temp);
    end loop;

    -- copy temporary result into aircraft attributes when finished
    Flying.Call := To_String(New_Callsign);
end Get_Callsign;





procedure New_Aircraft(Transponder_View: in out Aircraft)
is
    -- needed to use Ada.Numerics.Discrete_Random
    package Random_Gate is new Ada.Numerics.Discrete_Random(Which_Gate);
    use Random_Gate;
```

```
    G: Generator;
    GF: Ada.Numerics.Float_Random.Generator;
    Start_Gate: Which_Gate := G_1;
    End_Gate: Which_Gate;
    Plane: One_Aircraft;
    Entry_1: Gate_1;
    Entry_2: Gate_2;
    Entry_3: Gate_3;
    Entry_4: Gate_4;

    Temp_X: Integer := -1;
    Temp_Y: Integer := -1;


begin

    -- initialise Generator on every execution
    Reset(G);
    Ada.Numerics.Float_Random.Reset(GF);

    -- randomly choose a gate
    Start_Gate := Random(G);

    Put("The starting gate is ");
    if Start_Gate = G_1 then
        End_Gate := G_3;
        elsif Start_Gate = G_2 then
            End_Gate := G_4;
            Put("G_2");
        elsif Start_Gate = G_3 then
            End_Gate := G_1;
            Put("G_3");
        elsif Start_Gate = G_4 then
            End_Gate := G_2;
            Put("G_4");
    end if;

    -- make a function to create random strings of the right format
    Get_Callsign(Plane);
    --Plane.Call := "BA1441";
    Plane.Y := 0;

    case Start_Gate is
       when G_1 =>
           Plane.X := Integer (Float (Entry_1.X_Last) *
                   Ada.Numerics.Float_Random.Random(GF));
           Plane.Z := Entry_1.Z;

           Get_Y(Plane);

           Plane.Delta_Z := Entry_3.Z - Entry_1.Z;
```

```
        while Temp_X < Entry_3.X_First loop
            Temp_X := Integer(Ada.Numerics.Float_Random.Random(GF)
                   * Float (Entry_3.X_Last));
        end loop;
        Plane.Delta_X := Temp_X - Plane.X;

    when G_2 =>
        while Plane.X < Entry_2.X_First loop
            Plane.X := Integer(Ada.Numerics.Float_Random.Random(GF)
    * Float (Entry_2.X_Last));
        end loop;

        Plane.Z := Entry_2.Z;
        Get_Y(Plane);

        Plane.Delta_Z := Entry_4.Z - Entry_2.Z;

        while Temp_X <= Entry_4.X_First loop
            Temp_X := Integer(Ada.Numerics.Float_Random.Random(GF)
       * Float (Entry_4.X_Last));
        end loop;
        Plane.Delta_X := Temp_X - Plane.X;

    when G_3 =>
        while Plane.X < Entry_3.X_First loop
            Plane.X := Integer(Ada.Numerics.Float_Random.Random(GF)
    * Float (Entry_3.X_Last));
        end loop;

        Plane.Z := Entry_3.Z;
        Get_Y(Plane);

        Plane.Delta_Z := Entry_1.Z - Entry_3.Z;

        while Temp_X <= Entry_1.X_First loop
            Temp_X := Integer(Ada.Numerics.Float_Random.Random(GF)
        * Float (Entry_1.X_Last));
        end loop;
        Plane.Delta_X := Temp_X - Plane.X;

    when G_4 =>
        while Plane.X < Entry_4.X_First loop
            Plane.X := Integer(Ada.Numerics.Float_Random.Random(GF)
    * Float (Entry_4.X_Last));
        end loop;

        Plane.Z := Entry_4.Z;
        Get_Y(Plane);

        Plane.Delta_Z := Entry_2.Z - Entry_4.Z;
```

```
            while Temp_X <= Entry_2.X_First loop
                Temp_X := Integer(Ada.Numerics.Float_Random.Random(GF)
      * Float (Entry_2.X_Last));
            end loop;
            Plane.Delta_X := Temp_X - Plane.X;
    end case;

    New_Line;
    Put("The attributes of the new one are: ");
    New_Line;
    Put("Callsign  ");
    Put(Plane.Call);
    New_Line;
    Put("X-start   ");
    Put(Plane.X);
    New_Line;
    Put("Y-start   ");
    Put(Plane.Y);
    New_Line;
    Put("Z-start   ");
    Put(Plane.Z);
    New_Line;
    Put("Delta X   ");
    Put(Plane.Delta_X);
    New_Line;
    Put("Delta Y   ");
    Put(Plane.Delta_Y);
    New_Line;
    Put("Delta Z   ");
    Put(Plane.Delta_Z);
end New_Aircraft;


-- Changes the altitude of an aircraft by changing its Y-vector
procedure Change_Altitude(Transponder_View: in out One_Aircraft)
is
    Temp_Aircraft: One_Aircraft := Transponder_View;
begin
    Get_Y(Temp_Aircraft);

    -- ensure that altitude vector REALLY changes
    if Temp_Aircraft.Delta_Y = Transponder_View.Delta_Y then
        while Temp_Aircraft.Delta_Y = Transponder_View.Delta_Y loop
            Get_Y(Temp_Aircraft);
        end loop;
    end if;

    -- finally change altitude
    Transponder_View.Delta_Y := Temp_Aircraft.Delta_Y;
end Change_Altitude;
```

```
end Generate;
```