Tommy Ingulfsen
July 30, 2002

Pushing Tin Project - Third Design Draft

1 Introduction

This document contains the full design for the SPARK Automatic Collision Detection System (ACDS) and a partial design for the rest of the application. It builds upon the Second Design Draft, as laid out in "draft2.ps".

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.

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).

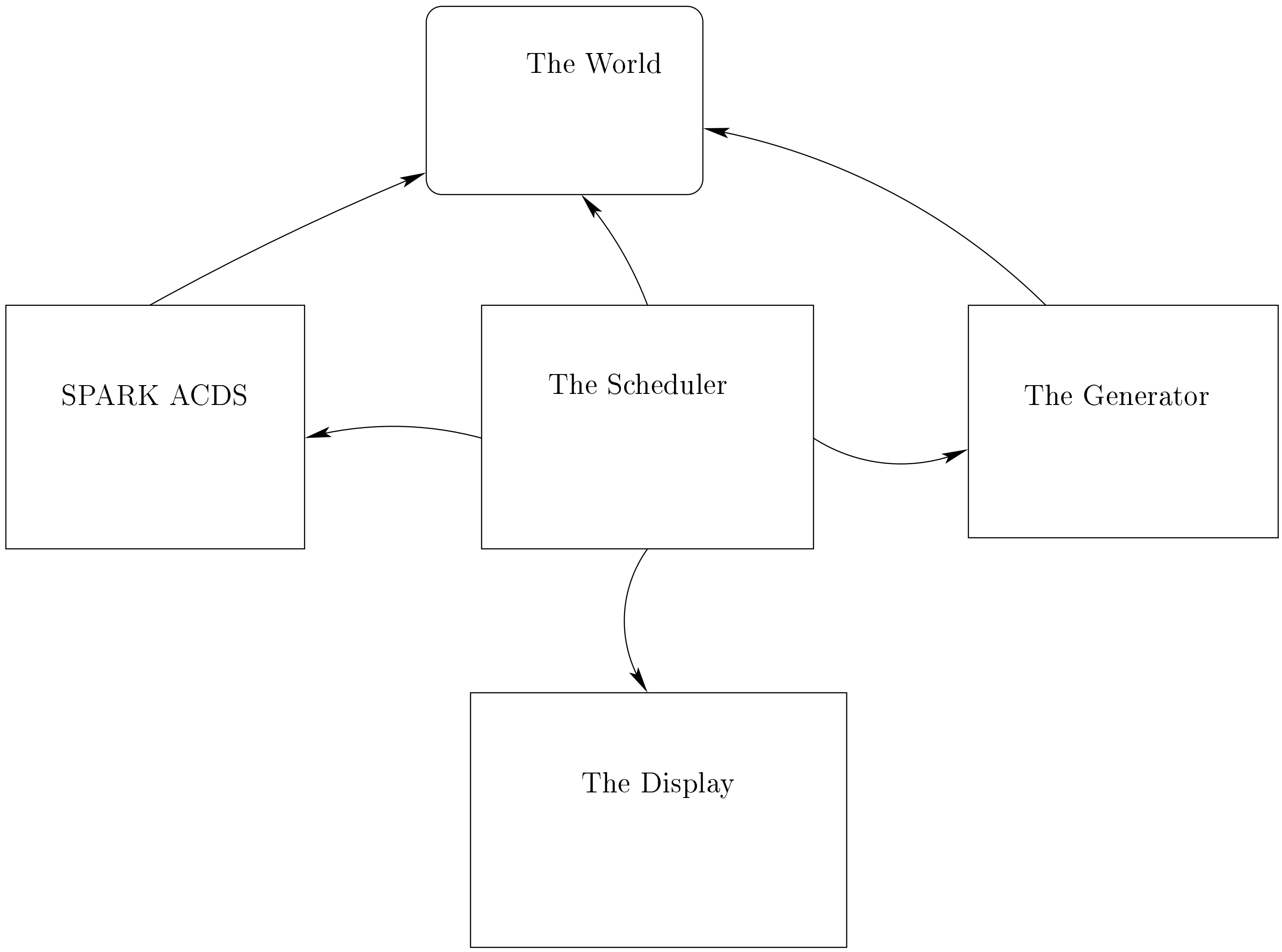

Figure 1: System Architecture

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 world. 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 sub programs, the current state of the world 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" (in the appendix).

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)
- Some form of mechanism to extract the next position of an aircraft from the current/prior positions (variable package - to be defined)

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. Others will be coded up during implementation.

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:

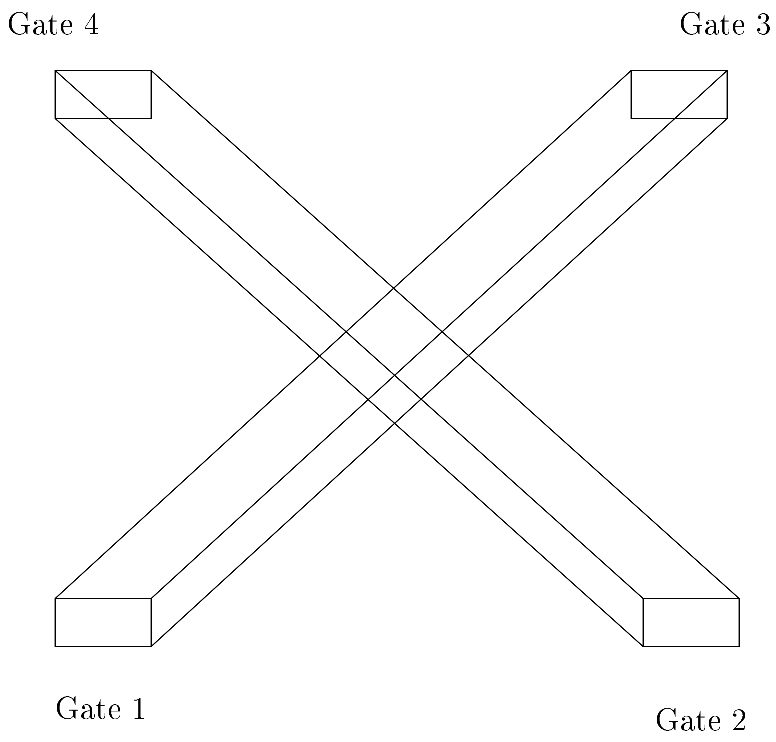

Figure 2: Airways

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 3 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

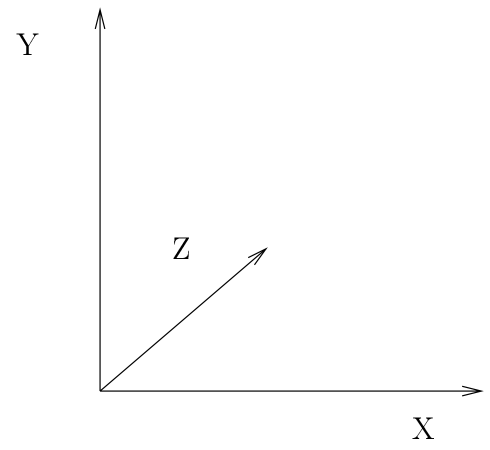

Figure 3: Co-ordinate System

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. The Scheduler will read information from this before passing on the co-ordinates to the ACDS.

5 Scheduler Design

The Scheduler will be the "main" sub program 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.

6 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.

7 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;
```