CS-Note* 27

Tommy Ingulfsen
August 16, 2002

Pushing Tin Project - Amended Design Draft 5

1 Introduction

This document is identical to Design Draft 5 apart from the last section, which explains how the system was actually implemented and what changes were made to the design.

2 System Architecture

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

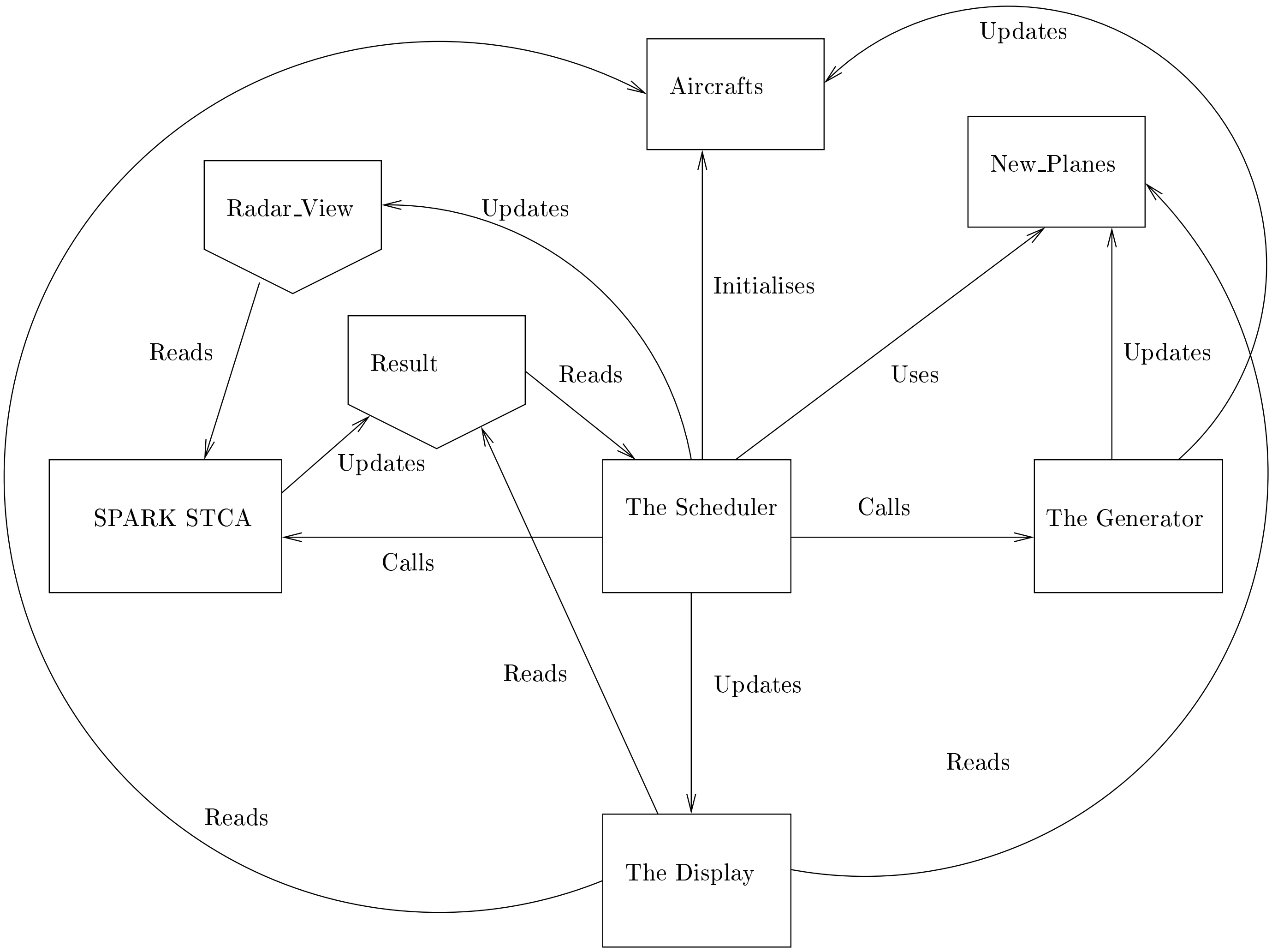

Figure 1: System Architecture

*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).

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 `STCA` that finds out if any aircraft are in danger of colliding and the `Display` that provides output to the user.

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 `STCA` 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 STCA Design

As mentioned, the INFORMED method of Praxis has been used for the `STCA` design. 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.

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 STCA:

```
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 Alert (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)
- `Situation` (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

The STCA will consist of two packages, one called STCA and another called Situation. As the name suggests, STCA will hold the main program. In addition it will also be made up by procedures to derive the aircraft paths from the Radar_View and to check the distances between aircraft. The Situation variable package is going to provide persistent storage. It will contain a data structure required to store the previous and current state of the radar image. Associated with this will be some procedures to initialise and update or read its contents. The data structure will be an own variable that is refined within the package, to ease the information flow annotations. Example derives annotations are shown below.

Note that Situation will be outside the SPARK boundary. This is to get nice derives annotations for the other procedures. Ideally, Situation would be inside the boundary, but this seems to complicate the code unnecessarily. Situation might be brought inside the boundary at a later stage.

There will also be a package called Line_Utils that assist the procedures in STCA. This will either be out width the SPARK boundary like Situation or be a private child package of STCA. Line_Utils is going to be a type package, defining types and procedures.

The "main" collision detection subprogram will be rather short. It will consist of calls to update the Situation 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 STCA.Proximity procedures will update the Result boundary package.

Given these packages and procedures, the STCA package could contain the following procedures with annotations:

```
procedure UpdateSituation;
--# global in out Situation.Stat
--#        in RadarNow.State;
--# derives Situation.State from Situation.State, RadarNow.State;

procedure Get_Lines(Lines: out Line_Array);
--# global in Situation.state;
--# derives Line_Array from Situation.State;

procedure Proximity(Lines: in Line_Array);
--# global out Results.State;
--# derives Results.State from LineInfo;
```

and these might in turn be broken down into smaller procedures or functions.

A typical execution of the SPARK Short-Term Collision Alert system will take the following form:

1. The Generator updates Aircrafts to hold the new picture of the radar.
2. The Scheduler updates Radar_View with the new data.
3. The Scheduler calls The SPARK STCA.
4. The SPARK STCA reads Radar_View.
5. The SPARK STCA uses data from the Radar_View to update the current Situation

6. The SPARK STCA derives the aircraft paths using Situation.

7. The SPARK STCA performs a proximity check to find out if any flight paths are hazardous.

8. The SPARK STCA updates the Result.

9. The Scheduler 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. The external packages Situation and Line_Utils are not shown on figure 1 because they are not used by anything outside SPARK-implemented subsystem. The INFORMED design notation is used.

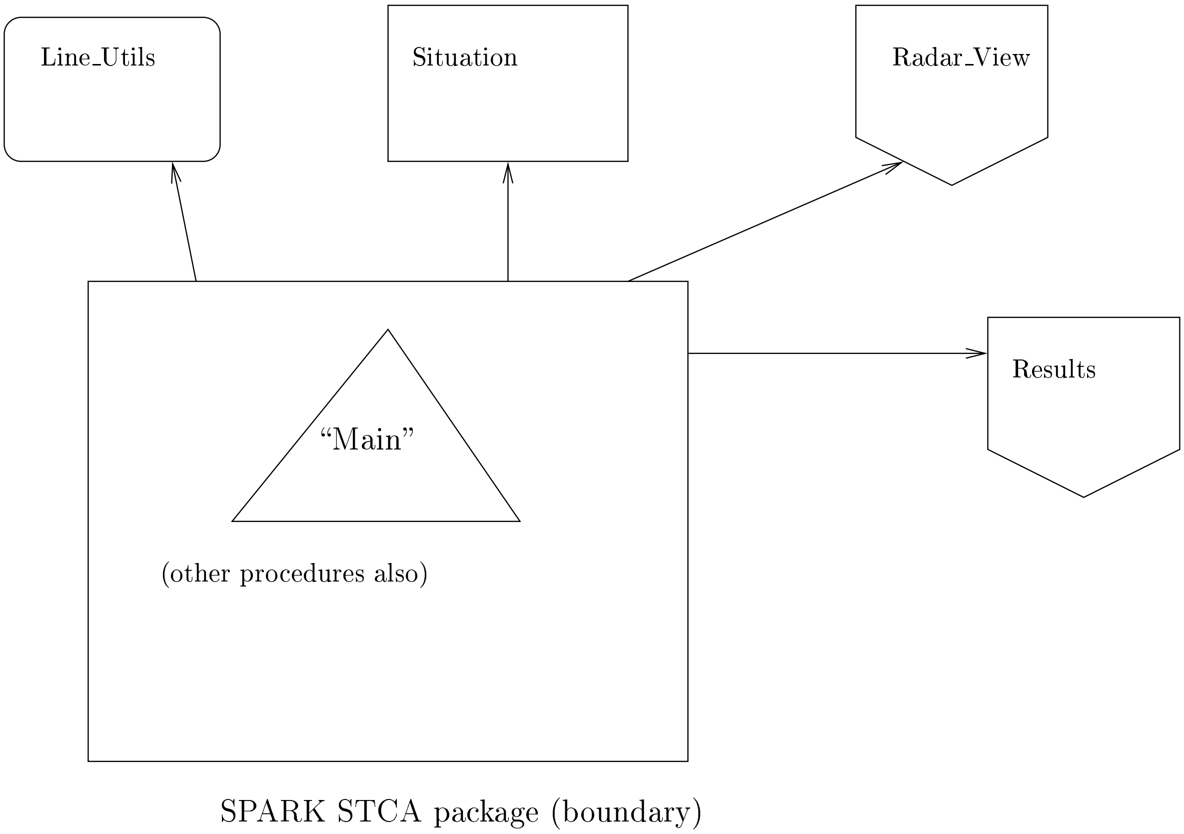

Figure 2: SPARK STCA

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, as can be seen in figure 3.

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 "VORs". A VOR beacon is a Very High Frequency Omnidirectional Range radio transmitter used to assist aircraft navigation. The VORs are given fixed co-ordinates according to the system given in figure 4 below.

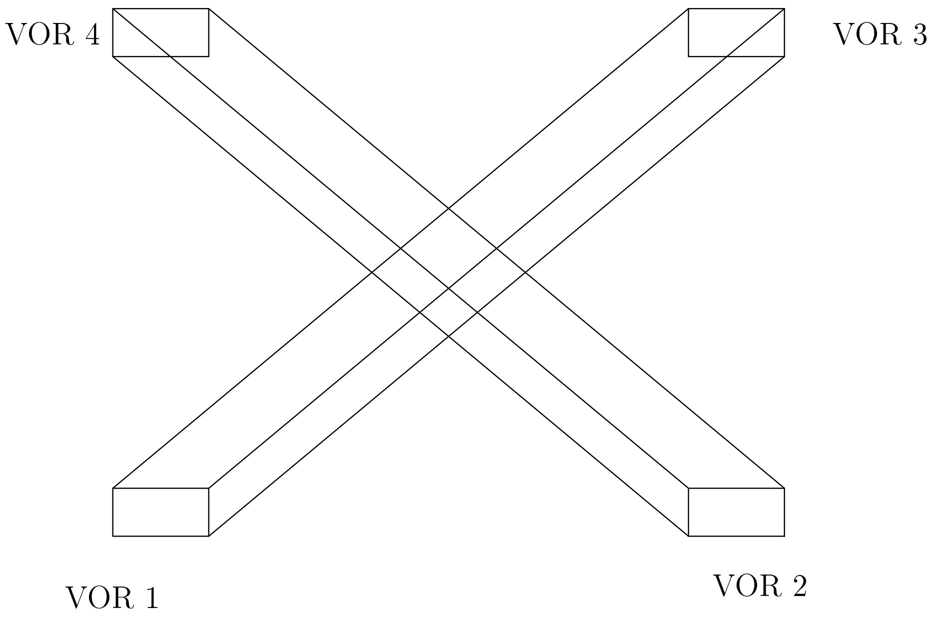

Figure 3: Airways

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:

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

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

X range:

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

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.

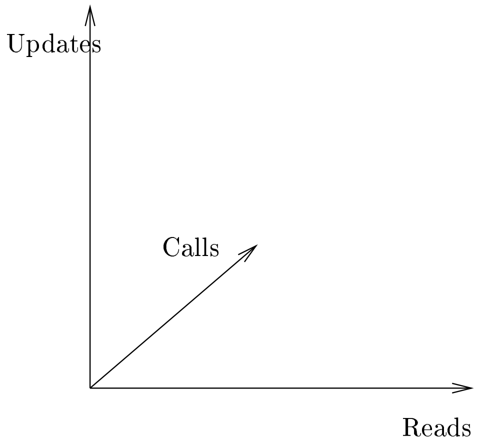

Figure 4: Co-ordinate System

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 Design Draft 4, 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. So the effect is not a “jump” to a different altitude, but a change in the course (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 updated by the `Generator`. See “aircrafts.ads”. The `Scheduler` will read information from this before passing on the co-ordinates to the STCA.

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)
- 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 `Aircrafts`, `Result`, `Radar_View` and `New_Planes` will be carried out.

In the loop there will be calls to the `Generator` to update the situation, calls to the `STCA` 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 Summary of globally available data structures

The following packages will be available to the entire program:

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

Again, `Situation` and `Line_Utils` are not included in the list, since they are only used by the Short-Term Conflict Alert System.

7 Display Design

To begin with, the `Display` component will be basic. It will negotiate Result 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 `STCA` 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 STCA 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 `SPARK STCA`) has been coded and at least some testing carried out.

9 The Final Design

The design laid out in Design Draft 5 proved to be good enough to implement, but as expected some additions had to be made. These include:

- The `Aircrafts` data structure was implemented in the package `Airport`. The name `Aircrafts` was kept for the Ada specification file that gives all the types used for aircraft.

- The `Globals` package and "line_utils.ads" contain global constants and utilities for working with lines and vectors.
- The `Square_Package` package implements a SPARK version of the square root operator.
- "points.ads" gives a type for aircraft positions in the form of a single point in space.
- A graphical display written in OpenGL was added to give a visualisation of the program execution in addition to the simple output from the Display module. The graphical display contains the Scheduler.
- In order to accommodate the OpenGL constructs, an additional file "stca.adb" was created to hold the main graphical program. The package `Push_Tin` holds the Scheduler itself in addition to the graphical display subprogram.
- For simplicity, the SPARK STCA subsystem was implemented as a collection of packages rather than a hierarchy. In practise only 2 packages were needed, `Spark_Subsys` (the "main" SPARK package) and `Proximity`, the proximity checker.

In all, these changes are relatively minor for a system of this size, and the underlying architecture has been kept.