SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 00-25

Formal Modelling of an Air-Traffic Control Simulator

David Leadbetter, Peter Lindsay and Andrew Hussey

December, 2000

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Formal Modelling of an Air-Traffic Control Simulator

David Leadbetter, Peter Lindsay and Andrew Hussey

Abstract

This technical report describes a simulator for a simple Air-Traffic Control system. The simulator is being used in the SafeHCI project to study modelling of cognitive processes and prediction of operator errors, particularly as they relate to Human Computer Interface (HCI) design. The HCI and core functionality of the simulator are formally modelled here in a combination of the Z and UAN notations. An appendix describes the models in more detail for readers unfamiliar with the formal notations.

Keywords human computer interface, Z, UAN, air-traffic control.

Contents

1 Introduction

Growing use of computers in safety-critical systems increases the need for Human Computer Interfaces (HCIs) to be both smarter – to detect human errors – and better designed – to reduce likelihood of errors. The SafeHCI project aims to develop a method for analysing hazards and error rates related to operator activities within interactive systems. The approach taken in the project builds on combining improved understanding of the psychological causes of human errors, with formal methods for modelling the operator's cognitive process and the human-computer interaction in which they engage.

The SafeHCI method is being developed on a highly simplified Air-Traffic Control (ATC) case study. Colleagues in the University of Queensland's Key Centre for Human Factors and Applied Cognitive Psychology are using a combination of psychological theories and simulator-based experiments to develop models of the cognitive processes underlying the ATC task for the case study. We are formalising these models and relating them to models of the simulator's functionality and HCI. The combined models are then being used to predict how errors might occur and the part played by the HCI in detecting and/or preventing operator errors.

This report describes the simulator and models it formally in a combination of Z [4] and the User Action Notation (UAN) [1]. Z is a widely used formal notation for describing data structures and state-based systems; the latter are modelled by describing the different (abstract) states of the system and transitions between states. UAN is a simple notation for describing "the behaviour of the user and the interface as they perform a task together" [1].

The report is structured as follows: Section 2 describes the ATC simulator and the operator task being used in the case study. Section 3 models the simulator's core functionality in Z. Section 4 models the simulator's HCI in a combination of Z and UAN; it includes an outline of UAN notation and the extensions we have introduced to improve its integration with Z and to model certain aspects of the case study. An appendix describes the models in more detail for readers unfamiliar with the formal notations.

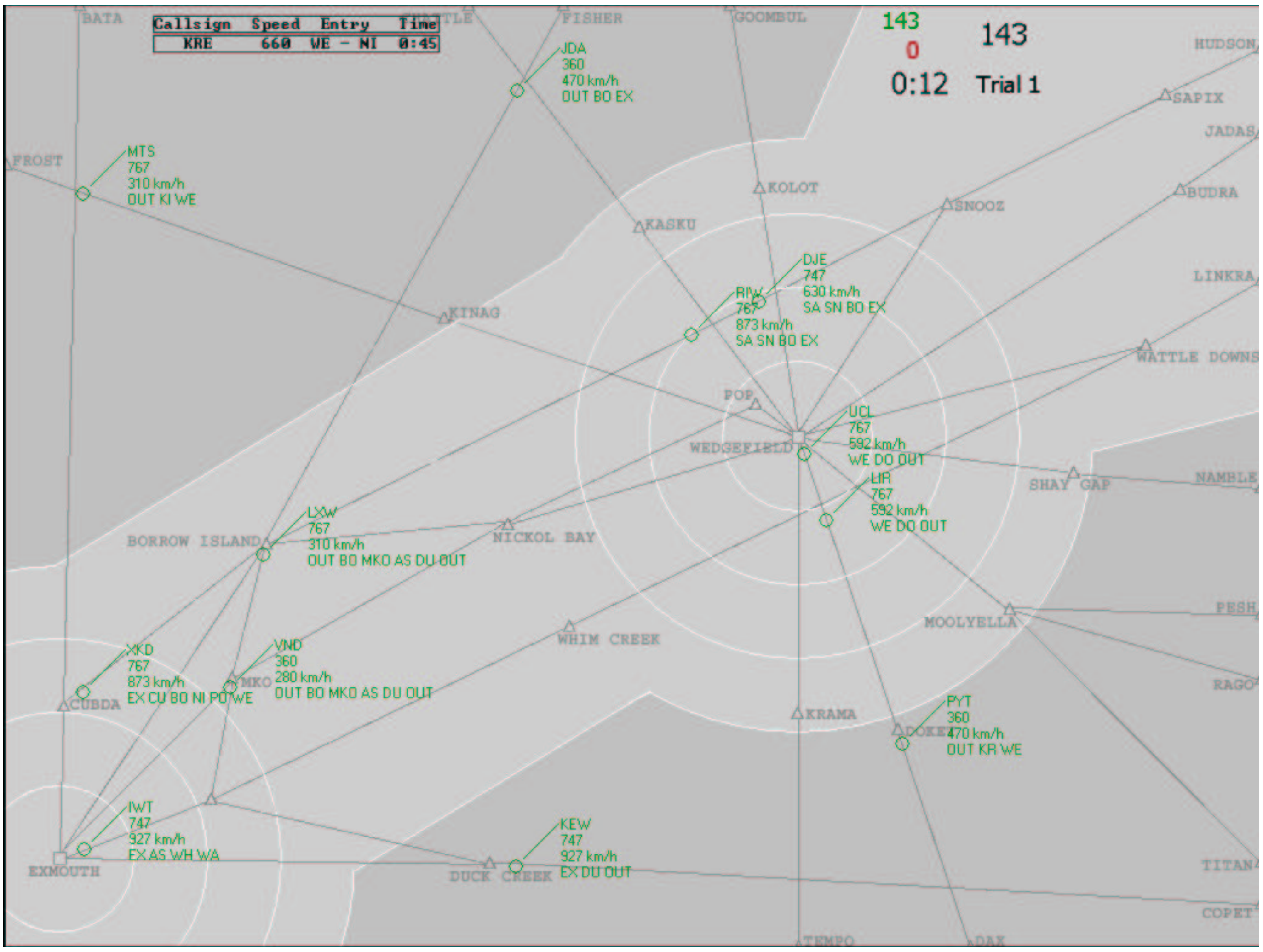
The reader is referred to other SafeHCI reports [2, 3] for more details of the cognitive models for the case study and how they are being used to analyse the possibility of operator errors.

2 Overview of the ATC case study

This section describes the ATC simulator and the ATC task which is the subject of the SafeHCI case study.

2.1 Description of the ATC system

The ATC simulator has a display which depicts a simulated sector of airspace – consisting of airports, waypoints and flight paths – together with the location and details of aircraft currently flying within the sector. A screendump of the simulator is shown in Figure 1. As seen in this figure, each aircraft in the sector is represented by a circle appropriately positioned on the displayed flight paths. Airports are shown as squares and the waypoints are displayed as triangles. The details of each aircraft (the call sign, aircraft type, speed, and flight route) are shown near the aircraft. in the top left hand corner of the screen is a table of incoming aircraft. Included in this list are both those planes entering the sector from the outside and those entering from airports. For simplicity the waypoints in an aircraft's flight route are represented as codes. Some experimental data is shown in the top right hand corner, but is not described further.



Callsign	Speed	Entry	Time
KRE	660	WE - NI	0:45

143
0
0:12
143
Trial 1
BATA
FISHER
GOOMBUL
HUDSON
SAPIX
JADAS
BUDRA
LINKRA
WATTLE DOWNS
NAMBLE
PESH
RAGO
TITAN
COPET
DAX
TEMPO
FROST
KOLOT
SNOOZ
KASKU
KINAG
POP
WEDGEFIELD
SHAY GAP
BORROW ISLAND
NICKOL BAY
WHIM CREEK
MOOLYELLA
KRAMA
DOKER
CUBDA
MKO
EXMOUTH
DUCK CREEK
JDA 360 470 km/h OUT BO EX
MTS 767 310 km/h OUT KI WE
DJE 747 630 km/h SA SN BO EX
RIW 767 873 km/h SA SN BO EX
UCL 767 592 km/h WE DO OUT
LIR 767 592 km/h WE DO OUT
LXW 767 310 km/h OUT BO MKO AS DU OUT
XKD 767 873 km/h EX CU BO NI PO WE
VND 360 280 km/h OUT BO MKO AS DU OUT
PYT 360 470 km/h OUT KR WE
IWT 747 927 km/h EX AS WH WA
KEW 747 927 km/h EX DU OUT

Figure 1: A screen shot of the ATC simulator

The display is updated at short intervals to give the impression that the aircraft are moving. The primary task of the "air-traffic controller" is to ensure that the aircraft moving through the sector remain separated by no less than the defined minimum separation distance.

The HCI provides the air-traffic controller with two main operations:

1. selecting an aircraft, and
2. changing the speed of the selected aircraft.

An aircraft is selected by clicking the left button when the cursor is positioned over an aircraft. The selected aircraft is indicated using a solid dot within the circle that represents the aircraft. Only one aircraft can be selected at a time - when an aircraft is selected the previously selected aircraft becomes unselected (and so no longer has the selection highlight).

An aircraft must be selected to enable display of the speed menu (shown in Figure 2). This

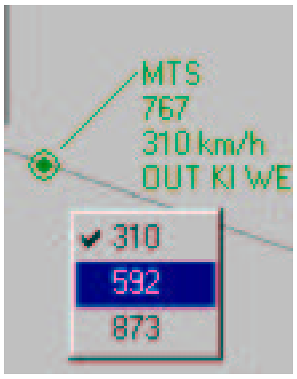


Figure 2: A screen shot of the Speed Menu and a selected aircraft

operation involves three steps:

1. Opening the speed menu (by clicking the right button);
2. Navigating the speed menu (by moving amongst the menu entries);
3. Selecting a speed (by left clicking on the desired speed).

The speed menu appears at the position of the cursor. The entries in the speed menu depend on the type of aircraft that is selected. Additionally, a tick (✓) within the menu indicates the aircraft's current speed (as shown in Figure 2). The air-traffic controller may abort this operation by clicking the left button when the cursor is positioned outside the speed menu.

If, at any time, the minimum separation distance is violated between two aircraft, this violation is alerted to the air-traffic controller using two mechanisms. Firstly, the involved aircraft are highlighted with a different colour (yellow in the case study), and secondly, an audible alarm sounds. This alert continues until the minimum separation between the two aircraft is restored.

2.2 Simplifications

For the purposes of developing the methodology, the case study has a number of obvious simplifications compared with a real ATC system. These simplifications include the following:

- Aircraft altitude is ignored. Consequently the sector is two dimensional and not three dimensional.
- Aircraft run on rails: the flight routes. The flight dynamics of the aircraft in the sector are of aircraft running on rails. This has various implications:
  - aircraft flight paths are restricted to the provided flight routes: they cannot fly freely about the sector;
  - aircraft change heading instantaneously when flying through waypoints;
- No other operation are provided in the air-traffic control system other than selecting an aircraft and changing the aircraft speed. For example, aircraft flight routes cannot be modified;
- Aircraft respond to instructions instantly, thus there is no acceleration or deceleration involved when an aircraft changes speed;

- The simulation does not include collisions, thus two aircraft in conflict will eventually fly right through each other without harm.

As a consequence of these simplifications the actions taken by the air-traffic controller to resolve potential conflicts are limited to changing the speeds of the aircraft involved. This will involve either changing the speed of one or both of the aircraft involved in a potential conflict.

2.3 Conflicts

Given the details of the air-traffic control system described above, there are two distinct types of conflict considered within our case study: overtaking conflicts and convergence conflicts.

An overtaking conflict is one in which a faster aircraft approaches a slower aircraft flying ahead of it along the same flight route. In such a situation a violation of the minimum separation distance will result if the faster aircraft catches up to the slower aircraft.

A convergence conflict in one in which two aircraft on different flight routes are both converging on a common point in the sector (and, more specifically, both aircraft are approximately the same time away from that common point). This conflict typically involves aircraft whose flight routes are converging on a common waypoint, but also includes those situations where the two flight routes cross over each other.

2.4 Scenarios

The air-traffic control simulation is run using scripted scenarios. A scenario script describes the starting positions, times, speeds, routes, etc of the aircraft involved in the scenario.

The simulator animates (in real-time) the flight of the aircraft according to their scripted details, and according to any instructions given by the air-traffic controller using the interface operations.

Each script typically presents the air-traffic controller with a number of conflicts to be resolved at different times throughout the scenario and includes both the aircraft involved in those conflicts and numerous 'filler' aircraft to populate the sector.

3 The ATC system core model

The ATC simulator core functionality can be decomposed into the following subsystems: Sector data; Aircraft specifications; Static aircraft details; Aircraft telemetry updates; and Warning systems for identifying separation violations. The HCI is modeled in Section 4.

More detailed descriptions of these models (aimed at readers not fluent in Z) are provided in Appendices A and B.

3.1 Time

Time is used throughout the ATC system, so we begin by abstractly modelling time in two forms – absolute time (*Time*), and time durations (*Duration*). The *timeDifference* function is provided to calculate the duration between two absolute times. We define a simple clock that keeps track of the current time.

$$\begin{array}{|l}
\mathit{timeDifference} : \\
\quad \mathit{Time} \times \mathit{Time} \to \mathit{Duration}
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Clock} \\
\mathit{now} : \mathit{Time} \\
\hline
\end{array}$$

3.2 Sector data

A sector contains a variety of objects: waypoints, airports, routes, etc, all based on the model of waypoints (*Waypoints*: for simplicity airports are considered to be waypoints, routes are made from waypoints, etc). Each waypoint has an associated position in the sector (*Position*).

A sector is modelled by defining the objects it contains: the *waypoints* and their positions in the sector; the *airports* in the sector; and the *routes* through the sector (where each pair of waypoints in the *routes* relation defines the single, straight route segment between the two waypoints).

$$\begin{array}{|l}
\hline
\textit{Sector} \\
\mathit{waypoints} : \mathbb{P}\, \mathit{Waypoint} \\
\mathit{position} : \mathit{Waypoint} ⇸ \mathit{Position} \\
\mathit{airports} : \mathbb{P}\, \mathit{Waypoint} \\
\mathit{routes} : \mathit{Waypoint} \leftrightarrow \mathit{Waypoint} \\
\hline
\mathit{airports} \subseteq \mathit{waypoints} \\
\mathit{waypoints} = \operatorname{dom} \mathit{position} \\
\operatorname{dom} \mathit{routes} \cup \operatorname{ran} \mathit{routes} = \mathit{waypoints} \\
\hline
\end{array}$$

3.3 Aircraft and air traffic data

We model aircraft callsigns (*Callsign*), types (*AircraftType*), and speed (*Speed*).

The air traffic in a sector consists of the set of *aircraft* in the sector (identified by callsigns). Each aircraft is of a specific *type*, and has a last known *telemetry* (modelled as a function from the callsign to the telemetry data: the position, speed, and time). Each aircraft has a flight plan (in the *flightPlan* function from the callsign to a sequence of waypoints with associated estimated times of arrival). Lastly, the ATC operator's last instruction to each aircraft is recorded (in the *instructions* function from the callsign to the instructed speed and time of instruction).

$$\begin{array}{|l}
\hline
\textit{Traffic} \\
\mathit{aircraft} : \mathbb{P}\, \mathit{Callsign} \\
\mathit{type} : \mathit{Callsign} ⇸ \mathit{AircraftType} \\
\mathit{telemetry} : \mathit{Callsign} ⇸ \mathit{Position} \times \mathit{Speed} \times \mathit{Time} \\
\mathit{flightPlan} : \mathit{Callsign} ⇸ \operatorname{seq}_1(\mathit{Waypoint} \times \mathit{Time}) \\
\mathit{instructions} : \mathit{Callsign} ⇸ (\mathit{Speed} \times \mathit{Time}) \\
\hline
\mathit{aircraft} = \operatorname{dom} \mathit{type} = \operatorname{dom} \mathit{telemetry} = \operatorname{dom} \mathit{flightPlan} \\
\operatorname{dom} \mathit{instructions} \subseteq \mathit{aircraft} \\
\hline
\end{array}$$

The air traffic telemetry information is updated using the *updateTelemetry* operation. The new telemetry for multiple aircraft is also received and updated.

$$\begin{array}{|l}
\hline \textit{updateTelemetry} \\
\Delta \textit{Traffic} \\
\textit{newTelemetry}? : \textit{Callsign} \pfun \textit{Position} \times \textit{Speed} \times \textit{Time} \\
\hline
\textit{telemetry}' = \textit{telemetry} \oplus \textit{newTelemetry}? \\
\textit{type}' = \textit{type} \\
\textit{flightPlan}' = \textit{flightPlan} \\
\textit{instructions}' = \textit{instructions} \\
\hline
\end{array}$$

This allows both frequent, regular telemetry updates of aircraft to be mixed in with infrequent, irregular telemetry updates of other aircraft. However, we shall assume for simplicity that aircraft telemetry data is updated regularly, and ignore that no means has been provided for removing information about air traffic as it leaves the sector. Given these assumptions, the information recorded in *Traffic* represents the current information on air traffic in the sector.

The operator's instructions to individual aircraft are recorded using the *changeSpeed* operation. This is the only instruction included in the simulation with only the latest instruction to each aircraft recorded.

$$\begin{array}{|l}
\hline \textit{changeSpeed} \\
\Delta \textit{Traffic} \\
\Xi \textit{Clock} \\
\textit{aircraft}? : \textit{Callsign} \\
\textit{speed}? : \textit{Speed} \\
\hline
\textit{instructions}' = \textit{instructions} \oplus \{\textit{aircraft}? \mapsto (\textit{speed}?, \textit{now})\} \\
\textit{type}' = \textit{type} \\
\textit{telemetry}' = \textit{telemetry} \\
\textit{flightPlan}' = \textit{flightPlan} \\
\hline
\end{array}$$

3.4 Separation violation

The minimum separation between aircraft is defined by a regulatory authority and is modelled as a constant distance (*Distance*).

$$\begin{array}{|l}
\textit{minimumSeparation} : \textit{Distance}
\end{array}$$

The separation between aircraft is calculated using the telemetry data. Functions are provided to extract information from the telemetry data.

$$\begin{array}{|l}
\textit{position} : \textit{Position} \times \textit{Speed} \times \textit{Time} \rightarrow \textit{Position} \\
\textit{time} : \textit{Position} \times \textit{Speed} \times \textit{Time} \rightarrow \textit{Time} \\
\hline
\forall\, p : \textit{Position};\ s : \textit{Speed};\ t : \textit{Time} \bullet \\
\qquad \textit{position}(p, s, t) = p \land \textit{speed}(p, s, t) = s \land \textit{time}(p, s, t) = t
\end{array}$$

The distance between two positions is calculated using the function *distanceBetween*.

$$| \; distanceBetween : Position \times Position \to Distance$$

The *detectSeparationViolation* function identifies (and outputs) the callsign of all aircraft that are currently within the minimum separation distance of another aircraft, and hence in violation of the separation regulations.

```
┌─ detectSeparationViolation ──────────────────────────────
│ ΞClock
│ ΞTraffic
│ violations! : ℙ Callsign
├──────────────
│ violations! = {ac1, ac2 : aircraft | ac1 ≠ ac2 ∧
│      distanceBetween(position(telemetry(ac1)),
│           position(telemetry(ac2))) ≤ minimumSeparation • ac1}
└──────────────────────────────────────────────────────────
```

3.5 Aircraft specifications

Aircraft of the same aircraft type have some common characteristics. These 'specifications' detail the minimum and maximum speeds of the aircraft.

```
┌─ AircraftSpecification ──────────────────────────────────
│ acTypes : ℙ AircraftType
│ minimumSpeed, maximumSpeed : AircraftType ⇸ Speed
├──────────────
│ dom minimumSpeed = dom maximumSpeed = acTypes
│ ∀ acType : acTypes •
│      minimumSpeed(acType) < maximumSpeed(acType)
└──────────────────────────────────────────────────────────
```

3.6 The ATC core system

The ATC system consists of the sector data, the air traffic data, the aircraft specification data, and the clock with the current time.

```
┌─ ATCCore ────────────────────────────────────────────────
│ Sector ∧ Traffic ∧ AircraftSpecification ∧ Clock
├──────────────
│ ran type ⊆ acTypes
└──────────────────────────────────────────────────────────
```

4 The ATC simulator HCI model

The ATC simulator HCI (shown in Figure 1) is modelled using an integrated approach, blending a formal Z model of the interface state with the User-Action notation for describing the user actions. In these models low level details of the HCI are ignored so as not to obscure the high level functionality of the HCI and the actions of the user.

The ATC HCI model includes modelling of the HCI elements relating to the air traffic, operator instructions, and violation warnings. The HCI elements relating to the sector data (the 'background' of the HCI as, for example shown in Figure 1) are not modelled.

A more detailed description of these models (aimed at readers not fluent in Z and UAN) is provided in Appendix B.

4.1 The underlying interface state

The underlying interface state defines the functional model of the ATC interface. The primary part of this is the mapping from the ATC system to the visual representation maintained on the interface devices.

The central aspect of this mapping is of the air traffic in the sector to the views that represent that traffic on the interface. The view of each aircraft is abstractly defined using the given type *AircraftView*. The function *makeView* creates the view from the telemetry and flight path information of an aircraft.

$$\begin{aligned} \mathit{makeView} : &\ \mathit{Callsign} \times (\mathit{Position} \times \mathit{Speed} \times \mathit{Time}) \times \\ &\ \mathrm{seq}_1(\mathit{Waypoint} \times \mathit{Time}) \rightarrow \mathit{AircraftView} \end{aligned}$$

The aircraft specifications appear in the interface in the form of the speed selection menu used when instructing an aircraft to change speed. This menu is abstractly defined using the given type *SpeedMenu*. The list of speeds that may be selected from the menu is derived from the minimum and maximum speeds of the aircraft, and is created using the function *speedList*. The function *makeMenu* creates the menu object from this list of speeds.

$$\begin{aligned} &\mathit{speedList} : \mathit{Speed} \times \mathit{Speed} \nrightarrow \mathrm{seq}\, \mathit{Speed} \\ &\mathit{makeMenu} : \mathrm{seq}\, \mathit{Speed} \rightarrow \mathit{SpeedMenu} \end{aligned}$$

Whenever there is a separation violation in the ATC system, an audible alarm sounds in the interface. The status of this alarm is modelled using a free type.

$$\mathit{AlarmStatus} ::= \mathit{on} \mid \mathit{off}$$

The ATC interface consists of the aircraft views (*views*; the *shows* function maps each view to the callsign of the aircraft it represents, providing the coupling from the HCI to the core system), the *selected* aircraft view (modelled as a set allowing either no aircraft or a single aircraft to be selected), the *speedMenu* used by the operator to instruct the selected aircraft to change speed, the set of aircraft that are currently in a separation violation (*warnings*), and the status of the *alarm*.

Two basic actions on the HCI correspond to clicking the left mouse button (on an aircraft view) and right mouse button. These are selecting an aircraft (*selectAircraft*) and opening the speed menu (*openSpeedMenu*) respectively.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ATCInterface}} \\
\hline
views : \mathbb{P}\, AircraftView \\
shows : AircraftView \rightarrowtail\!\!\!\!\!\!\!\leftrightarrow Callsign \\
selected : \mathbb{P}\, AircraftView \\
speedMenu : SpeedMenu \\
warnings : \mathbb{P}\, AircraftView \\
alarm : AlarmStatus \\
\hline
\mathrm{dom}\, shows = views \\
selected \subseteq views \\
\# selected \leq 1 \\
warnings \subseteq views \\
alarm = if\ warnings = \varnothing \\
\qquad\quad then\ off\ else\ on \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{selectAircraft}} \\
\hline
\Delta ATCInterface \\
aircraft? : AircraftView \\
\hline
selected' = \{aircraft?\} \\
views' = views \\
speedMenu' = speedMenu \\
warnings' = warnings \\
alarm' = alarm \\
shows' = shows \\
\hline
\end{array}$$

The *speedMenu* in the *ATCInterface* is only displayed when requested by the operator. Consequently a lazy approach to maintaining the consistency between *speedMenu* and *selected* is used: the menu is updated to correspond to the currently selected aircraft view when the speed menu is opened. Displaying of the menu is described using UAN in Section 4.4.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{openSpeedMenu}} \\
\hline
\Delta ATCInterface \\
\Xi Traffic \\
\Xi AircraftSpecification \\
\hline
selected \neq \varnothing \\
\exists_1\, acView : selected \bullet speedMenu' = \\
\qquad makeMenu(speedList(minimumSpeed(type(shows(acView))), \\
\qquad\qquad maximumSpeed(type(shows(acView))))) \\
selected' = selected \land views' = views \\
warnings' = warnings \land alarm' = alarm \\
shows' = shows \\
\hline
\end{array}$$

Once the speed menu is opened, the operator may select one of the speeds in the menu. The following operation *selectSpeed1* occurs when this happens. The index of the selected menu item is used to look up the corresponding speed.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{selectSpeed1}} \\
\hline
\Xi ATCInterface \\
\Xi Traffic \\
\Xi AircraftSpecification \\
menuIndex? : \mathbb{Z} \\
aircraft! : Callsign \\
speed! : Speed \\
\hline
selected \neq \varnothing \\
\exists_1\, acView : selected \bullet aircraft! = shows(acView) \\
speed! = speedList(minimumSpeed(type(aircraft!)), \\
\qquad maximumSpeed(type(aircraft!)))(menuIndex?) \\
\hline
\end{array}$$

4.2 Attaching the HCI to the functional core

The operation *selectSpeed1* represents the interface portion of the speed selection operation. Associated with this, the selected speed must be recorded as the instructed speed of the aircraft in the ATC core system. The whole *selectSpeed* operation consists of the operation *selectSpeed1* being piped to the operation *changeSpeed* (from Section 3.3). That is, the speed output from *selectSpeed1* is input into *changeSpeed*.

$$selectSpeed \mathrel{\widehat=} selectSpeed1 \gg changeSpeed$$

Some changes to the ATC system occur independently of the operator, such as the *updateTelemetry* operation defined in the ATC system model. These changes in the ATC system must also be reflected in the HCI by updating the underlying interface state. The *refreshScreen* operation refreshes the interface state (using a brute force approach) – the aircraft views are re-created from the air traffic (and the selected view is updated accordingly), the warnings are updated (via an input to the operation), and the alarm status is set accordingly.

```
── refreshHCI ──────────────────────────────
│ ΔATCInterface
│ ΞTraffic
│ violations? : ℙ Callsign
├──────────
│ shows' = {ac : aircraft •
│        makeView(ac, telemetry(ac), flightPlan(ac)) ↦ ac}
│ views' = dom shows'
│ selected' = (shows')~(| shows(| selected |) |)
│ speedMenu' = speedMenu
│ warnings' = views~(| violations? |)
│ alarm' = if violations? = ∅ then off else on
└────────────────────────────────────────────
```

When the interface is refreshed using *refreshHCI*, the warnings input into the operation need to be calculated. This calculation is provided by the operation *detectSeparationViolation* defined in the ATC core system. A complete update of the HCI thus involves piping the *detectSeparationViolation* operation to *refreshHCI*.

$$updateHCI \mathrel{\widehat=} detectSeparationViolation \gg refreshHCI$$

The entire ATC system includes both the *ATCInterface* defined above and the *ATCCore* system defined in Section 3.

```
── ATC ─────────────────────────────────────
│ ATCInterface
│ ATCCore
├──────────
│ ran shows = aircraft
└────────────────────────────────────────────
```

4.3 The User Action Notation

The remainder of the HCI model integrates the Z notation with the User Action Notation (UAN) [1].

The symbols specific to UAN used in the HCI model are summarised in Figure 3 (taken from [1]). Some additional feedback symbols for the various forms of highlighting the aircraft views in the ATC HCI are defined in Figure 4. The two forms of highlight provided by these symbols are not exclusive: they can be applied simultaneously.

Action	Meaning
$\sim$[X]	Move the cursor into the context of object X
$\sim$[x,y]	Move the cursor to point x,y outside any object
$\sim$[X in Y]	Move the cursor to object X within object Y
[X]$\sim$	Move the cursor out of context of object X
$M_L\vee\wedge$	Click (depress & release) the left mouse button
$M_R\vee\wedge$	Click the right mouse button
	task is performed zero or more times
⨟	task interrupt symbol - indicates the user may interrupt current task at this point
:	separator between condition and action or feedback
Feedback	**Meaning**
!	highlight object
$-$!	dehighlight object
@x,y	at point x,y
display(X)	display object X
erase(X)	erase object X

Figure 3: User Action Notation symbols used in the ATC HCI model

Feedback	Meaning
$!_S$	highlight object using the selection highlight (a circle is drawn around the aircraft dot in the aircraft view)
$!_W$	highlight object using the warning highlight (a different colour is used for the aircraft view)
$-!_S$	turn off selection highlight on object
$-!_W$	turn off warning highlight on object

Figure 4: Additional feedback symbols for highlighting aircraft views

4.4 Displaying the system state

The underlying interface state defines those interface objects that are used to visualise the core system state, but does not define how those objects are composed on the HCI. Here we describe (using a blending of Z with UAN) this aspect of the HCI associated with the *refreshHCI* operation. Identical names to those in the Z model indicate a coupling between these definitions and *refreshHCI*. In particular note that unprimed names (e.g. *selected*) refer to old interface objects before the update, and primed names (e.g. *selected'*) refer to the new interface objects after the update.

Visualisation of the ATC system state consists of displaying the aircraft views at the appropriate positions on the screen. In UAN, screen positions are described as points, so the *convertPosition* function is provided to convert between sector positions and screen positions (we assume that screen positions are represented as a pair of integers).

$$\mathit{convertPosition} : \mathit{Position} \to \mathbb{Z} \times \mathbb{Z}$$

We describe the effects of *refreshHCI* using the feedback symbols of UAN.

Erase the old aircraft views Because *refreshHCI* defines a brute force update (each aircraft view on the HCI is replaced with a new view), all of the old aircraft views must be erased.

$$\forall\, \mathit{oldView} : \mathit{views} \bullet \mathsf{erase}(\mathit{oldView})$$

Display the new aircraft views These are replaced by the new aircraft views in the appropriate positions.

$$\begin{array}{l}\forall\, \mathit{newView} : \mathit{views}' \bullet \\ \quad @\mathit{convertPosition}(\mathit{position}(\mathit{telemetry}(\mathit{shows}'(\mathit{newView})))) \\ \qquad \mathsf{display}(\mathit{newView})\end{array}$$

Apply the selection highlight The selection highlight is applied to the new aircraft views.

$$\forall\, \mathit{acView} : \mathit{selected}' \bullet \mathit{acView}!_{\mathsf{S}}$$
$$\forall\, \mathit{acView} : \mathit{views}' \setminus \mathit{selected}' \bullet \mathit{acView}-!_{\mathsf{S}}$$

Apply the warning highlight The warning highlight is applied to the new aircraft views.

$$\forall\, \mathit{acView} : \mathit{warnings}' \bullet \mathit{acView}!_{\mathsf{W}}$$
$$\forall\, \mathit{acView} : \mathit{views}' \setminus \mathit{warnings}' \bullet \mathit{acView}-!_{\mathsf{W}}$$

4.5 The user actions

In the following definitions of the user actions, the coupling between the user actions and the underlying interface state is implicit in the usage of common attribute and operation names. For example, the first user task defined below, *selectAircraft*, defines the behaviour of the user and the interface that accompanies the *selectAircraft* operation defined in the underlying interface state.

Task: *selectAircraft*

The operator selects an aircraft by moving the mouse over the appropriate aircraft view and clicking the left mouse button:

User Action	Interface Feedback	Operation input
~[aircraft_view]		
$\mathrm{M}_L\vee\wedge$	$\forall\, \mathit{acView} : \mathit{selected} \bullet$ $\mathit{acView}-!_{\mathsf{S}}$ aircraft_view $!_{\mathsf{S}}$	$\mathit{aircraft}? =$ aircraft_view

Task: *changeAircraftSpeed*

The operator instructs the selected aircraft to change speed by opening the speed menu, navigating the menu to the desired speed, then selecting it:

$\mathit{openSpeedMenu} \fatsemi \mathit{navigateSpeedMenu} \fatsemi \mathit{selectSpeed}$

Note that the '$\fatsemi$' symbol used above is the task interrupt symbol. If the user interrupts *changeAircraftSpeed* the effect is: $\mathsf{erase}(\mathit{speedMenu})$

Subtask: *openSpeedMenu*

If an aircraft view is selected, the operator can open the speed menu by clicking the right mouse button:

User Action	Interface Feedback	Interface State
$\mathit{selected} \neq \varnothing$:		
($\sim$[x,y] $\mathrm{M}_R\vee\wedge$)	@ x,y $\mathsf{display}(\mathit{speedMenu}')$	$\mathit{speedMenu}' = \mathit{makeMenu}(\ldots)$

Subtask: *navigateSpeedMenu*

The operator navigates within the speed menu by moving the mouse in and out of the lines in the menu:

User Action	Interface Feedback
$\sim$[line **m** in *speedMenu*]	line **m** !
($\fatsemi$ [line **m** in *speedMenu*]$\sim$ $\fatsemi$	line **m** $-$!
$\sim$[line **n** in *speedMenu*])*	line **n** !

If the user interrupts *navigateSpeedMenu* the effect is: $\mathsf{erase}(\mathit{speedMenu})$.

In the above, 'line **m**' refers to the $\mathbf{m}^{th}$ line in the menu.

Subtask: *selectSpeed*

The operator selects a speed from the speed menu when the mouse is over the appropriate speed line by clicking the left mouse button:

User Action	Interface Feedback	Operation input
$\sim$[line **m** in *speedMenu*] :		
$\mathrm{M}_L\vee\wedge$	$\mathsf{erase}(\mathit{speedMenu})$	$\mathit{menuIndex}? = \mathbf{m}$

5 Conclusions

In summary, this technical report outlines the ATC simulator used for the case study within the SafeHCI project. Formal models for both the HCI and core system of the simulator are provided. This simulation will be used to formulate cognitive models of the cognitive processes involved in the ATC task. Through the combination of the ATC simulator HCI and core system models with the cognitive models we aim to predict how errors can occur within the ATC task and how HCI design interventions affect these errors.

Acknowledgements

The ATC simulator was developed by the Key Centre for Human Factors and Applied Cognitive Psychology at the University of Queensland. Andrew Neal and Mike Humphreys helped develop and reviewed the descriptions of the ATC task and the models here. Finally, we gratefully acknowledge the assistance of Simon Connelly in preparation of this technical report.

References

[1] H. R. Hartson. Temporal Aspects of Tasks in the User Action Notation. *Human-Computer Interaction*, 7:1–45, 1992.

[2] D. Leadbetter, A. Hussey, P. Lindsay, A. Neal, and M. Humphreys. Towards Model Based Prediction of Human Error Rates in Interactive Systems. In *Proc Australasian User Interface Conference 2001*, volume 23 of *Australian Computer Science Communications.* IEEE Press, 2001. See also http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?00-33.

[3] D. Leadbetter, P. Lindsay, A. Neal, and M. Humphreys. Integrating the operator into formal models in the air-traffic control domain. Technical Report 00-34, Software Verification Research Centre, November 2000. http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?00-34.

[4] J.M. Spivey. *The Z Notation: a Reference Manual.* Prentice-Hall, New York, second edition, 1992. http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html.

A Details of the ATC System Core Model

This part of the report contains annotated versions of the specifications presented above, to assist readers who are less familiar with Z and UAN to understand the specifications.

A.1 Time

Two forms of time are modelled: absolute time and durations.

The given type *Time* models absolute time. It represents time such as 8:35am.

The given type *Duration* models durations. It represents durations such as 1:26, or 1 hour and 26 minutes.

$$[\mathit{Time}, \mathit{Duration}]$$

The difference (duration) between two absolute times is calculated using the given function called *timeDifference*. The definition of *timeDifference* given below defines the signature of the function only. It does not define how the function result is calculated from the arguments.

$$\mathit{timeDifference} : \mathit{Time} \times \mathit{Time} \rightarrow \mathit{Duration}$$

This function takes two absolute times as arguments and returns the duration between them. For example:

$$\mathit{timeDifference}(\text{8:35am}, \text{2:14pm}) = \text{5:39}$$

The schema *Clock* models a real-time clock. It's single attribute *now* represents the current time.

```
┌─ Clock ──────────────────────────
│ now : Time
└──────────────────────────────────
```

For example, at 8:35am each day $\mathit{Clock.now} = 8 : 35\text{am}$.

A.2 Sector data

Sector information is based on waypoints and positions.

Waypoints and airports are modelled using the given type *Waypoint*.

$$[\mathit{Waypoint}]$$

All waypoints and airports are members of this type. For example, the waypoint 'Nickol Bay' and the airport 'Exmouth' are both members of the *Waypoint* type: i.e.

$$\{\text{Nickol Bay}, \text{Exmouth}\} \subseteq \mathit{Waypoint}$$

The positions of waypoints in a sector are modelled using the given type *Position*.

$[Position]$

All positions are members of this type. For example, the position of Nickol Bay is latitude 20°39′S and longitude 116°52′E, and the position of Exmouth airport is latitude 21°56′S and longitude 114°08′E, so:

$\{(20°39'\mathrm{S}, 116°52'\mathrm{E}), (21°56'\mathrm{S}, 114°08'\mathrm{E})\} \subseteq \mathit{Position}$

Later on in the model the distance between positions will be important for defining the minimum separation between aircraft (and detecting separation violations), so we define a given type to represent distance (called *Distance*):

$[Distance]$

The distance between two positions is calculated using the given function called *distanceBetween*. The definition of *distanceBetween* given below defines the signature of the function only. It does not define how the function result is calculated from the arguments.

$| \quad \mathit{distanceBetween} : \mathit{Position} \times \mathit{Position} \rightarrow \mathit{Distance}$

This function takes two positions as arguments and returns the distance between them.

For example the distance between Nickol Bay and Exmouth airport is:

$\mathit{distanceBetween}((20°39'S, 116°52'E), (21°56'S, 114°08'E)) = 317\mathrm{km}$

The details of a sector are defined by the *Sector* schema that follows. The contents of this schema are described line by line.

```
── Sector ─────────────────────────────────
│ [1] waypoints : ℙ Waypoint
│ [2] position : Waypoint ⇸ Position
│ [3] airports : ℙ Waypoint
│ [4] routes : Waypoint ↔ Waypoint
├──────────────
│ [5] airports ⊆ waypoints
│ [6] dom position = waypoints
│ [7] dom routes ∪ ran routes = waypoints
└───────────────────────────────────────────
```

A sector consists of three things: waypoints, airports, and flight routes.

[1] The waypoints in a sector are modelled as a set of waypoints.
[2] The positions of these waypoints are modelled as a partial function from waypoints to positions.
[3] The airports in a sector are identified using a set of waypoints.
[4] The flight routes in a sector are modelled as a relationship from the set of all waypoints to itself. This relationship effectively consists of a set of pairs of waypoints, each pair of waypoints indicating a route joining the two waypoints. Note that modelling flight routes in this way requires that the entry/exit points of flight routes also occur at waypoints (in reality there would not normally be waypoints at these entry/exit points).
[5] The airports in the sector is a subset of the waypoints in the sector.
[6] All of the waypoints in the sector (and only those waypoints) have a position.
[7] All of the waypoints in the sector (and only those waypoints) are part of some route in the sector.

A.3 Aircraft and air traffic data

Information for an aircraft in the system consists of its call sign, aircraft type, speed, and flight route. Given types are provided to represent call signs, aircraft types, and aircraft speeds (everything needed to model flight routes is already provided). These types are *Callsign*, *AircraftType*, and *Speed* respectively.

$$[Callsign, AircraftType, Speed]$$

For example, for the aircraft QF053 Heavy travelling a 420knots to Borrow Island and then onto Exmouth airport, the following elements of these given types exist:

$$\text{QF053} \in Callsign$$
$$\text{Heavy} \in AircraftType$$
$$\text{420knots} \in Speed$$

Information on all air traffic in the sector is defined by the *Traffic* schema that follows. The contents of this schema are described line by line.

Traffic

[1] $aircraft : \mathbb{P}\, Callsign$
[2] $type : Callsign \pfun AircraftType$
[3] $telemetry : Callsign \pfun Position \times Speed \times Time$
[4] $flightPlan : Callsign \pfun \mathrm{seq}_1(Waypoint \times Time)$
[5] $instructions : Callsign \pfun (Speed \times Time)$

[6] $aircraft = \mathrm{dom}\, type = \mathrm{dom}\, telemetry = \mathrm{dom}\, flightPlan$
[7] $\mathrm{dom}\, instructions \subseteq aircraft$

A sector consists of aircraft for which the following information is recorded: the aircraft type, telemetry details, flight plan, and any current instruction from the ATC operator.

[1] The aircraft present in a sector is modelled using a set consisting of the aircraft callsigns.
[2] The types of those aircraft is modelled using a partial function from callsigns to aircraft types – each pair in this function describes the type of a single aircraft.
[3] Telemetry information consists of the aircraft position and speed at a certain time. The telemetry information for the aircraft in the sector is modelled using a partial function from callsigns to triples consisting of the position, speed, and time. Each pair in the telemetry function describes the telemetry of a single aircraft.
[4] An aircraft flight plan is modelled as a sequence of waypoints with corresponding ETAs (estimated times of arrival). The flight plans of the aircraft in the sector are modelled using a partial function from callsigns to such flight plan sequences. Each pair in this function (consisting of a callsign and flight plan sequence) describes the flight plan of a single aircraft.
[5] An instruction to an aircraft is a command to change speed that occurs at a specific time. An instruction is modelled as a pair consisting of the target speed and the time of issue. Aircraft instructions are modelled as a partial function from callsigns to such instructions. Each pair in this function describes the current instruction of a single aircraft (each aircraft may have at most one instruction at a time).
[6] The aircraft type, telemetry, and flight plan must be recorded for all aircraft in the sector

(but not for any aircraft that are not in the sector).
[7] Aircraft instructions are only recorded for aircraft in the sector, however aircraft do not have to have instructions.

The information about the traffic in the sector is updated via a number of operations.

Aircraft telemetry is updated by the *updateTelemetry* operation defined by the following operation schema. The contents of this schema are described line by line.

updateTelemetry

[1] $\Delta \mathit{Traffic}$
[2] $\mathit{newTelemetry?} : \mathit{Callsign} \pfun \mathit{Position} \times \mathit{Speed} \times \mathit{Time}$

---

[3] $\mathit{telemetry}' = \mathit{telemetry} \oplus \mathit{newTelemetry?}$
[4] $\mathit{type}' = \mathit{type}$
[5] $\mathit{flightPlan}' = \mathit{flightPlan}$
[6] $\mathit{instructions}' = \mathit{instructions}$

[1] The air traffic information in the *Traffic* schema is modified by this operation.
[2] The input to this operation is the new telemetry data. This input is modelled as a function from callsigns to telemetry tuples. Each pair in this functions represents one aircraft and its new telemetry information.
[3] The new telemetry information is stored in the telemetry relation in *Traffic*, **overriding** the existing telemetry information for the aircraft involved.
[4] The aircraft type information is unchanged.
[5] The flight plan information is unchanged.
[6] The aircraft instructions are unchanged.

Aircraft instructions are updated when the ATC operator issues a new instruction to an aircraft. Instructions to change speed (which is the only instruction included in the example) are given using the *changeSpeed* operation defined by the following operation schema. The contents of this schema are described line by line.

changeSpeed

[1] $\Delta \mathit{Traffic}$
[2] $\Xi \mathit{Clock}$
[3] $\mathit{aircraft?} : \mathit{Callsign}$
[4] $\mathit{speed?} : \mathit{Speed}$

---

[5] $\mathit{instructions}' = \mathit{instructions} \oplus \{\mathit{aircraft?} \mapsto (\mathit{speed?}, \mathit{now})\}$
[6] $\mathit{type}' = \mathit{type}$
[7] $\mathit{telemetry}' = \mathit{telemetry}$
[8] $\mathit{flightPlan}' = \mathit{flightPlan}$

[1] The air traffic information in the *Traffic* schema is modified by this operation.
[2] The information in *Clock* is referenced by this operation.
[3] The first input to this operation is the callsign of the aircraft to which the instruction applies.
[4] The second input to this operation is the target speed that the aircraft is being instructed to meet.
[5] The new instruction is stored in the instruction relation in *Traffic*, **overriding** the existing instruction information for the aircraft involved.

[6] The aircraft type information is unchanged.
[7] The telemetry information of the aircraft in the sector is unchanged.
[8] The flight plan information is unchanged.

A.4 Separation violation

The minimum regulatory distance that must be maintained between aircraft is called the minimum separation distance. This distance is modelled as a constant instance of the given type *Distance* and is defined as follows.

$$minimumSeparation : Distance$$

This definition defines the constant value *minimumSeparation*. While the value of *minimumSeparation* is not defined it always refers to the same value everywhere it is referenced, and cannot be changed (ever!).

The telemetry information for an aircraft is modelled as a triple. The two functions *position* and *time* take one such triple as an argument an extract the position and time from it respectively.

$$\begin{array}{|l}
position : Position \times Speed \times Time \rightarrow Position \\
time : Position \times Speed \times Time \rightarrow Time \\
\hline
\forall\, p : Position;\ s : Speed;\ t : Time \bullet \\
\quad position(p, s, t) = p \land speed(p, s, t) = s \land time(p, s, t) = t
\end{array}$$

Violations of the minimum separation distance are detected using the *detectSeparationViolation* operation that is defined by the following operation schema. The contents of this schema are described line by line.

$$\begin{array}{|l}
\underline{\quad detectSeparationViolation \qquad\qquad\qquad\qquad\qquad\qquad} \\
[1]\ \Xi Clock \\
[2]\ \Xi Traffic \\
[3]\ violations! : \mathbb{P}\ Callsign \\
\hline
[4]\ violations! = \{ac1, ac2 : aircraft \mid ac1 \neq ac2 \land \\
\quad distanceBetween(position(telemetry(ac1)), \\
\qquad position(telemetry(ac2))) \leq minimumSeparation \bullet ac1\} \\
\hline
\end{array}$$

[1] The information in *Clock* is referenced by this operation.
[2] The air traffic information in *Traffic* is referenced by this operation.
[3] The output from this operation is the set of aircraft callsigns that are violating the minimum separation distance.
[4] The set of violating aircraft callsigns are all those aircraft where the distance between them and another different aircraft is less than the minimum separation distance.

A.5 Aircraft specifications

When issuing aircraft with instructions, the ATC system must offer the operator appropriate options. For example, when instructing an aircraft to change speed, the speeds offered should all be speeds which the particular aircraft is capable of safely achieving. Such details are described by the aircraft specifications. The aircraft specifications are recorded by the *AircraftSpecification* schema that follows. The contents of this schema are described line by line.

AircraftSpecification

$$\begin{array}{l}
[1]\ acTypes : \mathbb{P}\ AircraftType \\
[2]\ minimumSpeed, maximumSpeed : AircraftType \nrightarrow Speed \\
\hline
[3]\ \mathrm{dom}\ minimumSpeed = \mathrm{dom}\ maximumSpeed = acTypes \\
[4]\ \forall\ acType : acTypes \bullet \\
\qquad minimumSpeed(acType) < maximumSpeed(acType)
\end{array}$$

[1] The aircraft types for which specifications are recorded are modelled as a set of aircraft types.
[2] The minimum speed and maximum speed of aircraft are each modelled by a partial function from aircraft type to speed. Each pair in this function includes an aircraft type and the minimum speed (or maximum speed in the respective function) for that aircraft type.
[3] All recorded aircraft types have both a minimum speed and a maximum speed.
[4] For all recorded aircraft types the minimum speed is less than the maximum speed.

A.6 The ATC core system

The ATC core system consists of the sector information, air traffic information, aircraft specification information, and clock that are modelled above. The ATC core system is modelled by the *ATCCore* schema that follows.

ATCCore

$$\begin{array}{l}
[1] Sector \land Traffic \land AircraftSpecification \land Clock \\
\hline
[2]\ \mathrm{ran}\ type \subseteq acTypes
\end{array}$$

[1] The ATC core system includes the sector information, air traffic information, aircraft specification information, and clock (as were previously defined) .
[2] All of the aircraft types recorded for the aircraft in the air traffic information are included in the aircraft specification information.

B Details of the ATC simulator HCI model

The ATC HCI presents a view of the ATC core system and provides interface operations for interacting with the ATC core system. The HCI model defines the dynamic aspects of the HCI only – these are the display of the aircraft in the sector and the HCI operations; the static aspects of the HCI are omitted from the HCI model – these are the display of the waypoints, airports, routes, etc.

B.1 The underlying interface state

The ATC HCI consists essentially of two different things: the views of the aircraft in the sector, and the menus used for giving instructions to aircraft.

The views of the aircraft in the sector are modelled using the given type *AircraftView*. Each aircraft view represents the graphic of the aircraft and its associated information (callsign, type, speed, and route) that is shown on the screen.

$[\mathit{AircraftView}]$

An aircraft view for a specific aircraft and telemetry is created using the given function *make-View* (technically views are not created, rather an appropriate view is picked). The definition of *makeView* given below defines the signature of the function only. The function takes three arguments (an aircraft callsign, telemetry information, and flight plan), and returns a corresponding aircraft view. The definition does not define how the function result is calculated from the arguments.

$$\begin{array}{l} \mathit{makeView} : \mathit{Callsign} \times (\mathit{Position} \times \mathit{Speed} \times \mathit{Time}) \times \\ \qquad\qquad \mathrm{seq}_1(\mathit{Waypoint} \times \mathit{Time}) \rightarrow \mathit{AircraftView} \end{array}$$

A single type of instruction can be given to aircraft: to change speed. The menu used to select the target speed of the aircraft is modelled by the given type *SpeedMenu*.

$[\mathit{SpeedMenu}]$

Each speed menu presents a list of speeds based on the minimum and maximum speeds of an aircraft (recorded in the aircraft specification of that aircraft type). The given function *speedList* is used to generate the list of speeds that appear in a speed menu given the minimum and maximum speed. This list is used to generate the speed menu by the given function *makeMenu*. The definitions of both of *speedList* and *makeMenu* given below define the signature of the function only. The definitions do not define how the result of each function is calculated from its arguments.

$$\begin{array}{l} \mathit{speedList} : \mathit{Speed} \times \mathit{Speed} \nrightarrow \mathrm{seq}\, \mathit{Speed} \\ \mathit{makeMenu} : \mathrm{seq}\, \mathit{Speed} \rightarrow \mathit{SpeedMenu} \end{array}$$

When a line is selected in a speed menu, the speed represented by that line must be looked up. The lookup capability is provided by the given function *lookupSpeed*. The definition of *lookupSpeed* given below defines the signature of the function only. The function takes a speed menu and the selected line as arguments and returns the appropriate speed. The definition does not define how the function result is calculated from the arguments.

$$\mathit{lookupSpeed} : \mathit{SpeedMenu} \times \mathbb{Z} \nrightarrow \mathit{Speed}$$

The ATC HCI includes an audible alarm that sounds continuously whenever a separation violation is in effect. The status of such an alarm (whether it is on or off) is modelled by the free type *AlarmStatus* defined below. There are two distinct status values for an alarm: *on* and *off*.

$AlarmStatus ::= on \mid off$

The details of the ATC HCI are defined by the *ATCInterface* schema that follows. The contents of this schema are described line by line.

$$
\begin{array}{|l}
\hline
\textit{ATCInterface} \\
\hline
[1]\ views : \mathbb{P}\, AircraftView \\
[2]\ shows : AircraftView \rightarrowtail\!\!\!\!\!\!\;\;\to Callsign \\
[3]\ selected : \mathbb{P}\, AircraftView \\
[4]\ speedMenu : SpeedMenu \\
[5]\ warnings : \mathbb{P}\, AircraftView \\
[6]\ alarm : AlarmStatus \\
\hline
[7]\ \mathrm{dom}\, shows = views \\
[8]\ selected \subseteq views \\
[9]\ \#selected \leq 1 \\
[10]\ warnings \subseteq views \\
[11]\ alarm = if\ warnings = \varnothing \\
\qquad\qquad then\ off\ else\ on \\
\hline
\end{array}
$$

The ATC HCI consists of aircraft views, up to one of which may be selected at any one time, a single speed menu. When separation violations occur warnings are indicated on the HCI for each involved aircraft, and the alarm is turned on.

[1] The aircraft in the sector are presented on the HCI by a set of aircraft views.
[2] The association between the views and the aircraft (what views show which aircraft) is modelled using a function from aircraft views to callsigns – each pair in the function describes the aircraft view that represents the aircraft with the associated callsign.
[3] The selected aircraft is modelled using a set of aircraft views. When there is no selection this set will be empty; when there is a selected aircraft this set will contain that single aircraft view.
[4] The speed menu is simply an instance of the speed menu type. Note that the value of the speed menu is only important while the menu is being displayed on the HCI; while it is not being displayed its value is unimportant.
[5] The current warnings on the HCI is modelled as a set of aircraft views to be displayed using a particular highlight. If there are no warnings (because there are no separation violations) this set will be empty.
[6] The warning alarm is simply modelled using the alarm status.
[7] All of the aircraft views on the HCI must correspond to an aircraft in the core system (recorded using the aircraft callsign).
[8] The selected aircraft view (if any) is one of the views on the HCI.
[9] There can be at most one selected aircraft view, but may be no selected aircraft views.
[10] All of the aircraft views in the current warnings must be aircraft views currently on the HCI.
[11] The alarm status is off if there are no warnings, and it is on if there are any warnings.

An aircraft (aircraft view) is selected on the HCI using the *selectAircraft* operation modelled by the following operation schema. The contents of this schema are described line by line.

See Appendix B.4 for details of the user actions involved in this action.

$$
\begin{array}{l}
\underline{\textit{selectAircraft}} \\
[1]\ \Delta \textit{ATCInterface} \\
[2]\ \textit{aircraft?} : \textit{AircraftView} \\
\hline
[3]\ \textit{selected}' = \{\textit{aircraft?}\} \\
[4]\ \textit{views}' = \textit{views} \\
[5]\ \textit{speedMenu}' = \textit{speedMenu} \\
[6]\ \textit{warnings}' = \textit{warnings} \\
[7]\ \textit{alarm}' = \textit{alarm} \\
[8]\ \textit{shows}' = \textit{shows}
\end{array}
$$

[1] The ATC interface information is modified by this operation.
[2] The input to this operation is the aircraft view that is to be selected.
[3] The aircraft selection is updated to be the singleton set containing the input aircraft view.
[4] The views on the HCI are unchanged.
[5] The speed menu is unchanged.
[6] The current warnings are unchanged.
[7] The alarm status is unchanged.
[8] The association between views and aircraft is unchanged.

The process of instructing an aircraft to change speed is begun by opening the speed menu. The speed menu is opened using the *openSpeedMenu* operation defined by the following operation schema. The contents of this schema are described line by line.

See Appendix B.4 for details of the user actions involved in this action.

$$
\begin{array}{l}
\underline{\textit{openSpeedMenu}} \\
[1]\ \Delta \textit{ATCInterface} \\
[2]\ \Xi \textit{Traffic} \\
[3]\ \Xi \textit{AircraftSpecification} \\
\hline
[4]\ \textit{selected} \neq \varnothing \\
[5]\ \exists_1\ \textit{acView} : \textit{selected} \bullet \textit{speedMenu}' = \\
\qquad \textit{makeMenu}(\textit{speedList}(\textit{minimumSpeed}(\textit{type}(\textit{views}(\textit{acView}))), \\
\qquad\qquad \textit{maximumSpeed}(\textit{type}(\textit{views}(\textit{acView}))))) \\
[6]\ \textit{selected}' = \textit{selected} \land \textit{views}' = \textit{views} \\
[7]\ \textit{warnings}' = \textit{warnings} \land \textit{alarm}' = \textit{alarm} \\
[8]\ \textit{shows}' = \textit{shows}
\end{array}
$$

[1] The ATC interface information is modified by this operation.
[2] The air traffic information is referenced by this operation (this information is from the ATC core system).
[3] The aircraft specification information is referenced by this operation (this information is from the ATC core system).
[4] **Pre-condition:** there must be a selected aircraft view.
[5] The speed menu is updated to correspond to the aircraft type of the selected aircraft.
[6] The aircraft selection is unchanged, and the views on the HCI are unchanged.
[7] The current warnings are unchanged, and the alarm status is unchanged.
[8] The association between views and aircraft is unchanged.

While the speed menu is open, the operator selects the target speed for the aircraft using the operation *selectSpeed1* defined by the following operation schema. The contents of this schema are described line by line.

See Appendix B.4 for details of the user actions involved in this action.

```
selectSpeed1
[1] $\Xi ATCInterface$
[2] $\Xi Traffic$
[3] $\Xi AircraftSpecification$
[4] $menuIndex? : \mathbb{Z}$
[5] $aircraft! : Callsign$
[6] $speed! : Speed$
────────────
[7] $selected \neq \varnothing$
[8] $\exists_1 \, acView : selected \bullet aircraft! = shows(acView)$
[9] $speed! = speedList(minimumSpeed(type(aircraft!)),$
      $maximumSpeed(type(aircraft!)))(menuIndex?)$
```

[1] The ATC interface information is referenced by this operation.
[2] The air traffic information is referenced by this operation (this information is from the ATC core system).
[3] The aircraft specification information is referenced by this operation (this information is from the ATC core system).
[4] The input to this operation is the line in the menu that is selected (specifically, the index of the line or line number).
[5] The first output from this operation is the callsign of the aircraft represented by the selected view.
[6] The second output from this operation is the speed that was selected from the speed menu.
[7] **Pre-condition:** there must be a selected aircraft view.
[8] The output aircraft callsign is the callsign of the selected aircraft view.
[9] The output speed is the speed at the chosen line of the speed menu.

B.2 Attaching the HCI to the functional core

A number of the above operation schemes only describe the HCI part of an operation in the ATC system. Specifically, selecting a speed from a speed menu in the HCI results in a corresponding change being recorded in the appropriate aircraft's instructions as described by the *changeSpeed* operation. The complete operation is the composition of the HCI aspects and the core system aspects of the operation. The complete operation is the *selectSpeed* operation defined as follows.

$$selectSpeed \mathrel{\widehat=} selectSpeed1 \gg changeSpeed$$

In this composition the outputs from *selectSpeed1* are used as the inputs to *changeSpeed*, so that the appropriate aircraft has its instructions updated with the appropriate target speed.

We have seen in the operation *updateTelemetry* that the telemetry information for air traffic can be updated. The HCI needs to be refreshed so as to display an accurate air picture of the traffic in the sector. The *refreshHCI* operation defined by the following operation schema provides the mechanism for updating the views, warnings, etc shown on the HCI to maintain accuracy with respect to the telemetry information. The contents of this schema are described line by line.

```
─ refreshHCI ──────────────────────────────────────────
│ [1] $\Delta ATCInterface$
│ [2] $\Xi Traffic$
│ [3] $violations? : \mathbb{P}\ Callsign$
├──────────────
│ [4] $shows' = \{ac : aircraft \bullet$
│       $makeView(ac, telemetry(ac), flightPlan(ac)) \mapsto ac\}$
│ [5] $views' = \mathrm{dom}\ shows'$
│ [6] $selected' = (shows')^{\sim}(\!|\ shows(\!|\ selected\ |\!)\ |\!)$
│ [7] $speedMenu' = speedMenu$
│ [8] $warnings' = views^{\sim}(\!|\ violations?\ |\!)$
│ [9] $alarm' = \mathit{if}\ violations? = \varnothing\ \mathit{then}\ \mathit{off}\ \mathit{else}\ \mathit{on}$
└──────────────────────────────────────────────────────
```

[1] The ATC interface information is updated by this operation.
[2] The air traffic information is referenced by this operation (this information is from the ATC core system).
[3] The input to this operation is the set of callsigns of all aircraft that are currently in violation of the minimum separation requirements.
[4] New aircraft views are associated with each of the aircraft in the sector.
[5] The new views in the HCI are those newly associated with the aircraft.
[6] The selected aircraft view has the same callsign as the previously selected aircraft view.
[7] The speed menu is unchanged.
[8] The current warnings are those aircraft views that relate to any of the aircraft callsigns that are currently in violation of the minimum separation requirements.
[9] The alarm status is off if there are no violations, and it is on if there are any violations.

The *refreshHCI* operation requires the set of callsigns that are currently in violation of the minimum separation requirements to be provided as input. This set of callsigns is output by the *detectSeparationViolation* operation from the ATC core system. Consequently, when the air traffic telemetry information is updated, the HCI is update by composing the *detect-SeparationViolation* operation with the *refreshHCI* operation in order to calculate the current violations and refresh the HCI. The *updateHCI* composes these two operations as required, and is defined as follows.

$$updateHCI \mathrel{\widehat=} detectSeparationViolation \gg refreshHCI$$

The complete ATC system consists of both the ATC core system and the ATC HCI. This is modelled by the *ATC* schema that is defined as follows.

```
─ ATC ─────────────────────────────────────────────────
│ [1] $ATCInterface$
│ $ATCCore$
├──────────────
│ [2] $\mathrm{ran}\ shows = aircraft$
└──────────────────────────────────────────────────────
```

[1] The ATC system includes the ATC HCI and the ATC core system.
[2] The aircraft presented by the views on the HCI are those aircraft in the core system.

B.3 Displaying the system state

Displaying the aircraft views in the correct positions on the HCI requires a function for converting an aircrafts position (as recorded in its current telemetry) to coordinates on the HCI. The given function *convertPosition* performs this conversion. The signature of this function is defined as follows.

$$| \; convertPosition : Position \rightarrow \mathbb{Z} \times \mathbb{Z}$$

When the HCI is refreshed (as defined by *refreshHCI*) the following effects result:
[1] The old aircraft views are erased.

[1] $\forall\, oldView : views \bullet \mathsf{erase}(oldView)$

[2] The new aircraft views are displayed in the appropriate positions in the HCI.

[2] $\forall\, newView : views' \bullet$
$\quad @convertPosition(position(telemetry(shows'(newView))))$
$\qquad \mathsf{display}(newView)$

The selection highlight is applied to the appropriate view such that:
[3] the selected view (if any) is highlighted, and [4] all non-selected views are not highlighted.

[3] $\forall\, acView : selected' \bullet acView!_{\mathsf{S}}$
[4] $\forall\, acView : views' \setminus selected' \bullet acView{-}!_{\mathsf{S}}$

The warning highlight is applied to the appropriate views such that:
[5] all warning views (if any) are highlighted, and
[6] all non-warning views are not highlighted.

[5] $\forall\, acView : warnings' \bullet acView!_{\mathsf{W}}$
[6] $\forall\, acView : views' \setminus warnings' \bullet acView{-}!_{\mathsf{W}}$

B.4 The user actions

The interface operations defined above (*selectAircraft*, *openSpeedMenu*, *selectSpeed*) are invoked in connection with user actions on the HCI (involving mouse movement, button presses, etc). The user actions and the related results of them for these operations are defined using the UAN. The user task associated with invoking each operation is considered in turn.

Task: *selectAircraft*

The *selectAircraft* task is used to select an aircraft on the HCI and results in the *selectAircraft* operation being invoked.

The following sequence of user actions is involved:
[1] The operator moves the mouse over the appropriate aircraft view and then
[2] The operator clicks the left mouse button to select it.

The outcome of these actions is:
[3] The aircraft view that was clicked on becomes selected (using the selection highlight) and all other aircraft views are not selected.

[4] The input to the *selectAircraft* operation is the aircraft that was clicked on.

User Action	Interface Feedback	Operation input
[1] ∼[aircraft_view]		
[2] $M_L \vee \wedge$	[3] $\forall$ *acView* : *selected* • *acView*−$!_S$ aircraft_view $!_S$	[4] *aircraft*? = aircraft_view

Task: *changeAircraftSpeed*

The *changeAircraftSpeed* task is used to instruct and aircraft to change speed.

[1] The following sequence of user actions is involved:
Open the speed menu;
Navigate the menu to the desired speed;
Select the desired speed.

The outcome of these actions is:
The aircraft is instructed to change speed as described in the operation *changeAircraftSpeed.*

[1] $openSpeedMenu \fatsemi navigateSpeedMenu \fatsemi selectSpeed$

If, however, the above sequence of actions is interrupted, the menu is erased: erase(*speedMenu*)

Subtask: *openSpeedMenu*

The *openSpeedMenu* task is used to open the speed menu.

The following sequence of user actions is involved:
[1] If there is a selected aircraft view then the speed menu is opened by
[2] Moving the mouse to an arbitrary position on the HCI and clicking the right mouse button.

The outcome of these actions is:
[3] The speed menu is displayed at the mouse position.

[4] The speed menu displayed is the menu corresponding to the selected aircraft view, as defined in the *openSpeedMenu* operation.

User Action	Interface Feedback	Interface State
[1] $selected \neq \varnothing$:		
[2] (∼[x,y] $M_R \vee \wedge$)	[3] @ x,y display(*speedMenu*′)	[4] $speedMenu' = makeMenu(\ldots)$

Subtask: *navigateSpeedMenu*

The operator navigates within the speed menu by moving the mouse in and out of the lines in the menu:

The *navigateSpeedMenu* task is used to navigate within the speed menu. This is done with the mouse.

The following sequence of user actions is involved:
[1] Move the mouse into the **m**-th line of the menu (the outcome of this being that the line is

highlighted [2]);
An arbitrary number of times:
[3] move the mouse out of the **m**-th line of the menu (the outcome of this being that the line is un-highlighted [4]), and
[5] into the **n**-th line of the menu (the outcome of this being that the line is highlighted [6]).

User Action	Interface Feedback
[1] $\sim$[line **m** in *speedMenu*]	[2] line **m** !
[3] ($\fatsemi$ [line **m** in *speedMenu*]$\sim$ $\fatsemi$	[4] line **m** $-$!
[5] $\sim$[line **n** in *speedMenu*])*	[6] line **n** !

If, however, the above sequence of actions is interrupted, the menu is erased: erase(*speedMenu*)

Subtask: *selectSpeed*

The operator selects a speed from the speed menu when the mouse is over the appropriate speed line by clicking the left mouse button:

The *selectSpeed* task is used to select a speed from the speed menu.

The following sequence of user actions is involved:
[1] If the mouse is in the **m**-th line of the speed menu, then
[2] Click the left mouse button.

The outcome of these actions is:
[3] The speed menu is erased from the HCI.

[4] The input to the *selectSpeed* operation is the line number of the menu that was clicked on.

User Action	Interface Feedback	Operation input
[1] $\sim$[line **m** in *speedMenu*] :		
[2] $M_L \vee \wedge$	[3] erase(*speedMenu*)	[4] $menuIndex? = \mathbf{m}$