Computer Science at Kent

Interpreting ODP Viewpoint Specification: Observations from a Case Study

Chris Taylor, Eerke Boiten and John Derrick

Technical Report No: 9-01
Date: September 2001

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK.

Interpreting ODP Viewpoint Specification: Observations from a Case Study

Chris Taylor, Eerke Boiten and John Derrick
Computing Laboratory, University of Kent,
Canterbury, CT2 7NF, UK
C.N.Taylor-1,E.A.Boiten,J.Derrick@ukc.ac.uk

Abstract

Open Distributed Processing (ODP) is a framework for specifying open distributed systems, under development by the International Standards Organisation (ISO). It is based on the general idea of "viewpoints" — i.e. partial specifications of an overall system, from different perspectives — but assumes five specific, named viewpoints, which are described informally in the ODP Reference Model (RM-ODP). This paper summarises some observations regarding an attempt to use ODP to specify a complex air traffic control system. Some of the key issues that arise are discussed further in the context of the formal specification of a simpler, idealised model, involving two formalised viewpoints — the Information Viewpoint (a high-level, abstract specification, in Z), and the Computational Viewpoint (a specification of distributed components and objects, in Object-Z).

1 Introduction

The ODP (Open Distributed Processing) framework is a general architecture for open distributed systems, proposed by the ISO (International Standards Organisation). The Reference Model for ODP (RM-ODP — see [10]) identifies five "viewpoints" for system specification, each providing a different, partial perspective on the overall system:

- *Enterprise Viewpoint* — focuses on the overall scope, purpose, and policies of the system. Represents the system as a "community" of "actors", serving an overall objective.
- *Information Viewpoint* — specifies in a fairly abstract way the information involved in the system, and how it is processed, without describing the distributed architecture that will be used.
- *Computational Viewpoint* — a functional decomposition of the system into objects that interact via specific interfaces.
- *Engineering Viewpoint* — a specification of the mechanisms and functions needed to support interaction between the distributed objects of the system.
- *Technology Viewpoint* — concerned with the concrete technological infrastructure of a system, in terms of the particular hardware and software components involved, and how they are interconnected and inter-related.

Since it is intended to provide an architecture for open systems, the RM-ODP does not prescribe particular specification formalisms, software, or hardware. The viewpoints are informally defined in natural language, and so are inevitably somewhat open to differing interpretations, although there have been attempts to provide formal or semi-formal models of some aspects of the viewpoints [2, 3, 14]. (For a discussion of the idea of viewpoints in general, see [9].)

There have been some attempts at large-scale applications of ODP, of which the European air traffic control organisation Eurocontrol's ECHO study [8] — a specification of a particular air traffic control system

— is an interesting example. Consideration of the ECHO study is a useful exercise, in that it raises some general issues regarding ODP and the interpretation of its Reference Model, which we discuss in an informal setting. These include, for example: (1) what the scope and nature of the Information Viewpoint should be (in the ECHO study, it defines datatypes subsequently used as class attribute types in the Computational Viewpoint, but does not encompass definitions of system state or system operations); (2) how a hierarchical division into subsystems can be integrated with a division according to viewpoints (the ECHO study uses the Enterprise Viewpoint to express some hierarchical structuring, equating subsystems with multiple "subcommunities"); (3) how structure in terms of *instances* of distributed objects should be specified in the Computational Viewpoint (in the ECHO study, Computational classes are defined, along with some constraints involving methods and cardinality links between classes, but structure in terms of instances is less explicit); and (4) what kinds of relationships should there be between the viewpoints (for example, the ECHO study uses Enterprise actor types as a proper subset of the Computational class types).

The remainder of this paper is organised as follows. Section 2 describes how ODP is used and interpreted in the ECHO study. In the light of some of the issues raised in section 2, section 3 presents a much simplified, idealised model of an air traffic control system, in which the formal specification languages Z [12] and Object-Z [11] are used to represent the Information and Computational Viewpoints respectively. The Enterprise Viewpoint is expressed informally, and used to constrain the structure of the Computational Viewpoint. Section 4 discusses the roles of the viewpoint correspondences, both in the ECHO study, and in our simplified formal model. Finally, section 5 concludes the paper with a summary of the main points.

2 ODP in the ECHO Study

The ECHO study provides specifications from three viewpoints — the Enterprise, Computational, and Information Viewpoints. This section looks at how these three are used to structure the overall specification, and comments on them individually.

2.1 Enterprise Viewpoint

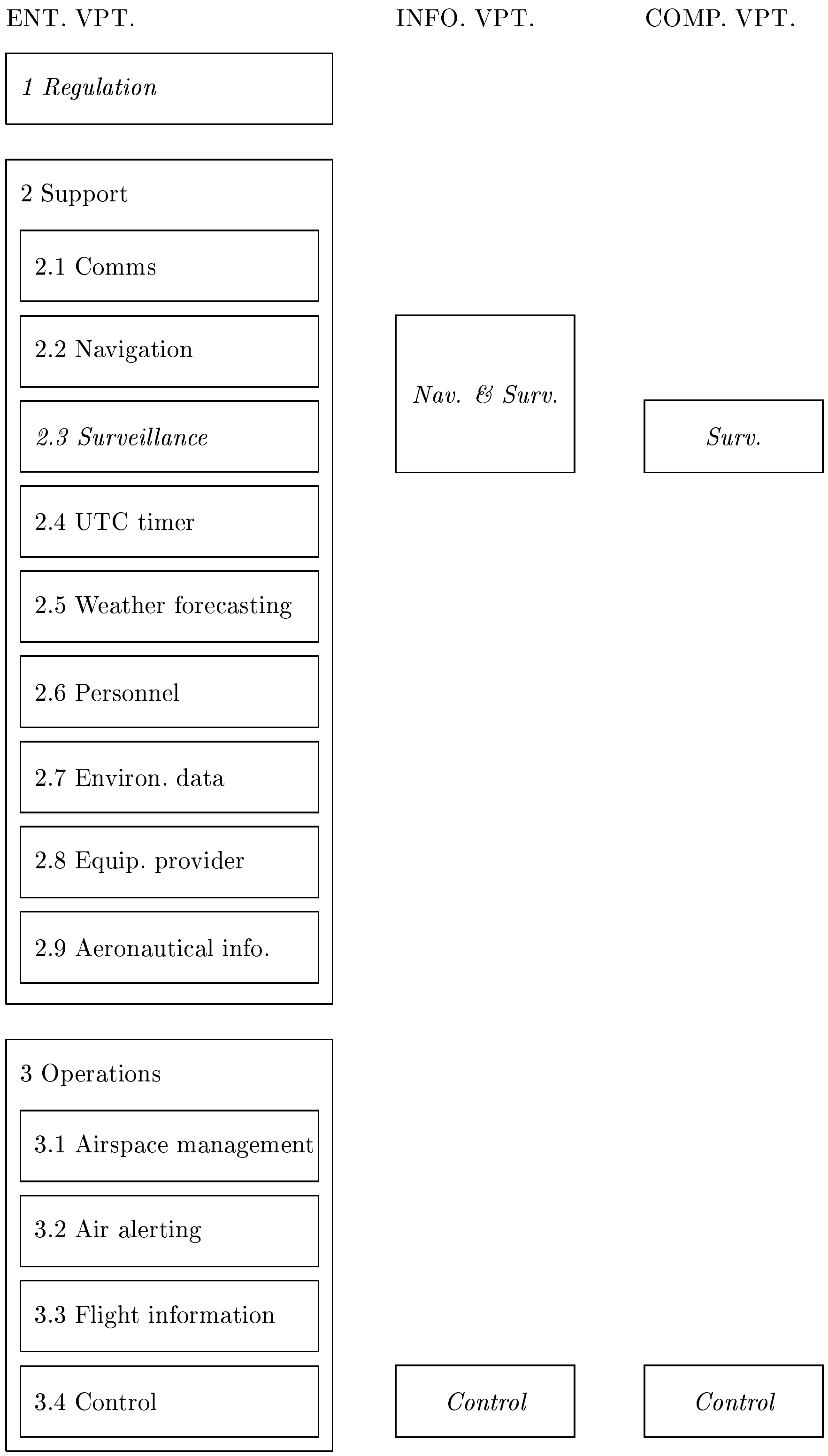
In the ECHO study, subsystems of the overall system are identified with "communities" in the terminology of the Enterprise Viewpoint — that is to say, with groups of actors serving a particular overall objective. The three top-level communities identified, and their objectives, are as follows:

- *Regulation.* Objective: To provide the rules and a structure of airspace in which operations can be carried out effectively and safely, and to ensure that operations work within this framework.
- *Support.* Objective: To supply services necessary for the operations community to operate.
- *Operations.* Objective: To provide the appropriate level of Air Traffic Service to airspace users.

The latter two are further broken down into subsystems or subcommunities. Detailed specifications are given for the Regulation, Control, Navigation, Surveillance, and Control communities. The Regulation system is specified from the Enterprise Viewpoint, and the Control system from the Information and Computational Viewpoints. Regarding Navigation and Surveillance, there is a combined specification from the Information Viewpoint, and a specification for the Surveillance system alone from the Enterprise and Computational Viewpoints. Figure 1 summarises the structure of the overall system specification in terms of viewpoints and named subsystems. Names in italics indicate parts of the system for which specifications are given in some detail.

Where subsystems are analysed in detail from the Enterprise Viewpoint, this is done by means of informal box and arrow diagrams in which the boxes represent either classes of actor, or other subsystems, and the arrows represent responsibilities for providing data of various kinds, in the directions indicated by the arrows. (For example, two arrows from the "Radar" class to the "SDPS" (Surveillance Data Processing System) class indicate that one or more members of the former class provide 2D and 3D positions to each member of the latter class.) There is also some supporting text. In the specification of the Surveillance subsystem, the five Enterprise "actor" type names form a proper subset of the Computational class names used in the Computational Viewpoint.

Figure 1: Structure of the ECHO specification



2.2 Information Viewpoint

The two detailed Information Viewpoint specifications (jointly for the Navigation and Surveillance subsystems, and for the Control subsystem) consist of UML-style diagrams, and some informal text. Various types of entity are named, and in some cases attributes are listed. No operations are specified for the Information Viewpoint, which seems to have been interpreted as a viewpoint for specifying datatypes used by the system, at a fairly high level of abstraction. From this perspective, the combination of the Navigation and Surveillance systems into one Information Viewpoint specification indicates a large overlap in the kinds of data that the two systems use, but does not imply that they are being regarded as a single subsystem.

2.3 Computational Viewpoint

The Computational Viewpoint specifications (for the Surveillance and Control subsystems) are expressed using informal text and diagrams. Classes of object are presented, with attributes and methods being listed by name. Attributes are assumed to come with implicit "get" and "put" (i.e. set) methods. Attribute types are not given explicitly, but can in many cases be inferred fairly obviously from attribute names which resemble the names of types identified in the Information Viewpoint. Many of the Computational Viewpoint classnames appear to correspond to types of entity whose instances would be fairly substantial distributed applications at the implementation level — e.g. databases of various kinds. For some methods, inputs and outputs are given, together with an informal "Offers" statement which describes the function of the method concerned, and a "Uses" list, stating the names of other methods (of the same class or of others) "used" by the method concerned.

The overall structure of each subsystem, in terms of object instances of the Computational classes, is not stated explicitly. For example, as regards the Control subsystem, there is no explicit specification of how many objects of each class are involved, or of when or how such objects are created or destroyed. There are no "main" classes for the various subsystems, and the nature of the boundary between the Control System and other subsystems is also not entirely clear — for example, it is not specified whether some Computational object instances are shared between functional subsystems. This is a significant issue, because it affects how the external interfaces of subsystems are defined, which in turn may affect how the Computational Viewpoint specification can be related to other viewpoints.

Box and link diagrams are provided, in which the boxes represent classes, and the links provide cardinality constraints between the classes, and in some cases a direction arrow, indicating that methods of one class act on objects of the other class pointed to, but not vice versa. For example, in the Computational Viewpoint specification of the Surveillance community, there is a one-to-one link from the SDPS (Surveillance Data Processing System) class to the FDPD (Flight Data Processing and Distribution) class. Both of these classes, and also the Aircraft and ATSU (Air Traffic Service Unit) classes, also occur in the Computational Viewpoint specification of another subsystem — the Control community. However, some of the links involving these classes are different in the two detailed subsystem specifications — e.g. in the Control community specification, no link is shown between the SDPS and ATSU classes. It is not clear whether the Control and Surveillance subsystems actually share some *instances* — i.e. individual objects — of the Computational classes that they have any common, or whether the sets of instances involved in the two subsystems are entirely disjoint. Again, this issue affects the way that interfaces can be defined between subsystems, which may also affect the way that correspondences can be drawn between the Computational Viewpoint and other viewpoints.

Methods whose names contain the prefix "spontaneous" are listed for some classes in the Computational Viewpoint. Such a method is said to be "a trigger which may be based on time or some other unspecified event (this is often the start point of a method thread)." Some such "spontaneous" methods could possibly be interpreted as "internal" operations, i.e. those which do not involve interaction with an external environment. "Use case scenarios" in the OID (Object Interaction Diagram) notation illustrate some behaviour associated with individual objects, although there are some discrepancies in terms of signature between these scenarios and the Computational Viewpoint, which arose because the scenarios were produced independently.

2.4 Observations and Issues Arising from the ECHO Study

Consideration of the ECHO study suggests the following inter-related points:

- *Subsystems versus viewpoints.* When viewpoint methods such as ODP are applied to very large and complex systems, there is a tendency to describe the system in terms of components or subsystems as well as in terms of viewpoints.

- *Subsystems in the ECHO study.* In the ECHO case, there is at least some correlation between major subsystems and the Enterprise Viewpoint description, insofar as several Enterprise "communities" are identified which seem to correspond to separate subsystems.

- *Subsystem interaction.* In an ODP specification, how should the interactions and interfaces between major subsystems be described, and how should this relate to the viewpoints? The Computational Viewpoint would seem the most appropriate one for dealing with this, but in the ECHO study, there is little correspondence between the classes listed for the Computational Viewpoint, and the major subsystems or "subcommunities" identified in the Enterprise Viewpoint. Even though some of the Computational Viewpoint classes appear to represent fairly substantial distributed applications, such as databases, they appear to be subsystems at a lower level than those identified as "communities" in the Enterprise Viewpoint.

- *Viewpoints and subsystem hierarchy.* If a system consists of a hierarchy of subsystems, how should the hierarchical structure be described in a viewpoint specification? For example, should each subsystem — at any of the levels, be specified from several viewpoints? Should the hierarchical structure be specified within one or more of the viewpoints — and if so, which ones?

- *Specification layout and structure.* How should an ODP specification — or indeed a viewpoint specification in general — be structured and organized?

- *Object instances versus classes.* In the ECHO study's detailed Computational Viewpoint specifications (of which there are two, for two specific subsystems), *classes* of distributed objects are defined, but the structure of each subsystem in terms of *instances* of those classes is not stated explicitly (although there are some cardinality constraints between classes, in informal diagrams), and there is no class corresponding to the whole subsystem. Such information, concerning how instances of Computational Viewpoint object classes are composed into an overall system or subsystem, would be a useful addition to the Computational Viewpoint.

3 A Partially Formalised ODP Specification

This section discusses a simple model of an ATC system, which addresses some of the issues arising from the ECHO study, using the formal languages Z [12] and Object-Z [11] to specify the Information and Computational Viewpoints, respectively. A realistic model of an ATC system would, of course, be far more complex than this one — the aim here is to use an idealised model to explore some of the general issues involved, not to provide an accurate model of an air traffic control system.

The ODP Reference Model is informally stated, goes into little detail about how the viewpoints or correspondences between them should be represented, and is deliberately not expressed in terms of any particular specification formalism or underlying formal model. Consequently, what is presented here is not claimed to be a definitive interpretation, but is one possible way of interpreting the outline provided by the Reference Model.

3.1 Enterprise Viewpoint

According to the informal account in the ODP Reference Model, the Enterprise Viewpoint should represent the system as a "community" of "actors" of certain types, each serving a role in order to satisfy some overall system objective. The structure described is rather "flat", consisting of a single level of actors, serving one objective. It is not very clear how a hierarchical division into subsystems should be integrated with such a picture. The ECHO study provides one interpretation, by using the Enterprise Viewpoint to identify a hierarchy of named subsystems — each of which is then regarded as an Enterprise "community" in its own right — as well as introducing some classes of distributed objects which are subsequently used as a subset of those referred to in the Computational Viewpoint. It is not made clear, however, whether the subsystems are viewed as overlapping, in terms of the distributed object instances that they involve. Another way of interpreting the Reference Model might be to allow some overlap between the concepts of “actor” and “community”, i.e. to allow at least some of the “actors” in a top-level Enterprise Viewpoint specification to be treated as “communities” themselves, which are then analysed in more detail — from multiple ODP viewpoints if necessary — at a lower level.

In our simplified air traffic control specification, we assume that the Enterprise Viewpoint consists of an informal statement of the hierarchy of subsystems involved, and of their objectives. (An alternative approach, based on a language for specifying Enterprise “policies” (i.e. permissions, prohibitions, and obligations) — and a translation from that language into Object-Z — is described in [13].) As just suggested, some of these subsystems could be thought of as being both “actors” and “communities” (or alternatively, the Reference Model should be interpreted in such a way that it allows a community to consist of several “subcommunities”, as was assumed in the ECHO study). In the Computational Viewpoint, these subsystems will be identified with instances of object classes, and any overlap between them, in terms of common components, will be stated explicitly — but in the Enterprise Viewpoint, we merely state their functional objectives.

Like the system described in the ECHO study, the one described in this section has “communities” or subsystems at more than one level. At the top-level, the overall system has two subsystems: a “control” system, whose objective is to allocate flights and resolve conflicts, and a “support” system, whose objective is to provide and update flight data. The “control” system has two lower-level subsystems — a “flight manager”, whose objective is to make the decisions, and a “flight database”, whose objective is to keep track of flight information. The “support” system has two lower-level components — a “surveillance” subsystem for collecting data, and a “flight database” for storing the data. In the Computational Viewpoint, the flight databases of the “control” and “support” subsystems are identified as being in fact the same entity. The Enterprise Viewpoint could also be used — as in the ECHO study — to list types of actor other than the subsystems themselves — e.g. “radars”, “controllers”, etc. — that are subsequently used as some of the classes in the Computational Viewpoint.

3.2 Information Viewpoint

In the ECHO study, the Information Viewpoint is applied separately to parts of the overall system. It may indeed in many cases be useful to provide such subsystem specifications from the Information Viewpoint, particularly as a way of structuring the presentation of datatypes (as in the ECHO study). However, we suggest that it would also be helpful for the Information Viewpoint to provide a high-level, fairly abstract specification of the system as a whole. In addition to defining datatypes used as class attribute types in the Computational Viewpoint, this should adefine an abstract system state, and high-level operations (which can give an initial indication of the top-level operations required in the Computational Viewpoint). The specification in our simplified air traffic control model, described in detail in this section, is a specification of this kind. It would have to be refined further in practice, for example by providing definitions of certain global predicates which are not defined in detail, such as those relating to conflict. Likewise, the operations defined would require further refinement, to impose additional constraints, and further operations might need to be added.

Coordinates, points, and paths. Latitudes, longitudes, and aircraft types are represented here abstractly as given types, and “flight levels” and times as natural numbers. A “4D point” is a 4-tuple consisting of a latitude, longitude, flight level, and time. The function *time* returns the fourth element of such a tuple, i.e. its time coordinate.

$$[Lat, Long, AircraftType]$$
$$FlightLevel == \mathbb{N}$$
$$Time == \mathbb{N}$$
$$Pt_{4D} == Lat \times Long \times FlightLevel \times Time$$

$$time : Pt_{4D} \rightarrow Time$$

$$\forall (lat, long, fl, t) : Pt_{4D} \bullet time((lat, long, fl, t)) = t$$

A "4D path" is a spatio-temporal trajectory — represented by a finite sequence of 4D points — that is possible for a specified type of aircraft. The "start time" of a (non-empty) path is the time coordinate of the first 4D point in that path's sequence of 4D points.

$$
\begin{array}{|l}
\hline
\mathit{Path}_{4D} \\
\hline
\mathit{atype} : \mathit{AircraftType} \\
\mathit{pts} : \mathrm{seq}\, \mathit{Pt}_{4D} \\
\hline
\mathit{pts}\ \underline{\mathit{possibleFor}}\ \mathit{atype} \\
\hline
\end{array}
$$

$$
(\_\ \underline{\mathit{possibleFor}}\ \_) : \mathrm{seq}\, \mathit{Pt}_{4D} \leftrightarrow \mathit{AircraftType}
$$

$$
\begin{array}{|l}
\mathit{startTime} : \mathit{Path}_{4D} \nrightarrow \mathit{Time} \\
\hline
\mathrm{dom}\, \mathit{startTime} = \{ p : \mathit{Path}_{4D} \mid p.\mathit{pts} \neq \langle\, \rangle \} \\
\forall\, p : \mathit{Path}_{4D} \bullet p \in \mathrm{dom}\, \mathit{startTime} \Rightarrow \mathit{startTime}(p) = \mathit{time}(\mathit{head}(p.\mathit{pts}))
\end{array}
$$

Flights and Conflict. The main functions of an ATC system are to assign and monitor flights, and to resolve any conflicts that arise. Our idealised model assumes four broad types of conflict, identified in the ECHO case study:

(1) **Aircraft–aircraft (AA) conflicts** — those involving unacceptable proximity between two aircraft.

(2) **Deviation conflicts** — when an aircraft's actual path deviates too far from its assigned path.

(3) **Request conflicts** — in which an aircraft requests an alteration to its assigned path.

(4) **Resource conflicts** — those in which an aircraft's path conflicts with an "environmental object", such as a mountain or an airspace boundary.

The four conflict types are formalised differently — (1) as a relation between flights, (2) and (3) as properties of flights, and (4) as a relation between flights and environment objects. Accordingly, the *Flight* schema has three 4D paths for a particular aircraft type *atype*, representing the path actually taken so far, the path mostly recently assigned by air traffic control, and the path most recently requested by the aircraft (either before take-off, or to make an inflight alteration). In this simplified model, the assigned and actual paths are required to have the same start time.

$$
\begin{array}{|l}
\hline
\mathit{Flight} \\
\hline
\mathit{atype} : \mathit{AircraftType} \\
\mathit{actualPath}_{4D}, \mathit{assignedPath}_{4D}, \mathit{requestedPath}_{4D} : \mathit{Path}_{4D} \\
\hline
\mathit{atype} = \mathit{actualPath}_{4D}.\mathit{atype} \\
= \mathit{assignedPath}_{4D}.\mathit{atype} = \mathit{requestedPath}_{4D}.\mathit{atype} \\
\mathit{startTime}(\mathit{assignedPath}_{4D}) = \mathit{startTime}(\mathit{actualPath}_{4D}) \\
\hline
\end{array}
$$

In a more detailed specification, further invariants would need to be added to this schema, to impose further constraints on the inter-relations between the three paths.

The four types of conflict involving flights are defined by assuming three relations of conflict between 4D paths, which would have to be constrained further in a fuller specification. "AA conflict" (aircraft–aircraft conflict) is defined as involving two paths belonging to different flights — either two actual paths, two assigned paths, or an assigned path and an actual path. Generic definitions of symmetric, reflexive, and irreflexive relations, for an arbitrary type X, are used in the definitions relating to conflict.

$$
\begin{array}{l}
\mathit{symmReln}[X] == \{ R : X \leftrightarrow X \mid \forall\, x, y : X \bullet (x, y) \in X \Rightarrow (y, x) \in X \} \\
\mathit{reflexReln}[X] == \{ R : X \leftrightarrow X \mid \forall\, x : X \bullet (x, x) \in X \} \\
\mathit{irreflexReln}[X] == \{ R : X \leftrightarrow X \mid \forall\, x : X \bullet (x, x) \notin X \}
\end{array}
$$

$$\begin{array}{|l}
inAAConflict, inDevConflict, inReqConflict : symmReln[Path_{4D}] \\
\hline
\{inDevConflict, inReqConflict\} \subseteq irreflexReln[Path_{4D}] \\
inAAConflict \in reflexReln[Path_{4D}]
\end{array}$$

$$\begin{array}{l}
DevConflict == \{f : Flight \mid \\
\quad (f.actualPath_{4D}, f.assignedPath_{4D}) \in inDevConflict\} \\
ReqConflict == \{f : Flight \mid \\
\quad (f.assignedPath_{4D}, f.requestedPath_{4D}) \in inReqConflict\} \\
AAConflict == \{(f_1, f_2) : Flight \times Flight \mid f_1 \neq f_2 \wedge \\
\quad ((f_1.actualPath_{4D}, f_1.assignedPath_{4D}) \in inAAConflict \\
\quad \vee (f_1.assignedPath_{4D}, f_2.actualPath_{4D}) \in inAAConflict \\
\quad \vee (f_1.actualPath_{4D}, f_2.actualPath_{4D}) \in inAAConflict \\
\quad \vee (f_1.assignedPath_{4D}, f_2.assignedPath_{4D}) \in inAAConflict)\}
\end{array}$$

A "resource conflict" relation is declared (but not precisely defined) between 4D paths and "environmental objects" (which are either airspaces or physical objects, such as mountains).

$$[Airspace, PhysObj]$$

$$\begin{array}{lcl}
EnvObj & ::= & airspace\langle\langle Airspace\rangle\rangle \\
 & \mid & physObj\langle\langle PhysObj\rangle\rangle
\end{array}$$

$$\begin{array}{|l}
ResConflict : Flight \leftrightarrow EnvObj
\end{array}$$

System state schema. The abstract system state has as its main attributes a system time, a finite set of environmental objects, a flight index, and a set of "current flights" (those in progress or waiting for take-off) which is a subset of the flights indexed. The flight index is an injective sequence of flights (so flights are uniquely numbered). Other attributes, whose values are determined by state invariants, record the various kinds of conflicts currently present, and the set of all current flights which are involved in some kind of conflict. The initialisation schema $Sys\textit{INIT}$ requires that in an initial state, the system time is 0, and the flight index is the empty sequence (hence, given the state invariants, there are no current flights or flights in conflict).

$$\begin{array}{|l}
\multicolumn{1}{l}{Sys} \\
\hline
sysTime : Time \\
envObjs : \mathbb{F}\, EnvObj \\
flightIndex : \mathrm{iseq}\, Flight \\
currentFlights, allFlightsInConflict : \mathbb{F}\, Flight \\
aaConflicts : \mathbb{F}(Flight \times Flight) \\
devConflicts, reqConflicts : \mathbb{F}\, Flight \\
resConflicts : \mathbb{F}(Flight \times EnvObj) \\
\hline
allFlightsInConflict \subseteq currentFlights \subseteq \mathrm{ran}\, flightIndex \\
aaConflicts = AAConflict \cap (currentFlights \times currentFlights) \\
devConflicts = DevConflict \cap currentFlights \\
reqConflicts = ReqConflict \cap currentFlights \\
resConflicts = ResConflict \cap (currentFlights \times envObjs) \\
allFlightsInConflict = \\
\quad devConflicts \cup reqConflicts \cup \mathrm{dom}\, aaConflicts \cup \mathrm{dom}\, resConflicts \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{Sys\textit{INIT}} \\
\hline
Sys' \\
\hline
sysTime' = 0 \wedge flightIndex' = \langle\,\rangle \\
\hline
\end{array}$$

Operations. The operations defined are: *Tick*, *AssignFlight*, *TakeOff*, *Landing*, *FlightObs*, and *ResolveConflict*. All of these increment the system time, but in the case of the *Tick* operation, nothing else changes.

$$
\begin{array}{|l}
\textit{Tick} \\
\hline
\Delta \mathit{Sys} \\
\hline
\mathit{flightIndex}' = \mathit{flightIndex} \land \mathit{currentFlights}' = \mathit{currentFlights} \\
\mathit{envObjs}' = \mathit{envObjs} \land \mathit{sysTime}' = \mathit{sysTime} + 1 \\
\hline
\end{array}
$$

The *AssignFlight* operation adds a new flight $f!$ to the set of current flights (those already assigned — either in progress, or awaiting take-off) and to the index of all flights. Its outputs are the flight $f!$, and the start time $t!$ of its assigned flightpath, which must be different to those of other current flights, and later than the current system time. (In this simplistic model,, we assume that we are dealing with the air traffic control system of a single airport, and that simultaneous take-offs are ruled out on safety grounds, even if there is more than one runway.) The sequence of points of the actual path of the new flight must be empty (because the flight has not taken off yet). The new flight must not result in any new conflicts arising.

$$
\begin{array}{|l}
\textit{AssignFlight} \\
\hline
\Delta \mathit{Sys} \\
f! : \mathit{Flight};\ t! : \mathit{Time} \\
\hline
\mathit{startTime}(f!.\mathit{assignedPath}_{4D}) = t! \land t! > \mathit{sysTime} \\
\forall f : \mathit{currentFlights} \bullet \mathit{startTime}(f.\mathit{assignedPath}_{4D}) \neq t! \\
\mathit{flightIndex}' = \mathit{flightIndex} \frown \langle f! \rangle \\
\mathit{currentFlights}' = \mathit{currentFlights} \cup \{f!\} \\
f!.\mathit{actualPath}_{4D}.\mathit{pts} = \langle\,\rangle \\
\mathit{allFlightsInConflict}' = \mathit{allFlightsInConflict} \\
\mathit{envObjs}' = \mathit{envObjs} \land \mathit{sysTime}' = \mathit{sysTime} + 1 \\
\hline
\end{array}
$$

The *TakeOff* operation represents an aircraft taking off. Its inputs are a flight number $n?$, which is one of those in the domain of the flight index, and a flight $f?$, which is the flight indexed by that $n?$. The output of the operation is a flight $f!$, identical to $f?$, except that its actual 4D path is now non-empty, with its sequence of points containing one point — the first 4D point in the assigned path of $f?$. $f?$ must be a current flight which is not in conflict. In this simplified model, *TakeOff* is assumed to occur at the scheduled take-off time as specified in the assigned path.

$$
\begin{array}{|l}
\textit{TakeOff} \\
\hline
\Delta \mathit{Sys} \\
n? : \mathbb{N};\ f?, f! : \mathit{Flight} \\
\hline
n? \in \mathrm{dom}\, \mathit{flightIndex} \land \mathit{flightIndex}(n?) = f? \\
f? \in \mathit{currentFlights} \land f? \notin \mathit{allFlightsInConflict} \\
\mathit{startTime}(f?.\mathit{assignedPath}_{4D}) = \mathit{sysTime} \\
\mathit{flightIndex}' = \mathit{flightIndex} \oplus \{n? \mapsto f!\} \\
f! = \lparr\ \mathit{atype} == f?.\mathit{atype}, \\
\qquad \mathit{assignedPath}_{4D} == f?.\mathit{assignedPath}_{4D}, \\
\qquad \mathit{requestedPath}_{4D} == f?.\mathit{requestedPath}_{4D}, \\
\qquad \mathit{actualPath}_{4D} == \\
\qquad\qquad \lparr\ \mathit{atype} == f?.\mathit{atype}, \mathit{pts} == \langle \mathit{head}\, f?.\mathit{assignedPath}_{4D}.\mathit{pts} \rangle\ \rparr\ \rparr \\
\mathit{currentFlights}' = (\mathit{currentFlights} \setminus \{f?\}) \cup \{f!\} \\
\mathit{envObjs}' = \mathit{envObjs} \land \mathit{sysTime}' = \mathit{sysTime} + 1 \\
\hline
\end{array}
$$

The *Landing* operation models an aircraft landing. It removes a flight from the set of current flights (which, if it is in conflict, implicitly removes it from the flights in conflict, as well, given the invariants in the state schema *Sys*). In a more refined specification, further constraints would be imposed on this operation.

$$
\begin{array}{|l}
\hline
\textit{Landing} \\
\Delta Sys \\
f? : Flight \\
\hline
f? \in currentFlights \land currentFlights' = currentFlights \setminus \{f?\} \\
flightIndex' = flightIndex \\
envObjs' = envObjs \land sysTime' = sysTime + 1 \\
\hline
\end{array}
$$

FlightObs represents the receipt of an observation of an aircraft's position at a particular time (in the form of a 4D point), regarding one of the current flights. $f?$ is the input flight, and $f!$ the modified output flight, incorporating the new observation. The time coordinate of the observation must be less than or equal to the current system time. The constraints specified in the *Flight* datatype schema implicitly ensure that the observation here must be sensible for the flight, i.e. it must be possible for the type of aircraft involved, given the actual path of 4D points so far.

$$
\begin{array}{|l}
\hline
\textit{FlightObs} \\
\Delta Sys \\
n? : \mathbb{N};\ pt? : Pt_{4D};\ f?, f! : Flight \\
\hline
n? \in \operatorname{dom} flightIndex \land flightIndex(n?) = f? \\
f? \in currentFlights \land time(pt?) < sysTime \\
f?.actualPath_{4D}.pts \neq \langle\,\rangle \land time(pt?) > time(last(f?.actualPath_{4D}.pts)) \\
f! = \langle\!|\ atype == f?.atype, \\
\qquad actualPath_{4D} == \\
\qquad\qquad \langle\!|\ atype == f?.atype, pts == f?.actualPath_{4D}.pts \frown \langle pt? \rangle\ |\!\rangle, \\
\qquad requestedPath_{4D} == \\
\qquad\qquad \langle\!|\ atype == f?.atype, pts == f?.requestedPath_{4D}.pts\ |\!\rangle \\
\qquad assignedPath_{4D} == \\
\qquad\qquad \langle\!|\ atype == f?.atype, pts == f?.assignedPath_{4D}.pts\ |\!\rangle\ |\!\rangle \\
flightIndex' = flightIndex \oplus \{n? \mapsto f!\} \\
currentFlights' = (currentFlights \setminus \{f?\}) \cup \{f!\} \\
envObjs' = envObjs \land sysTime' = sysTime + 1 \\
\hline
\end{array}
$$

ResolveConflict represents in a very abstract way the resolution of a conflicting set *flights* of flights, which is a subset of the current flights in conflict. The partial injective functions *indexedFlights*? and *indexedFlights*! represent respectively the initial indexing of those flights (a subset of the overall flight index), and the revised indexing for the same flight numbers, after the conflicting set of flights has been replaced by modified flights. An invariant states that *flights*? is a "self-contained" conflicting set, in the sense that none of its members are in "AA conflict" (the only type of conflict involving pairs of flights) with any other flights currently in conflict. Further invariants state that *indexedFlights*! represents a "resolution for" *indexedFlights*?; that the new index $flightIndex'$ uses the revised flights as indexed by $indexedFlights'$; and that no new conflicts have been introduced as a result.

$$
\begin{array}{|l}
\hline
\textit{ResolveConflict} \\
\Delta Sys \\
flights? : \mathbb{F}\, Flights;\ indexedFlights?, indexedFlights! : \mathbb{N}_1 \rightarrowtail\!\!\!\!\!\!\!+\;\; Flight \\
\hline
indexedFlights? \subseteq flightIndex \\
\operatorname{dom} indexedFlights? = \operatorname{dom} indexedFlights! \\
flights? = \operatorname{ran} indexedFlights? \land flights? \subseteq allFlightsInConflict \\
aaConflicts \cap (flights? \times (allFlightsInConflict \setminus flights?)) = \varnothing \\
indexedFlights!\ \underline{resolutionFor}\ indexedFlights?\ \underline{wrt}\ envObjs \\
flightIndex' = flightIndex \oplus indexedFlights! \\
currentFlights' = (currentFlights \setminus flights?) \cup \operatorname{ran} indexedFlights! \\
allFlightsInConflict' = allFlightsInConflict \setminus flights? \\
envObjs' = envObjs \land sysTime' = sysTime + 1 \\
\hline
\end{array}
$$

The 3-place conflict resolution used in this operation schema is declared as follows:

$$(\_ \underline{resolutionFor} \_ \underline{wrt} \_) : \mathbb{P}((\mathbb{N}_1 \rightarrowtail\!\!\!\!\!\twoheadrightarrow Flight) \times (\mathbb{N}_1 \rightarrowtail\!\!\!\!\!\twoheadrightarrow Flight) \times (\mathbb{F}\, EnvObj))$$

$$\begin{array}{l} \forall fn_1, fn_2 : (\mathbb{N}_1 \rightarrowtail\!\!\!\!\!\twoheadrightarrow Flight);\ objSet : \mathbb{F}\, EnvObj \bullet \\ \quad ((fn_1\ \underline{resolutionFor}\ fn_2\ \underline{wrt}\ objSet) \Rightarrow \mathrm{dom}\, fn_1 = \mathrm{dom}\, fn_2) \\ \quad \land (\forall n : \mathbb{N}_1 \bullet n \in \mathrm{dom}\, fn_1 \Rightarrow fn_1(n).atype = fn_2(n).atype) \\ \quad \land \neg (fn_1\ \underline{conflictFreeWrt}\ objSet) \\ \quad \land (fn_2\ \underline{conflictFreeWrt}\ objSet) \end{array}$$

Here, fn_1 is an indexed set of flights in which at least some conflicts are present, with respect to a set $objSet$ of environmental objects, and fn_2 is a second set of flights (with corresponding numerical indices, and corresponding aircraft types for flights with the same index number) that is conflict-free with respect to the same set $objSet$.

3.3 Computational Viewpoint Specification

The Computational Viewpoint specification is expressed in Object-Z [11], a language which, given its intrinsically object-oriented nature, is better suited than standard Z to the specification of systems of distributed objects. An Object-Z specification includes several *class schemas*, each corresponding to an ADT, of which one represents the type of system being modelled, e.g. in our case study, the *Main* class. Variable declarations such as $x : ClassName$ are allowed, where x denotes an unique identifier for, or pointer to, an object of the class *Classname*. If *Op* is an operation of class *Classname*, the notation $x.Op$ represents the execution of *Op* on the object to which x refers. For example, in the specification given shortly, the *Main* class has an attribute *clock* of class *Clock*, and the expression $clock.Tick$ represents the execution on that clock of its own (lower-level) *Tick* operation. When a higher-level operation is defined in this way, by promoting an operation on a component object, the higher-level operation implicitly has the same input and output variables (if any) as the component object operation being promoted, and the higher-level operation implicitly has an empty Δ-list — so that it does not change any attributes of the higher-level object.

Each operation has an optional Δ-list, showing attributes that it allows to change — attributes not in the Δ-list are not changed by that operation (unless they are 'secondary' attributes, separated from the main attributes by a Δ symbol, in which case they may change in any way consistent with preserving the state invariant). Class operations and attributes can be declared as "public", or not, at the top of each class — if such declarations are not shown (as they are not in the following specification), this means that all operations and attributes are visible. Object-Z provides several operators for combining operations — including $\land$ (schema conjunction), and $\|$. The latter operator implicitly introduces invariants which equate the output variables of one operation with similarly named input variables of another (e.g. if $Op1$ has an output variable $x!$ and $Op2$ has an input variable $x?$, then the operation $Op1 \| Op2$ includes an implicit invariant $x! = x?$).

The "communities" identified informally in the Enterprise Viewpoint are equated with particular instances of some of the classes specified in the Computational Viewpoint. Since particular instances of Object-Z classes may share component objects, specifying the communities in this way allows the different communities to overlap in terms of the objects involved — something that is quite likely in a system composed of distributed objects, in which some individual objects may operate as part of several ensembles of objects, each of which collectively performs a particular system function. (For example, an individual server might be shared by several functional groups of system components.)

The main focus in an initial Computational Viewpoint specification should be to give a broad indication of system structure, in terms of identifiable subsystems, and the objects of which they are composed. As regards operations, the aim is to specify them in outline only, in terms of the objects that they involve, whether they involve synchronisations of lower-level operations, and so forth. Operations are thus associated with particular distributed objects, rather than being defined in a purely abstract way, as they were in the Information Viewpoint specification. Detailed constraints on the inputs and outputs of operations, and operation invariants, can be left to subsequent refinement of those operations, which might also involve adding extra inputs and outputs. Some such refinement may occur in the course of unification with other viewpoints — particularly with the Information Viewpoint.

Main system. The overall system is viewed as an instance of the *Main* class. At the top level of abstraction, an object of this class has a subsystem *control* of type *Control*, for allocating flights and detecting and resolving conflicts, and a subsystem *support* of type *Support*, which provides and updates the data required by *control*. The two subsystems share a database *flightDbase*, which holds information about flights. (The subscript © is an abbreviated notation in Object-Z for "object containment" — for example, the attribute declaration $control : Control_{©}$ implies a global invariant stating that each instance of the *Main* class "uniquely contains" its own object instance *control* of type *Control*, which cannot be shared with other instances of *Main*.) The *AssignFlight* operation involves a synchronisation of a request for a flight from the support subsystem, with the actual assignment of a flight by the control subsystem. The operations *FlightObs*, *Landing*, and *TakeOff* are promotions of the identically named operations of the surveillance subsystem. *ResolveConflict* promotes the operation of the same name of the *Control* subsystem.

Main

$$control : Control_{©}$$
$$support : Support_{©}$$
$$clock : Clock$$

$$control.flightDbase = support.flightDbase$$
$$clock = control.clock = support.clock$$

$$\mathrm{INIT} \mathrel{\widehat{=}} clock.\mathrm{INIT} \wedge control.\mathrm{INIT} \wedge support.\mathrm{INIT}$$
$$Tick \mathrel{\widehat{=}} clock.Tick$$
$$AssignFlight \mathrel{\widehat{=}} Tick \wedge (support.RequestFlight \parallel control.AssignFlight)$$
$$TakeOff \mathrel{\widehat{=}} Tick \wedge support.TakeOff$$
$$Landing \mathrel{\widehat{=}} Tick \wedge support.Landing$$
$$FlightObs \mathrel{\widehat{=}} Tick \wedge support.FlightObs$$
$$ResolveConflict \mathrel{\widehat{=}} Tick \wedge control.ResolveConflict$$

The *Clock* class is defined as follows, with an attribute *time*, and an operation *Tick* which increments the value of *time*.

Clock

$$time : Time$$

$$\mathrm{INIT} \mathrel{\widehat{=}} time = 0$$
$$Tick \mathrel{\widehat{=}} [\, \Delta(time) \mid time' = time + 1 \,]$$

The clock object of a given instance of the *Main* class is shared by all the other direct and indirect component objects of that instance of *Main*. For the sake of clarity, the ticking of the clock is shown explicitly in all operations, at all levels — even though this is not strictly necessary, given that the clock is shared.

Control subsystem. An object of the class *Control* has a subsystem *flightManager*, which executes control functions, and a subsystem *flightDbase*, containing information about current flights. These two subsystems have a clock object in common. The *AssignFlight* operation, involving the assignment of a new flight, is represented as the synchronisation of a flight selection operation by the flight manager subsystem, and an operation on the flight database subsystem which records that selection. The *ResolveConflict* operation also involves a synchronisation of operations of the same name involving the flight database and the flight manager.

$$
\begin{array}{l}
\textit{Control} \\
\hline
\mathit{flightManager} : \mathit{FlightManager}_{\copyright} \\
\mathit{flightDbase} : \mathit{FlightDbase}_{\copyright} \\
\mathit{clock} : \mathit{Clock} \\
\hline
\mathit{clock} = \mathit{flightManager.clock} = \mathit{flightDatabase.clock} \\
\hline
\mathrm{INIT} \mathrel{\widehat=} \mathit{clock}.\mathrm{INIT} \wedge \mathit{flightManager}.\mathrm{INIT} \wedge \mathit{flightDbase}.\mathrm{INIT} \\
\mathit{AssignFlight} \mathrel{\widehat=} \mathit{Tick} \wedge \\
\quad (\mathit{flightManager.SelectFlight} \| \mathit{flightDbase.AssignFlight}) \\
\mathit{ResolveConflict} \mathrel{\widehat=} \mathit{Tick} \wedge \\
\quad (\mathit{flightDbase.ResolveConflictSet} \| \mathit{flightManager.ResolveConflict}) \\
\mathit{Tick} \mathrel{\widehat=} \mathit{clock.Tick}
\end{array}
$$

An object of the class *FlightManager* has as its attributes a finite set of one or more "controllers", and a clock (which is common to all the controllers). The *SelectFlight* and *ResolveConflict* operations represent the selection of a flight and the resolution of a conflict set, respectively, by a particular controller. These are promotions of identically named operations of the class *Controller*.

$$
\begin{array}{l}
\textit{FlightManager} \\
\hline
\mathit{controllers} : \mathbb{F}_1\ \mathit{Controller} \\
\mathit{clock} : \mathit{Clock} \\
\hline
\forall\, c : \mathit{controllers} \bullet c.\mathit{clock} = \mathit{clock} \\
\hline
\mathrm{INIT} \mathrel{\widehat=} \mathit{clock}.\mathrm{INIT} \wedge (\bigwedge c : \mathit{controllers} \bullet c.\mathrm{INIT}) \\
\mathit{SelectFlight} \mathrel{\widehat=} \mathit{Tick} \wedge [c1 : \mathit{controllers}] \bullet \\
\quad c1.\mathit{SelectFlight} \wedge (\bigwedge c2 : \mathit{controllers} \setminus \{c1\} \bullet c2.\mathit{Tick}) \\
\mathit{ResolveConflict} \mathrel{\widehat=} \mathit{Tick} \wedge [c1 : \mathit{controllers}] \bullet \\
\quad c1.\mathit{ResolveConflict} \wedge (\bigwedge c2 : \mathit{controllers} \setminus \{c1\} \bullet c2.\mathit{Tick}) \\
\mathit{Tick} \mathrel{\widehat=} \mathit{clock.Tick}
\end{array}
$$

The *Controller* class has a single attribute — a clock. The *SelectFlight* operation represents the selection of a flight by a controller. Its input is an aircraft type, and its outputs are a flight and a take-off time. This operation is defined only in terms of inputs and outputs here — it would be refined further to impose constraints on the inputs and outputs. The same comment applies to the *ResolveConflict* operation, which represents the resolution of a particular set of conflicting flights by an individual controller.

$$
\begin{array}{l}
\textit{Controller} \\
\hline
\mathit{clock} : \mathit{Clock} \\
\hline
\mathrm{INIT} \mathrel{\widehat=} \mathit{clock}.\mathrm{INIT} \\
\mathit{SelectFlight} \mathrel{\widehat=} \mathit{Tick} \wedge \\
\quad [\, \mathit{atype}? : \mathit{AircraftType};\ f! : \mathit{Flight};\ t! : \mathit{Time} \,] \\
\mathit{ResolveConflict} \mathrel{\widehat=} \\
\quad [\, \mathit{flights}? : \mathbb{F}_1\ \mathit{Flight};\ \mathit{oldIndex}?, \mathit{newIndex}! : \mathbb{N}_1 \rightarrowtail\!\!\!\!\twoheadrightarrow \mathit{Flight} \,] \\
\mathit{Tick} \mathrel{\widehat=} \mathit{clock.Tick}
\end{array}
$$

The *FlightDbase* class has a state schema very similar to the global state schema *Sys* in the Z specification for the Information Viewpoint. As in the ECHO study, it assumes datatypes (such as *Flight*) and global definitions of sets, functions, etc., that are already defined in the previously given Information Viewpoint specification. (N.B. An Object-Z specification can include conventional Z schema type declarations and global definitions, so this is unproblematical.) The operations *AssignFlight*, *TakeOff*, *Landing*, *FlightObs*, and *ResolveConflict* are given in outline form here, simply with inputs and/or outputs of specific types. They would need to be refined later, in order to incorporate the constraints embodied in the operations of the same names in the top-level state of the Information Viewpoint specification.

$$
\begin{array}{l}
\textit{FlightDbase} \\
\hline
\quad envObjs : \mathbb{F}\, EnvObj \\
\quad flightIndex : \mathrm{iseq}\, Flight \\
\quad currentFlights : \mathbb{F}\, Flight \\
\quad clock : Clock \\
\quad \Delta \\
\quad allFlightsInConflict, reqConflicts, devConflicts : \mathbb{F}\, Flight \\
\quad aaConflicts : \mathbb{F}(Flight \times Flight) \\
\quad resConflicts : \mathbb{F}(Flight \times EnvObj) \\
\hline
\quad allFlightsInConflict \subseteq currentFlights \subseteq \mathrm{ran}\, flightIndex \\
\quad aaConflicts = AAConflict \cap (currentFlights \times currentFlights) \\
\quad devConflicts = DevConflict \cap currentFlights \\
\quad reqConflicts = ReqConflict \cap currentFlights \\
\quad resConflicts = ResConflict \cap (currentFlights \times envObjs) \\
\quad allFlightsInConflict = devConflicts \cup reqConflicts \\
\qquad \cup (\mathrm{dom}\, aaConflicts) \cup (\mathrm{dom}\, resConflicts) \\
\hline
\mathit{INIT} \mathrel{\widehat{=}} clock.\mathit{INIT} \wedge flightIndex = \langle\, \rangle \\
AssignFlight \mathrel{\widehat{=}} Tick \wedge [\, \Delta(flightIndex, currentFlights) \\
\qquad atype? : AircraftType;\ f? : Flight;\ t! : Time\,] \\
TakeOff \mathrel{\widehat{=}} Tick \wedge [\, \Delta(flightIndex, currentFlights) fNo? : \mathbb{N}_1\,] \\
Landing \mathrel{\widehat{=}} Tick \wedge [\, \Delta(currentFlights) fNo? : \mathbb{N}_1\,] \\
FlightObs \mathrel{\widehat{=}} Tick \wedge \\
\qquad [\, \Delta(flightIndex, currentFlights) fNo? : \mathbb{N}_1;\ pt? : Pt_{4D}\,] \\
ResolveConflict \mathrel{\widehat{=}} Tick \wedge [\, \Delta(flightIndex, currentFlights) \\
\qquad newIndex? : \mathbb{N}_1 \rightarrowtail\!\!\!\!\!\rightarrow Flight\,] \\
Tick \mathrel{\widehat{=}} clock.Tick
\end{array}
$$

Support subsystem. An object of the class *Support* consists of a surveillance subsystem, a flight database subsystem, and a clock, which is common to both of those subsystems. The operation *FlightObs* represents the receipt of a mid-flight observation, and the operations *Landing* and *TakeOff* represent the registering of a landing and a take-off, respectively. These three operations are analysed as sychronisations of a surveillance operation with a corresponding update operation affecting the flight database. The operation *RequestFlight* represents the receipt by the surveillance system of a request for a flight to be assigned.

$$
\begin{array}{l}
\textit{Support} \\
\hline
\quad surv : Surv_{©} \\
\quad flightDbase : FlightDbase_{©} \\
\quad clock : Clock \\
\hline
\quad clock = surv.clock = flightDbase.clock \\
\hline
\mathit{INIT} \mathrel{\widehat{=}} clock.\mathit{INIT} \wedge surv.\mathit{INIT} \wedge flightDbase.\mathit{INIT} \\
FlightObs \mathrel{\widehat{=}} Tick \wedge (surv.FlightObs \| flightDbase.FlightObs) \\
Landing \mathrel{\widehat{=}} Tick \wedge (surv.Landing \| flightDbase.Landing) \\
TakeOff \mathrel{\widehat{=}} Tick \wedge (surv.TakeOff \| flightDbase.TakeOff) \\
RequestFlight \mathrel{\widehat{=}} Tick \wedge surv.RequestFlight \\
Tick \mathrel{\widehat{=}} clock.Tick
\end{array}
$$

An object of the *Surv* (surveillance system) class consists of a set of one or more radar stations, and a clock, which is also a component of all the radars. The *FlightObs* models the receipt of an observation from one of the radars. The *Landing* and *TakeOff* operations are assumed to be observed directly, and so do not involve a radar station. The operation *RequestFlight* represents very abstractly the receipt of a request for a flight to be assigned for an aircraft of a specific type *atype*!.

Surv

$$
\begin{array}{l}
radars : \mathbb{F}_1\ Radar \\
clock : Clock \\
\hline
\forall\, r : radars \bullet r.clock = clock
\end{array}
$$

$$
\begin{array}{l}
\textsc{Init} \mathrel{\widehat=} clock.\textsc{Init} \land (\bigwedge r : radars \bullet r.\textsc{Init}) \\
FlightObs \mathrel{\widehat=} Tick \land [r1 : radars] \bullet \\
\qquad r1.FlightObs \land (\bigwedge r2 : radars \setminus \{r1\} \bullet r2.Tick) \\
Landing \mathrel{\widehat=} Tick \land [\, fNo! : \mathbb{N}_1 \,] \\
TakeOff \mathrel{\widehat=} Tick \land [\, fNo! : \mathbb{N}_1 \,] \\
RequestFlight \mathrel{\widehat=} Tick \land [\, atype! : AircraftType \,] \\
Tick \mathrel{\widehat=} Tick \land (\bigwedge r : radars \bullet r.Tick)
\end{array}
$$

Objects of the *Radar* class are modelled very abstractly, with only a single attribute — a clock. The operation *FlightObs* represents the receipt of a 4D point observation for a particular flight number (with the time coordinate being the current clock time), plus the ticking of the clock; and *Tick* represents the ticking of the clock only.

Radar

$$
\begin{array}{l}
clock : Clock
\end{array}
$$

$$
\begin{array}{l}
\textsc{Init} \mathrel{\widehat=} clock.\textsc{Init} \\
FlightObs \mathrel{\widehat=} Tick \land [\, fNo! : \mathbb{N}_1 ;\ pt! : Pt_{4D} \mid time(Pt_{4D}) = clock.time \,] \\
Tick \mathrel{\widehat=} clock.Tick
\end{array}
$$

4 Viewpoint Correspondences

Previous related work [2, 3] involving the authors and other colleagues has developed a general approach to viewpoint specification that is independent of particular formal specification languages, and that is applicable to, but not restricted to, ODP viewpoints. The central idea of this approach is that multiple viewpoints — in some cases expressed in several different languages — can be shown to be mutually consistent by developing a specification that is a common refinement of all the viewpoints, a process described as "unification". In practice this can be done step by step rather than in one go, unifying specifications a pair at a time, until ultimately a binary tree of specifications has been constructed, in which the leaves are the initial viewpoint specifications, and the root is their common refinement.

To apply this general approach to a particular formalism requires a well-defined notion of refinement for that formalism. When several formalisms are used, methods for translating from one formalism to another are also needed (the translations already considered include, for example, that from LOTOS to Object-Z [7]).

In this section, our aim is to consider what the relationships between ODP viewpoint specifications might typically look like, using the ECHO study and our simplified air traffic control model as an example. Given the informal nature of the ODP Reference Model, this is of course not a definitive interpretation, but one possible interpretation.

In the ECHO study, the roles of the Enterprise, Information, and Computational Viewpoints, and the main correspondences between them, can be summarised briefly as follows:

- *Enterprise Viewpoint*. This is used to outline the structure of the overall system, in the sense of equating "subcommunities" with subsystems, some of which are then specified in detail from the Information and Computational Viewpoints. Also, Enterprise "actor" type names are subsequently used as a proper subset of the class names in the Computational Viewpoint.

- *Information Viewpoint*. This is treated as a viewpoint for defining types of data used by the system, at a fairly high level of abstraction — no operations are defined for this viewpoint. The Information Viewpoint types are used as class attribute types in the Computational Viewpoint. Their presentation is subdivided into groups of related datatypes for individual Enterprise subsystems, or pairs of such subsystems, in one case where there is a large overlap in the datatypes used.

- *Computational Viewpoint*. This consists of classes defined by means of UML class diagrams, and informal text. Attributes and methods are listed — in some cases with inputs, outputs, and information about the purpose of classes and the names of other methods (of the same or different classes) that they use.

- In addition, the Technology Viewpoint is said to constrain the Enterprise Viewpoint (although no specification from the Technology viewpoint is provided).

In our simplified partially formalised model, we have suggested that the Information Viewpoint should be interpreted in such a way that it provides a fairly high-level, abstract formal specification of the state and operations of the overall system, as well as providing datatype definitions. The more concrete Computational Viewpoint specification can then be thought of as a particular, more implementation-oriented specification that gives more concrete information about system structure, in terms of distributed objects.

There should be a one-to-one correspondence between the top-level operations of the Information Viewpoint and the top-level operations of the *Main* class in the Computational Viewpoint. The latter operations will often be defined, however, in terms of lower-level Computational Viewpoint operations, in some cases involving synchronisations or sequential compositions of such lower-level operations. If the Information Viewpoint were used to specify individual subsystems as well as the overall system, then a similar one-to-one correspondence between Information and Computational Viewpoint operations should be obliged to hold at the top levels of each such subsystem.

A possible role for the Enterprise Viewpoint is to provide a starting point for the Computational Viewpoint by identifying functional subsystems and their objectives (as in the ECHO study and in our simplified model), without specifying the concrete objects used to implement them, or the extent to which the subsystems overlap in terms of objects. This could be done not only at the top level, but as a way of initiating the structuring of subsystems developed in the Computational Viewpoint. In addition, the Enterprise Viewpoint may identify some of the classes of object to be used in the Computational Viewpoint. As in the simplified air traffic control model described in this paper, the Enterprise Viewpoint need not be expressed formally, even if the Information and Enterprise Viewpoints are.

One general point that is apparent from the ECHO case study is that there is a need to interpret the ODP Reference Model in a way which allows hierarchical structure to be specified and subsystems to be identified. As regards the Enterprise Viewpoint, viewing the overall system as a single, one-level community of actors, serving a single objective, seems inadequate for some complex systems. Allowing some actors to be thought of as "communities" in their own right, which can then be analysed in more detail by a recursive application of the ODP viewpoints, is one possible way of interpreting the Reference Model that allows such hierarchical structuring. At lower levels, it might not be considered necessary to specify every subsystem from two or three viewpoints, e.g. a Computational Viewpoint specification alone might suffice.

The layout and organization of an ODP specification need not always consist simply of a sequence of separate viewpoint specifications. In some cases, it might be more convenient to intermingle the viewpoints to some extent, with the overall specification reflecting the hierarchical structure, and with different viewpoint specifications of the same subsystem being adjacent to one another. For large-scale specifications, software tools which can record and graphically display networks of specifications from different viewpoints, and their common refinements, would be a very useful aid. Such a framework of specifications could ultimately be used to structure an implementation into executable code.

5 Conclusion

This paper began with a description of a case study (the ECHO air traffic control system) in which an attempt was made to apply the ODP viewpoints framework to specify a complex distributed system, using three of the five ODP viewpoints — the Information, Computational, and Enterprise Viewpoints. Some observations concerning that study were used to raise some general issues relating to the interpretation of the ODP Reference Model. It was emphasised that an interpretation of the ODP Reference Model should provide some way of describing functional subsystems or components of the overall system — including how they are organised hierarchically, and how they are composed of *instances* of distributed objects, in some cases shared between different functional subsystems (an aspect which is not very explicit in the ECHO study). It was also suggested that the Information Viewpoint should provide not only a definition of datatypes (as in the ECHO study), but also a high-level abstract definition of overall system state and operations.

A simple, idealised air traffic control model was presented to explore some of these issues. In this model, the Enterprise Viewpoint was used informally to identify a hierarchy of the main functional subsystems and their objectives (as in the ECHO study). The language Object-Z was used to describe the structure of distributed objects in the Computational Viewpoint. The state-based language Z was used to provide a high-level, abstract description of datatypes and top-level system operations (which corresponded one-to-one with the operations of the top-level *Main* class in the Computational Viewpoint). It was suggested that such a more explicit representation of these aspects of system structure and behaviour could usefully augment those aspects focussed upon in the ECHO study, such as the identification of Information Viewpoint datatypes and Computational Viewpoint classes.

References

[1] E.A. Boiten, J. Derrick, H. Bowman, and M.W.A. Steen, "Constructive Consistency Checking for Partial Specification in Z", *Science of Computer Programming*, 35, 1999, pp. 29–75.

[2] E.A. Boiten, H. Bowman, J. Derrick, P.F. Linington, and M.W.A. Steen, "Viewpoint Consistency in ODP", *Computer Networks*, pp. 503–537, August 2000.

[3] H. Bowman, E.A. Boiten, J. Derrick, and M.W.A. Steen, "Viewpoint Consistency in ODP, a General Interpretation", *Formal Methods for Open Object-Based Distributed Systems*, eds. E. Najm and J.-B. Stefani, Chapman & Hall, March 1996, pp. 189–204.

[4] J. Derrick and E.A. Boiten, "Separating Component and Context Specification Using Promotion", in K. Araki, A. Galloway, and K. Taguchi, eds., *International Conference on Integrated Formal Methods*, pp. 293–312, Springer, July 1999.

[5] J. Derrick and E.A. Boiten, "Non-atomic Refinement in Z", in J.M. Wing, J.C.P. Woodcock, and J. Davies, eds., *FM'99 World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pp. 1477–1496, Berlin, Springer, 1999.

[6] J. Derrick, E.A. Boiten, H. Bowman, and M.W.A. Steen, "Specifying and Refining Internal Operations in Z", *Formal Aspects of Computing*, 10:125–159, December 1998.

[7] J. Derrick, E.A. Boiten, H. Bowman, and M.W.A. Steen, "Viewpoints and Consistency — Translating LOTOS to Object-Z", *Computer Standards and Interfaces*, pp. 251–272, December 1999.

[8] European Organisation for the Safety of Air Navigation, *ECHO Final Report*, 1.0 edition, December 1997.

[9] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development", *Int. Jour. on Software Engineering and Knowledge Engineering*, 1992, 2(1), pp. 31–58.

[10] P.F. Linington, "RM-ODP: The Architecture", *ICODP*, eds. K. Raymond and L. Armstrong, *Open Distributed Processing: Experiences with Distributed Environments*, Chapman and Hall, February 1995, Brisbane, Australia, pp. 15–33.

[11] G. Smith, *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.

[12] J.M. Spivey, *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall, 1992.

[13] M.W.A. Steen and J. Derrick, "ODP Enterprise Viewpoint Specification", *Computer Standards and Interfaces*, 22:165–189, September 2000.

[14] C. Taylor, J. Derrick, and E.A. Boiten, "A Case Study in Partial Specification: Consistency and Refinement for Object-Z", in *Proc. of ICFEM 2000*, pp. 177–185. IEEE, September 2000.