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
E.A.Boiten@ukc.ac.uk

Abstract Open Distributed Processing (ODP) is a viewpoints based ISO framework for specifying open distributed systems. This paper considers an application of ODP to the specification of an air traffic control (ATC) system. The key issues that arise from this are discussed further in the context of the formal specification of a simpler model — the Information Viewpoint in Z, and the Computational Viewpoint in Object-Z.

Keywords: ODP, viewpoints, correspondences, air traffic control, Z, Object-Z

1. Introduction

The ODP (Open Distributed Processing) framework (Linington, 1995) is a general architecture for open distributed systems, proposed by the International Standards Organisation. The Reference Model (RM-ODP) identifies five *viewpoints* (Finkelstein et al., 1992) for system specification, each providing a different, partial perspective on the overall system: *Enterprise*: the scope, purpose, and policies of the system, a "community" of "actors" serving an overall objective; *Information*: the information involved in the system, and how it is processed, without describing the distributed architecture; *Computation*: a functional decomposition of the system into objects that interact via interfaces; *Engineering*: the mechanisms and functions needed to support interaction between the distributed objects; and *Technology*: the concrete technological infrastructure, in terms of the particular hardware and software components involved, and their connections and relations.

The viewpoints are connected using *correspondences*. These are necessary to indicate the relations (equality, overlap, decomposition) between elements of the viewpoints. Some of these correspondences are pre-defined by RM-ODP, but most will need to be provided as part of the overall system specifica-

tion. The use of identical names may provide an elementary level of implicit correspondence.

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 defined in natural language, and so are inevitably somewhat open to differing interpretations.

An interesting example of a large-scale application of ODP is Eurocontrol's ECHO study (European Organisation for the Safety of Air Navigation, 1997), a specification of a particular ATC system. Consideration of this study raises some general issues regarding the interpretation and use of ODP, which we discuss here. These include, for example: (1) what the scope and nature of the Information Viewpoint should be; (2) how a hierarchical division into subsystems can be integrated with a division according to viewpoints ; (3) how structure in terms of *instances* of distributed objects should be specified in the Computational Viewpoint; and (4) what kinds of relations there should be between the viewpoints.

The remainder of this paper is organised as follows. Section 2 describes how ODP is used and interpreted in the ECHO study, and Section 3 lists a number of key issues arising from this study. In the light of these issues, Section 4 presents a simplified model of an ATC system, in which the formal specification languages Z and Object-Z are used to represent the Information and Computational Viewpoints respectively. In conclusion, Section 5 discusses the roles of the viewpoint correspondences and other relations between the viewpoints in both models.

2. ODP in the ECHO Study

The ECHO study provides specifications from three viewpoints — the Enterprise, Computational, and Information Viewpoints.

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: (1) *Regulation*: 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; (2) *Operations*: to provide the appropriate level of Air Traffic Service to airspace users; and (3) *Support*: to supply services necessary for the Operations community to operate. Navigation and Surveillance are subcommunities of Operations, specified separately in the Computational Viewpoint but having a joint Information Viewpoint specification. Control is a subcommunity of Support which is specified from both those viewpoints; a further 11 subcommunities are not considered in detail.

Apart from the specification of communities and their decompositions, the Enterprise Viewpoint is enhanced by UML diagrams indicating responsibilities of and relations between the actors. Names of Enterprise "actors" reoccur as names of classes in the Computational Viewpoint.

Information Viewpoint. The two Information Viewpoint specifications (jointly for the Navigation and Surveillance subsystems, and for the Control subsystem) consist of annotated UML diagrams. No operations are specified for this viewpoint, which has been interpreted as a viewpoint for specifying data types used by the system. From this perspective, the combination of two subsystems into one Information Viewpoint specification indicates a large overlap in the kinds of data that the subsystems use, without implying they are a single subsystem.

Computational Viewpoint. The Computational Viewpoint specifications (for Surveillance and Control) are expressed using annotated UML class diagrams. The attribute types are taken from the Information Viewpoint. Many of the Computational Viewpoint classnames correspond to entities which would have substantial distributed applications at the implementation level — e.g., databases. For some methods, inputs and outputs are given, together with an "Offers" statement which describes the method's function, and a "Uses" statement, listing the other methods it invokes. In addition, use case scenarios are used to illustrate behaviour associated with individual objects.

However, the specifications do not describe the (initial) configurations of the subsystems in terms of *instances* of the classes described. This kind of information would be especially useful in situations where a particular class occurs in multiple subsystems. For example, the Control community and the Surveillance community both include the Aircraft and ATSU (Air Traffic Service Unit) classes. However, it is not clear whether the subsystems actually have some *instances* of these Computational classes in common. This impacts on the definition of interfaces between the subsystems, and the correspondences between the Computational Viewpoint and other viewpoints.

3. Issues Arising from the ECHO Study

The following issues stand out from the ECHO study:

- *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. It appears these are two orthogonal dimensions, but how is their combination achieved in practice? Should each subsystem be described from all viewpoints?

- *Subsystems through the Enterprise Viewpoint.* In the ECHO case, there is a correlation between major subsystems and the Enterprise Viewpoint description: Enterprise "communities" correspond to separate subsystems. Is this sufficient information to impose the same structuring on the other viewpoints?
- *Subsystem interaction.* In an ODP specification, how should the connections between major subsystems be described, and how does this relate to the viewpoints and correspondences? The Computational Viewpoint would seem the most appropriate one for dealing with this, as it is concerned with interfaces and interactions. However, some of its classes in the ECHO case do not represent or form part of a subsystem identified in the Enterprise Viewpoint.
- *Object instances versus classes.* In the ECHO study's detailed Computational Viewpoint specifications *classes* of distributed objects are defined, but the structure of each subsystem in terms of *instances* of those classes is not stated explicitly. How should this configuration information be added?

4. A Partially Formalised ODP Specification

This section discusses a simple formalised model of an ATC system, which indicates one way of resolving some of the questions above. The formal languages Z (Spivey, 1992) and Object-Z (Smith, 2000) are used to specify the Information and Computational Viewpoints, respectively. The choice of specification notations here is a pragmatic one: they allow us to give a precise formalisation of the aspects we feel to be of relevance in the case study. Other specification languages may be equally suitable for this. 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. In presenting this specification, we give (yet) another interpretation of RM-ODP, strongly inspired by the semi-formal interpretation used in the ECHO case study. The specification is given in full in (Taylor et al., 2001), the presentation here omits many trivial details and simple definitions.

4.1. Enterprise Viewpoint

According to RM-ODP, 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. There are various ways in which a hierarchical division into subsystems could 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 classes

of distributed objects which are subsequently used in the Computational Viewpoint.

In our simplified ATC specification, the Enterprise Viewpoint consists of an informal statement of the hierarchy of subsystems involved, and of their objectives. These objectives will not be formalised at this level; see (Steen and Derrick, 2000) for an approach to formalising policies. Some of these subsystems could be thought of as being both "actors" and "communities," i.e. as being composed of smaller subcommunities. 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.

As in the ECHO case study, our system 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 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 the same entity. The Enterprise Viewpoint is also used to list types of actor other than the subsystems themselves — e.g. "radars", "controllers", etc. — that are used as classes in the Computational Viewpoint.

4.2. Information Viewpoint

In the ECHO study, the Information Viewpoint acts as a "data dictionary". Using a design-oriented notation like UML, this is the obvious way of specifying the Information Viewpoint. However, if we use a *formal specification* approach to ODP, the information present in the system can be characterised more abstractly. One aspect of this may be *not* to use objects, as this strongly suggests a decomposition of the information which is irrelevant at this point. Another aspect is that the emphasis shifts from the actual data representation of the system's data types to their interfaces and abstract behaviours — and thus, *operations* are more naturally included. Of course, such operations can give an initial indication of the top-level operations required in the Computational Viewpoint. However, the relation between the Information and Computational Viewpoints need not be a refinement relation: the system may still be a common refinement of *both*.

For these reasons, we give a specification which does include operations below. The operations defined require further refinement, to impose additional constraints, and further operations need to be added.

Coordinates, points, and paths. A "4D point" consists of a latitude, longitude, flight level, and time.

$$Pt_{4D} == Lat \times Long \times FlightLevel \times Time$$

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.

$Path_{4D}$

$$atype : AircraftType;\ pts : \operatorname{seq} Pt_{4D}$$

$$pts\ \underline{possibleFor}\ atype$$

$$startTime : Path_{4D} \pfun Time$$

$$\operatorname{dom} startTime = \{p : Path_{4D} \mid p.pts \neq \langle\rangle\}$$
$$\forall p : \operatorname{dom} startTime \bullet startTime(p) = time(head(p.pts))$$

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, as 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; and (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 ATC and the path most recently requested by the aircraft. The assigned and actual paths are required to have the same start time.

$Flight$

$$atype : AircraftType$$
$$actualPath_{4D}, assignedPath_{4D}, requestedPath_{4D} : Path_{4D}$$

$$atype = actualPath_{4D}.atype$$
$$= assignedPath_{4D}.atype = requestedPath_{4D}.atype$$
$$startTime(assignedPath_{4D}) = startTime(actualPath_{4D})$$

Stronger correlations between these three paths as system invariants follow indirectly from the available operations.

The four types of conflict involving flights are defined by assuming three relations of conflict between 4D paths. "AA conflict" (aircraft–aircraft conflict) is defined as involving two (actual or assigned) paths belonging to different flights.

$$| \; inAAConflict, inDevConflict, inReqConflict : Path_{4D} \leftrightarrow Path_{4D}$$

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

A "resource conflict" relation is declared between 4D paths and "environmental objects" (which are either airspaces or physical objects, such as mountains).

$$| \; ResConflict : Flight \leftrightarrow EnvObj$$

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.

Sys

$$\begin{aligned}
&sysTime : Time;\ envObjs : \mathbb{F}\, EnvObj \\
&flightIndex : \text{iseq}\, Flight \\
&currentFlights, allFlightsInConflict : \mathbb{F}\, Flight \\
&aaConflicts : \mathbb{F}(Flight \times Flight) \\
&devConflicts, reqConflicts : \mathbb{F}\, Flight \\
&resConflicts : \mathbb{F}(Flight \times EnvObj)
\end{aligned}$$

$$\begin{aligned}
&currentFlights \subseteq \text{ran}\, flightIndex \\
&aaConflicts = AAConflict \cap (currentFlights \times currentFlights) \\
&devConflicts = DevConflict \cap currentFlights \\
&reqConflicts = ReqConflict \cap currentFlights \\
&resConflicts = ResConflict \cap (currentFlights \times envObjs) \\
&allFlightsInConflict = \\
&\quad devConflicts \cup reqConflicts \cup \text{dom}\, aaConflicts \\
&\quad \cup \text{dom}\, resConflicts
\end{aligned}$$

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. Initially, there are no current flights and thus no conflicts.

SysINIT

$$Sys'$$

$$sysTime' = 0 \land flightIndex' = \langle\rangle$$

Operations. The operations defined are: *AssignFlight*, *TakeOff*, *Landing*, *FlightObs*, and *ResolveConflict*. As all of the operations leave the environmental objects unchanged and increase the system time, we use an auxiliary definition:

SysOp

$$\Delta Sys$$

$$envObjs' = envObj \land sysTime' = sysTime + 1$$

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 later than the current system time. 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.

AssignFlight

$$SysOp;\ f! : Flight;\ t! : Time$$

$$startTime(f!.assignedPath_{4D}) = t! \land t! > sysTime$$
$$flightIndex' = flightIndex \frown \langle f! \rangle$$
$$currentFlights' = currentFlights \cup \{f!\}$$
$$f!.actualPath_{4D}.pts = \langle\rangle$$
$$allFlightsInConflict' = allFlightsInConflict$$

The inputs of the *TakeOff* operation are a current flight's index $n?$ and the corresponding flight $f?$, which must not be in conflict. The output of the operation is a flight $f!$, identical to $f?$ except for its actual 4D path now containing the first point in its assigned path.

TakeOff

$SysOp$; $n? : \mathbb{N}$; $f?, f! : Flight$

$n? \in \mathrm{dom}\, flightIndex \land flightIndex(n?) = f?$
$f? \in currentFlights \land f? \notin allFlightsInConflict$
$startTime(f?.assignedPath_{4D}) = sysTime$
$flightIndex' = flightIndex \oplus \{n? \mapsto f!\}$
$f!.actualPath_{4D}.pts = \langle head\, f?.assignedPath_{4D}.pts \rangle$
`[All other fields of f! equal those of f?]`
$currentFlights' = (currentFlights \setminus \{f?\}) \cup \{f!\}$

Landing removes a flight from the set of current flights. Implicitly, this may also remove it from the flights in conflict. Further conditions might be imposed to model regular landing, relating the destination to the actual path.

Landing

$SysOp$; $f? : Flight$

$f? \in currentFlights \land currentFlights' = currentFlights \setminus \{f?\}$
$flightIndex' = flightIndex$

FlightObs represents the receipt of an observation of an aircraft's position at a particular time (in the form of a 4D point). $f?$ is the input flight, and $f!$ the modified output flight, incorporating the new observation. Due to the invariants in *Flight*, the observation must be possible for the aircraft involved, given its trajectory so far.

FlightObs

$SysOp$; $n? : \mathbb{N}$; $pt? : Pt_{4D}$; $f?, f! : Flight$

$n? \in \mathrm{dom}\, flightIndex \land flightIndex(n?) = f?$
$f? \in currentFlights \land f?.actualPath_{4D}.pts \neq \langle\rangle$
$time(last(f?.actualPath_{4D}.pts)) < time(pt?) < sysTime$
$f!.actualPath_{4D}.pts = f?.actualPath_{4D}.pts \frown \langle pt? \rangle$
`[All other fields of f! equal those of f?]`
$flightIndex' = flightIndex \oplus \{n? \mapsto f!\}$
$currentFlights' = (currentFlights \setminus \{f?\}) \cup \{f!\}$

ResolveConflict, omitted here, represents in a very abstract way the resolution of a "self-contained" conflicting set of flights, in the sense that its members may only be in "AA conflict".

4.3. Computational Viewpoint Specification

The Computational Viewpoint specification is expressed in the object oriented specification language Object-Z (Smith, 2000) An Object-Z specification includes several *class schemas*, of which one (here: *Main*) represents the system being modelled. An attribute declaration x : *ClassName* denotes a *reference* 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, inheriting its inputs and outputs if any. For example, in the specification below, the *Main* class has an attribute *support* of class *Support*, and the expression *support.TakeOff* represents the execution on that object of its own (lower-level) *TakeOff* operation.

Each operation has an optional Δ-list, showing attributes that it allows to change. Object-Z provides several operators for combining operations — including $\wedge$ (schema conjunction), and $\|$ which equates the output variables of one operation with matching input variables of another, and hides both.

The "communities" identified in the Enterprise Viewpoint are equated with instances of classes specified in the Computational Viewpoint. Instances of Object-Z classes may share component objects, thus communities can overlap in terms of the objects involved.

The main focus in this initial Computational Viewpoint specification is to give a broad indication of system structure, in terms of identifiable subsystems, and the objects of which they are composed. The aim is to specify operations 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.

Main system. The overall system is viewed as an instance of the *Main* class. 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 which cannot be directly contained in any other object.) 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*, *TakeOff*, and *ResolveConflict* are promoted from the subsystems.

Main

$control : Control©;\ support : Support©$

$control.flightDbase = support.flightDbase$

$\textit{INIT} \mathrel{\widehat{=}} control.\textit{INIT} \wedge support.\textit{INIT}$
$AssignFlight \mathrel{\widehat{=}} (support.RequestFlight \| control.AssignFlight)$
$TakeOff \mathrel{\widehat{=}} support.TakeOff$
$Landing \mathrel{\widehat{=}} support.Landing$
$FlightObs \mathrel{\widehat{=}} support.FlightObs$
$ResolveConflict \mathrel{\widehat{=}} control.ResolveConflict$

The full specification in (Taylor et al., 2001) refers to *time* on the data level through the use of a *clock* attribute (corresponding to *sysTime* in the Information Viewpoint), which is shared (ensured by predicates) by all object instances. The details of this have been omitted below — references to *clock.time* below are to this global (shared) clock.

Control subsystem. A *Control* object has a subsystem *flightManager*, which executes control functions, and a subsystem *flightDbase*, containing information about current flights. 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 similar synchronisation between the flight database and the flight manager.

Control

$flightManager : FlightManager©$
$flightDbase : FlightDbase$

$\textit{INIT} \mathrel{\widehat{=}} flightManager.\textit{INIT} \wedge flightDbase.\textit{INIT}$
$AssignFlight \mathrel{\widehat{=}} (flightManager.SelectFlight$
$\quad \| flightDbase.AssignFlight)$
$ResolveConflict \mathrel{\widehat{=}} (flightDbase.ResolveConflictSet$
$\quad \| flightManager.ResolveConflict)$

An object of the class *FlightManager* has as its attribute a set of "controllers". The *SelectFlight* and *ResolveConflict* operations represent the selection of a flight and the resolution of a conflict set, respectively, by a particular controller, promoted from operations of the class *Controller*.

For the *Controller* operations, only the signatures are given. The *SelectFlight* operation represents the selection of a flight by a controller. Its input is an air-

craft type, and its outputs are a flight and a take-off time. The *ResolveConflict* operation represents the resolution of a particular set of conflicting flights by an individual controller.

FlightManager

$$controllers : \mathbb{F}_1\ Controller$$

$$\textit{INIT} \mathrel{\widehat=} (\bigwedge c : controllers \bullet c.\textit{INIT})$$
$$SelectFlight \mathrel{\widehat=} [c1 : controllers] \bullet c1.SelectFlight$$
$$ResolveConflict \mathrel{\widehat=} [c1 : controllers] \bullet c1.ResolveConflict$$

Controller

$$SelectFlight \mathrel{\widehat=} [atype? : AircraftType;\ f! : Flight;\ t! : Time]$$
$$ResolveConflict \mathrel{\widehat=} [flights? : \mathbb{F}_1\ Flight;\ oldIndex?, newIndex! : \mathbb{N}_1 \rightarrowtail Flight]$$

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 data types (such as *Flight*) and global definitions of sets, functions, etc., that are already defined in the previously given Information Viewpoint specification.

FlightDbase

$$envObjs : \mathbb{F}\ EnvObj$$
$$flightIndex : \text{iseq}\ Flight;\ currentFlights : \mathbb{F}\ Flight$$
$$\Delta$$
$$allFlightsInConflict, reqConflicts, devConflicts : \mathbb{F}\ Flight$$
$$aaConflicts : \mathbb{F}(Flight \times Flight)$$
$$resConflicts : \mathbb{F}(Flight \times EnvObj)$$

$$allFlightsInConflict \subseteq currentFlights \subseteq \text{ran}\ flightIndex$$
$$aaConflicts = AAConflict \cap (currentFlights \times currentFlights)$$
$$devConflicts = DevConflict \cap currentFlights$$
$$reqConflicts = ReqConflict \cap currentFlights$$
$$resConflicts = ResConflict \cap (currentFlights \times envObjs)$$
$$allFlightsInConflict = devConflicts \cup reqConflicts$$
$$\cup(\text{dom}\ aaConflicts) \cup (\text{dom}\ resConflicts)$$

$$
\begin{array}{l}
\textit{INIT} \mathrel{\widehat=} \textit{flightIndex} = \langle\,\rangle \\
\textit{AssignFlight} \mathrel{\widehat=} [\,\Delta(\textit{flightIndex}, \textit{currentFlights}) \\
\qquad\qquad \textit{atype}? : \textit{AircraftType};\ f? : \textit{Flight};\ t! : \textit{Time}\,] \\
\textit{TakeOff} \mathrel{\widehat=} [\,\Delta(\textit{flightIndex}, \textit{currentFlights});\ \textit{fNo}? : \mathbb{N}_1\,] \\
\textit{Landing} \mathrel{\widehat=} [\,\Delta(\textit{currentFlights});\ \textit{fNo}? : \mathbb{N}_1\,] \\
\textit{FlightObs} \mathrel{\widehat=} [\,\Delta(\textit{flightIndex}, \textit{currentFlights}) \\
\qquad\qquad \textit{fNo}? : \mathbb{N}_1;\ \textit{pt}? : \textit{Pt}_{4D}\,] \\
\textit{ResolveConflict} \mathrel{\widehat=} [\,\Delta(\textit{flightIndex}, \textit{currentFlights}) \\
\qquad\qquad\qquad \textit{newIndex}? : \mathbb{N}_1 \rightarrowtail\!\!\!\!\!\!\rightarrow \textit{Flight}\,]
\end{array}
$$

The operations are given in signature form here, to be refined to implementations of the constraints defined on them in the Information Viewpoint specification. The attributes listed below the Δ symbol are *secondary*, i.e. they can change in any operation without appearing in its *delta*-list.

Support subsystem. An object of the class *Support* consists of a surveillance subsystem and a flight database subsystem. 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 synchronisations 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.

Support

$$
\begin{array}{l}
\textit{surv} : \textit{Surv}_{\textcircled{c}};\ \textit{flightDbase} : \textit{FlightDbase} \\
\hline
\textit{INIT} \mathrel{\widehat=} \textit{surv.INIT} \wedge \textit{flightDbase.INIT} \\
\textit{FlightObs} \mathrel{\widehat=} (\textit{surv.FlightObs} \| \textit{flightDbase.FlightObs}) \\
\textit{Landing} \mathrel{\widehat=} (\textit{surv.Landing} \| \textit{flightDbase.Landing}) \\
\textit{TakeOff} \mathrel{\widehat=} (\textit{surv.TakeOff} \| \textit{flightDbase.TakeOff}) \\
\textit{RequestFlight} \mathrel{\widehat=} \textit{surv.RequestFlight}
\end{array}
$$

An object of the *Surv* (surveillance system) class consists of a set of radar stations. 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

$$radars : \mathbb{F}_1\ Radar$$

$$\textit{INIT} \mathrel{\widehat=} (\bigwedge r : radars \bullet r.\textit{INIT})$$
$$FlightObs \mathrel{\widehat=} [r1 : radars] \bullet r1.FlightObs$$
$$Landing \mathrel{\widehat=} [fNo! : \mathbb{N}_1]$$
$$TakeOff \mathrel{\widehat=} [fNo! : \mathbb{N}_1]$$
$$RequestFlight \mathrel{\widehat=} [atype! : AircraftType]$$

The *Radar* class is where the clock is actually used (rather than just shared with component objects), and so we include it explicitly here. The operation *FlightObs* represents a 4D point observation for a particular flight number (with the time coordinate being the current clock time).

Radar

$$clock : Clock$$

$$\textit{INIT} \mathrel{\widehat=} clock.\textit{INIT}$$
$$FlightObs \mathrel{\widehat=} [fNo! : \mathbb{N}_1;\ pt! : Pt_{4D} \mid time(Pt_{4D}) = clock.time]$$

5. Correspondences and Conclusions

Previous work (Boiten et al., 2000; Bowman et al., 1996) 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 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". 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 (Derrick et al., 1999)). For techniques for refinement and viewpoint unification in Z and Object-Z, see (Boiten et al., 1999; Derrick and Boiten, 2001b).

In order to combine the viewpoints, correspondences relating their elements and possibly other structuring information is necessary. The ECHO study contains a large amount of implicit information about the correspondences, following from the relations between the viewpoints observed in Section 3. In addition, the ECHO study observes that the Enterprise Viewpoint needs to obey constraints imposed by the Technology Viewpoint. In the formalisation, the structuring and correspondence was largely given by the informal text in

the Enterprise Viewpoint. For consistency checking between the viewpoints and other formal checks, further information would be necessary, for example the refinement relation relating each viewpoint to acceptable implementations would have to be specified.

One crucial point from the ECHO case study is that there is a need to interpret RM-ODP in a way which allows hierarchical structure to be specified and subsystems to be identified. The Enterprise Viewpoint may have a central rôle in this, for example by providing a starting point for the Computational Viewpoint (and others) by identifying functional subsystems and their objectives, without specifying the concrete objects used to implement them, or the extent to which the subsystems overlap in terms of objects. For the specification of the subsystems in the Information and Computational Viewpoints, this informal decomposition appeared both useful and sufficient. Apart from a subsystem decomposition, the Enterprise Viewpoint should also provide *policies*. (Steen and Derrick, 2000) addresses the specification of such policies and how these could be checked with the other viewpoints' requirements.

The issue of classes vs. instances, i.e. the configurations of the various viewpoints and their subsystems, was adequately addressed by the common Object-Z practice of introducing of a *Main* class. By the use of object containment (© symbol), Object-Z can also specify that various instances of the same class must be different.

A further analysis of the viewpoints and correspondences in the ECHO study may be found in (Derrick and Boiten, 2001a).

Acknowledgment This work was partially sponsored by the EPSRC grant "ODP Viewpoints in a Development Framework".

References

Boiten, E., Bowman, H., Derrick, J., Linington, P., and Steen, M. (2000). Viewpoint consistency in ODP. *Computer Networks*, 34(3):503–537.

Boiten, E., Derrick, J., Bowman, H., and Steen, M. (1999). Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75.

Bowman, H., Boiten, E., Derrick, J., and Steen, M. (1996). Viewpoint consistency in ODP, a general interpretation. In Najm, E. and Stefani, J.-B., editors, *First IFIP International workshop on Formal Methods for Open Object-based Distributed Systems*, pages 189–204, Paris. Chapman & Hall.

Derrick, J. and Boiten, E. (2001a). Applying ODP to an air traffic management system: viewpoints and correspondences. Submitted for publication.

Derrick, J. and Boiten, E. (2001b). *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer-Verlag.

Derrick, J., Boiten, E., Bowman, H., and Steen, M. (1999). Viewpoints and consistency: translating LOTOS to Object-Z. *Computer Standards and Interfaces*, 21:251–272.

European Organisation for the Safety of Air Navigation (1997). ECHO final report, 1.0 edition.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. (1992). Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal on Software Engineering and Knowledge Engineering, Special issue on Trends and Research Directions in Software Engineering Environments*, 2(1):31–58.

Linington, P. F. (1995). RM-ODP The Architecture. In Raymond, K. and Armstrong, L., editors, *IFIP TC6 International Conference on Open Distributed Processing*, pages 15–33, Brisbane, Australia. Chapman and Hall.

Smith, G. (2000). *The Object-Z Specification Language*. Kluwer Academic Publishers.

Spivey, J. M. (1992). *The Z notation: A reference manual*. Prentice Hall, 2nd edition.

Steen, M. and Derrick, J. (2000). ODP Enterprise Viewpoint Specification. *Computer Standards and Interfaces*, 22:165–189.

Taylor, C., Boiten, E., and Derrick, J. (2001). Interpreting ODP viewpoint specification. Technical Report 9-01, Computing Laboratory, University of Kent at Canterbury.