Richard O. Sinnott and Kenneth J. Turner. Specifying Multimedia Binding Objects in Z. Proc. Workshop on Trends in Distributed Systems, Aachen, Germany, October 1996.

Specifying Multimedia Binding Objects in Z

Richard O. Sinnott and Kenneth J. Turner,
Department of Computing Science,
University of Stirling,
Stirling FK9 4LA,
Scotland.
email: ros || kjt@cs.stir.ac.uk

July 12, 1996

Abstract

The current standardisation activity of Open Distributed Processing (ODP) has attempted to incorporate multimedia flows of information into its architecture through the idea of stream interfaces. At present the reference model of ODP (ODP-RM) abstracts from the precise nature of the flows of information. As a consequence of this, the ODP-RM only deals with syntactic aspects of stream interfaces and does not require them to satisfy any behavioural considerations. It is shown in this paper how the formal notation Z can be used to reason about these flows of information in a manner that enables behavioural as well as temporal aspects to be considered. The example given to highlight the approach is the ODP concept of a binding object.

Keywords: Z; Open Distributed Processing; Architectural Semantics; Temporal Logic; Object-Orientation.

1 Introduction

The ODP-RM is a framework that is being developed to enable standards for distributed systems to be developed in a uniform, consistent and expedient fashion. It is based upon concepts derived from current distributed processing developments and, as far as possible, on the use of formal description techniques to specify the architecture. The ODP-RM itself is based on an *extended* classical object-oriented model where objects are encapsulated entities that interact at interfaces. The term extended is used here to note that as well as the typical object-oriented (RPC-like) paradigm of message passing, ODP also attempts to incorporate more complex message passing phenomenon, namely: (multimedia) flows of information.

It is shown in this paper how the formal notation Z can be used to reason about these flows of information in a manner that enables behavioural as well as temporal aspects to be considered. The management and control of these flows of information is also considered. The example given to highlight the approach is the ODP concept of a binding object.

The rest of the paper is structured as follows. Section 2 gives a brief overview of ODP and the ODP-RM. Section 3 looks in more detail at the computational viewpoint language of ODP. Section 4 considers aspects concerned with the formalisation of the computational viewpoint language, and in particular at those concepts related to binding of multimedia streams.

Section 5 looks at the formalisation of stream and control computational interfaces. Section 6 looks at binding as given by ODP; highlights it weaknesses and proposes extensions to it. Section 7 considers binding objects and the relation between control and stream interfaces. Finally section 8 draws some conclusions on the work and gives some acknowledgements.

2 The Reference Model of ODP

The ODP-RM consists of four main parts (documents). Part 1 (ISO/IEC 1995*a*) contains an overview and guide to use of the ODP-RM. Part 2 (ISO/IEC 1995*b*) contains the definition of concepts and gives the framework for description of distributed systems. It also introduces the principles of conformance and the way they may be applied to ODP. In effect Part 2 provides the vocabulary with which distributed systems may be described, reasoned about and developed, *i.e.* it is used as the basis for understanding the concepts contained within Part 3 of the ODP-RM. Part 3 (ISO/IEC 1995*c*) contains the specification of the required characteristics that qualify distributed system as open, *i.e.* constraints to which ODP systems must conform. The main features of Part 3 include the viewpoint languages, conformance issues, functions and transparencies. It is the viewpoint languages that are of concern in this paper, in particular the computational viewpoint. This viewpoint focuses primarily on the functional decomposition of a given ODP system, and on the interworking and portability of ODP functions, *i.e.* it is here that objects interworking with one another are considered in detail. Finally, Part 4 (ISO/IEC 1995*d*, ISO/IEC 1995*e*) of the ODP-RM contains a formalisation of a subset of the ODP concepts. This formalisation is achieved through "interpreting" each concept in terms of the constructs of a given formal specification language. This work is concerned with ensuring that the reference model for ODP is consistent with itself. It brings formal expression to the semi-formal concepts, *i.e.* concepts written in formal English, contained within the reference model. It achieves this through interpreting the different concepts in various formal languages. Presently, LOTOS (ISO/IEC 1989*b*), Z (Spivey 1992), ESTELLE (ISO/IEC 1989*a*) and SDL'92 (ITU-T 1992) are under consideration. The aim is that it will not be possible to produce incompatible ODP specifications, as was the case with OSI; see (Turner 1996).

3 Overview of the Computational Viewpoint Language

The computational viewpoint language is used to consider issues of distribution. Since the approach taken in ODP-RM is an object-oriented one. That is, objects and their interfaces [1] are the fundamental components with which systems are reasoned about. The computational viewpoint language focuses on the interfaces that computational objects should support. It provides rules that enable interfaces to be structured correctly, thus permitting meaningful interactions to take place between objects. The advantages of object-oriented approaches with regard to distributed systems development are discussed in (Blair & Lea 1993).

Since the ODP-RM is a framework for developing standards, it is not possible to be overly prescriptive with regard to the behaviour of any given object. Rather, objects will in general have different behaviours depending upon the application they are used for. As a result, the ODP-RM addresses only the syntactic aspects associated with the interfaces to given objects,

[1] ODP objects may have more than one interface, unlike objects defined in other object models, *e.g.* OMG's CORBA object model.

i.e. their interface signatures. Various naming rules for operations [2] and parameters found in interfaces are given that interfaces must adhere to. Following this type checking is done — inadequately! — based on interfaces having these syntactic structurings. The consequences of this are that all messages passed to objects will at least have an understood format.

The ODP-RM prescribes three particular types of interface: operational; stream and signal. Operational interfaces contain either announcements or interrogations, and are used to represent object interactions as represented by most message passing object models. Announcements are used to represent send-only interactions, whilst interrogations are used to represent send and receive interactions. Signal interfaces consist of signals, where a signal may be regarded as the most basic unit of interaction in the computational viewpoint. They may be considered as single, atomic actions between computational objects. Stream interfaces contain multimedia flows of information. The syntactic formalisation in Z of these three types of interface has been shown previously in (Sinnott & Turner 1996). In this paper we focus in more detail on stream interfaces.

Paramount to the successful interworking of computational objects is the ODP concept of a binding object. A binding in ODP may loosely be regarded as the composition (synchronisation) of two or more interfaces. A binding object is responsible for, amongst other things, ensuring that a certain level of quality of service is maintained between the interacting objects. The following diagram, figure 1, shows a simplistic system incorporating a binding object between a producer and consumer of multimedia streams.

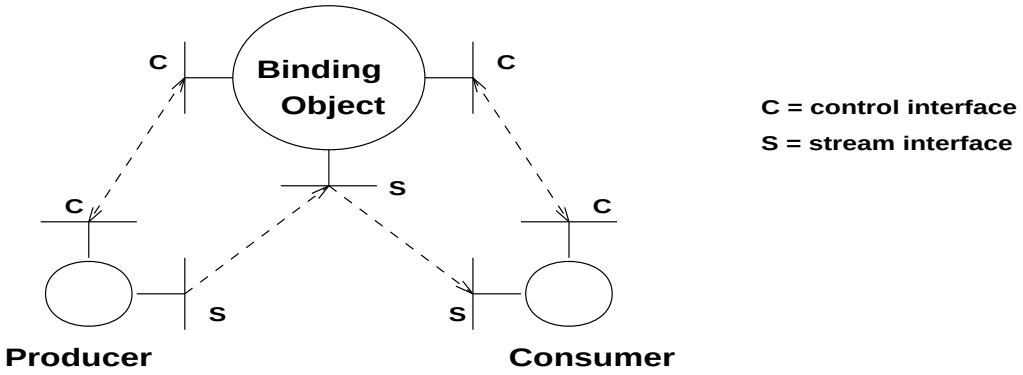

Figure 1: Simplistic Multimedia Binding Object Configuration

The ODP-RM does not prescribe the format of control interfaces, however, it is likely that they will take the form of operational interfaces.

Stream interfaces have associated signatures consisting of flow signatures, where a flow signature contains a name for the flow, the type of the flow and an indication of the causality of the flow. All flow names in a given stream interface must be unique within that interface. The causality of a flow can be either *Producer* or *Consumer*. The issue of typing of flows was abstracted from in (Sinnott & Turner 1996). That is, it was simply represented as a basic type in Z. Whilst this enabled a formalisation to be given, that satisfied the rules given in (ISO/IEC 1995*c*), it was not a very satisfactory solution. That is, typing of interfaces should ideally be more than simple syntax checking. This is especially so with multimedia flows of information where behavioural and temporal considerations are critical to meaningful interactions.

[2] and signals and streams.

4 Formalising Concepts Associated with Binding Objects

To formalise a binding object, it is necessary to consider different aspects of that object identified by ODP. For the purpose of this paper, we shall look at stream and control interfaces. These have associated with them: an interface signature; a behaviour specification and an environment contract. We shall look at each of these in turn.

4.1 Formalising Syntactic Aspects of Multimedia Streams

The ODP-RM abstracts away from the contents of flows of information. As a result, it says very little about binding multimedia streams together and how they might be type checked. For the purpose of this paper, we shall consider a generic idea of information flow. Here the flow of information is represented by a sequence of **frames** where a frame may be regarded as a particular item in the flow of information. Each frame can be considered as a unit consisting of data (this will normally be compressed) which we represent by ($Data$) and a time stamp used for modelling the time at which this particular frame was sent or received. It is also often the case in multimedia flows, that particular frames are required for synchronisation, *e.g.* synchronisation of audio with video for example. Therefore we associate a particular label ($Label$) with each frame. This can then be used for selecting a particular frame from the flow as required. From this, we may model a frame as:

Frame

$$data : Data$$
$$timestamp : \mathbb{N}$$
$$label : Label$$

It should be noted here that we model time as a natural number as done by (Delisle & Garlan 1989). It might well be the case that real (dense) time could be used as done by (King 1989), or time intervals (Coombes 1990). For simplicity here though, we restrict ourselves to discrete time, *i.e.* represented as a natural number.

Information flows have inherent characteristics given by the nature of their flows. For example, flows can be *isochronous* which implies that each frame is sent/received in equal time segments. Alternatively, flows can be *bursty* in nature which implies that the time intervals between successive frames is not necessarily equal.

We may thus represent a flow characteristic as:

$$FlowCharacteristic ::= Isochronous\langle\langle\mathbb{N}\rangle\rangle \mid Bursty$$

As stated, interfaces in the computational viewpoint have causalities associated with them, where a causality may loosely be regarded as some notion of expected behaviour. These causalities can be associated with the interface as a whole (operational) or with the individual actions associated with the interfaces (stream). The different causalities may be represented by:

$$Causality ::= Producer \mid Consumer \mid Client \mid Server$$

From this, we may represent a generic multimedia flow as:

$$
\begin{array}{|l}
\textit{muFlowType} \\
\hline
\textit{frames} : \operatorname{seq} \textit{Frame} \\
\textit{flowChar} : \textit{FlowCharacteristic} \\
\textit{rate} : \mathbb{N} \\
\hline
\forall f_1, f_2 : \textit{Frame} \mid \langle f_1, f_2 \rangle \text{ in } \textit{frames} \bullet f_2.\textit{timestamp} > f_1.\textit{timestamp} \land \\
\textit{flowChar} = \textit{Isochronous}(\textit{rate}) \Rightarrow \\
\quad (\forall f_1, f_2 : \textit{Frame} \mid \langle f_1, f_2 \rangle \text{ in } \textit{frames} \bullet f_2.\textit{timestamp} - f_1.\textit{timestamp} = \textit{rate}) \\
\textit{flowChar} = \textit{Bursty} \Rightarrow \\
\quad (\exists f_1, f_2, f_3 : \textit{Frame} \mid \langle f_1, f_2 \rangle \text{ in } \textit{frames} \land \langle f_2, f_3 \rangle \text{ in } \textit{frames} \bullet \\
\qquad f_2.\textit{timestamp} - f_1.\textit{timestamp} \neq f_3.\textit{timestamp} - f_2.\textit{timestamp}\,) \\
\hline
\end{array}
$$

This states that a multimedia flow type is given by a sequence of frames with some flow characteristic and temporal ordering. All frames in the sequence have time stamps in ascending order. Isochronous flows have frames separated by equal time intervals, whereas bursty flows may have frames separated by non-equal time intervals.

To formalise stream interfaces, it is necessary to introduce names (*Name*) for things, *e.g.* names of action templates for flows. A flow signature represented as a name, flow type and causality may thus be represented by:

$$
\begin{array}{|l}
\textit{flowSig} \\
\hline
\textit{fName} : \textit{Name} \\
\textit{fType} : \textit{muFlowType} \\
\textit{role} : \textit{Causality} \\
\hline
\textit{role} \in \{\textit{Producer}, \textit{Consumer}\} \\
\hline
\end{array}
$$

Stream interfaces consist of sets of flow signatures. Each flow signature name in a given stream interface signature is required to be uniquely identified. This can be represented as:

$$
\begin{array}{|l}
\textit{strIntSig} \\
\hline
\textit{flows} : \mathbb{P}\, \textit{flowSig} \\
\hline
\forall \textit{fs}_1, \textit{fs}_2 : \textit{flowSig} \bullet \textit{fs}_1 \in \textit{flows} \land \textit{fs}_2 \in \textit{flows} \land \textit{fs}_1 \neq \textit{fs}_2 \Rightarrow \textit{fs}_1.\textit{fName} \neq \textit{fs}_2.\textit{fName} \\
\hline
\end{array}
$$

This schema describes the syntactic structure of stream interface signatures satisfying the rules given in the ODP-RM. However, it does not prescribe any particular behaviour. Before we look into issues of behaviour and how it affects binding objects, it is necessary to consider control interfaces.

4.2 Formalising Syntactic Aspects of Control Interfaces

Control interfaces are used to manage the flow of information in a given system. ODP does not prescribe the form of control interfaces, however, it is very likely that they will be based on operations as opposed to signals or streams. Further, for the sake of simplicity here, we shall only consider control interfaces based upon announcements. Operational interfaces including interrogations were considered in more detail in (Sinnott & Turner 1996).

Announcements may have parameters associated with them. These parameters may be represented by a name and a type (*TypeIdentifier*) used to to represent all types in the system.

It should always be possible to determine the type of a parameter in a given system. Thus *param* is introduced as an injective function from names to types.

$$param : Name \rightarrowtail TypeIdentifier$$

It is also useful to introduce sequences of these parameters.

$$paramList == \operatorname{seq} param$$

Now announcements consist of a single invocation action, where an invocation action consists of a name for the invocation and the number, name and type of the argument parameters associated with the invocation. An announcement may thus be represented by the following schema:

annSig

$$invName : Name$$
$$inArgs : paramList$$

Operational interface signatures consist of sets of announcements and interrogations [3], and the interface as a whole is given a causality: client or server. Naming considerations of the components of the interface are also required. That is, all announcement names in the interface are required to be unique, as are the parameters that are associated with them. This can be represented as:

ctrIntSig

$$anns : \mathbb{P}\, annSig$$
$$role : Causality$$

$$role \in \{Client, Server\}$$
$$\forall\, as_1, as_2 : annSig;\ p_1, p_2 : param \bullet$$
$$(as_1 \in anns \land as_2 \in anns \land as_1 \neq as_2 \Rightarrow as_1.invName \neq as_2.invName) \land$$
$$(as_1 \in anns \land \langle p_1 \rangle \text{ in } as_1.inArgs \land \langle p_2 \rangle \text{ in } as_1.inArgs) \land p_1 \neq p_2 \Rightarrow first\, p_1 \neq first\, p_2$$

This schema describes the syntactic structure of control interfaces satisfying the rules given in the ODP-RM. However, it does not prescribe any particular behaviour. As stated, this is necessarily so since objects will have their own different behaviours generally. It can be said generally however, that a behaviour specification consists of a (possibly infinite) set of distinct [4] actions with constraints on their occurrence. These constraints impose a partial ordering on the set of actions. The actions themselves can be internal (*Internal*) to the object or observable to the environment, *i.e.* require participation (synchronisation) with the environment to occur.

[3] but for simplicity, we restrict ourselves to announcements only when modelling control interfaces.

[4] If the actions in a behaviour specification were not distinct then the actual actions associated with an object or interface could be represented by a bag in Z to overcome problems of multiplicity, *e.g.* in recursive behaviour.

4.3 An Elementary Notion of Behaviour

In order to consider issues of behaviour in the computational viewpoint, it is necessary to introduce some functions that map action signatures to actions, *i.e.* flow signatures to flow actions etc. The actions under consideration in this paper are internal actions, flow actions and announcements. This can be represented by a parameterised free type definition as:

$$action ::= isIntAction\langle\langle Internal\rangle\rangle \mid isAnnAction\langle\langle annSig\rangle\rangle \mid isFlowAction\langle\langle flowSig\rangle\rangle$$

A behaviour specification as a collection of actions with an ordering relation between them may thus be represented by:

$$behspec == \{ar : action \leftrightarrow action \mid ar = ar\ ^* \land ar \cap ar^\sim = \mathrm{id}\ action\}$$

Here a set of relations between actions is being built. These relations are partial orders. That is, the expression $ar_1 = ar_1^*$ states that the relation is equal to its reflexive-transitive closure, which is the same as saying that it is reflexive and transitive. The expression $ar_1 \cap ar_1^\sim = \mathrm{id}\ action$ ensures that the relation is anti-symmetric, *i.e.* no two different actions in the relation are related by the inverse of the relation also. Thus the relation ar is a relation that is transitive, anti-symmetric and reflexive, *i.e.* a partial order.

4.4 Formalising Aspects of Environment Contracts

Computational interface templates may have environment contracts associated with them. These may be regarded as agreements on the behaviour between the interface and its environment. They may include quality of service constraints such as throughput, delay and jitter, as well as usage and management constraints. We may represent several different types of constraint in Z but restrict ourselves to certain quality of service constraints for simplicity, in particular: throughput and maximum delay.

Throughput may be regarded as the number of frames that a producer of a flow can produce, or the number of frames that a consumer can consume. Since we treat time as a natural number, we deal with number of frames per second. Isochronous flows should have a consistent throughput, whereas bursty flows may have situations where more frames are output (or input) than at other times. In bursty flows it is especially useful to put an upper limit on the maximum throughput of data. Thus throughput may be represented as:

$$maxThru : muFlowType \to \mathbb{N}$$

$$\begin{aligned}
&\forall\, mft : muFlowType \bullet \\
&\quad mft.flowChar = Isochronous(mft.rate) \Rightarrow maxThru(mft) = 1 \operatorname{div} mft.rate \land \\
&\quad mft.flowChar = Bursty \Rightarrow maxThru(mft) = \\
&\qquad max\,\{s : \mathrm{seq}\, Frame;\ f_1, f_2 : Frame \mid s \ \mathsf{in}\ mft.frames \land \\
&\qquad\quad f_1 = head\ s \land f_2 = last\ s \land f_2.timestamp - f_1.timestamp \leq 1 \bullet \#s\}
\end{aligned}$$

Here the throughput of isochronous flows is simply represented by the reciprocal of the rate, *i.e.* if the time difference between successive frames was 0.1 seconds then the throughput would be 10. Establishing the throughput of bursty flows is a little more involved however. Here the maximum throughput of a bursty flow is obtained by calculating the maximum subsequence of the flow with a timestamp difference of less than or equal to one second from its first and last elements.

The maximum delay of a multimedia flow may be regarded as the upper limit on the time window at which a frame is expected. For example, a consumer may be able to wait for a certain time for the next frame to arrive. Without considering issues such as buffering [5], this can be represented as:

$$
\begin{array}{|l}
maxDelay : muFlowType \to \mathbb{N} \\
\hline
\forall\, mft : muFlowType \bullet \\
\quad mft.flowChar = Isochronous(mft.rate) \Rightarrow maxDelay(mft) = mft.rate \land \\
\quad mft.flowChar = Bursty \Rightarrow maxDelay(mft) = \\
\qquad\qquad min\,\{f_1, f_2 : Frame \mid \langle f_1, f_2\rangle \text{ in } mft.frames \bullet f_2.timestamp - f_1.timestamp\}
\end{array}
$$

For isochronous flows, the maximum delay that a consumer can tolerate without buffering is given by the time difference between two frames. That is, if after this time period the frame is not received, then an error has occurred and some remedying action must be taken, *e.g.* show last frame again. For bursty flows, the maximum delay is given by the minimum time difference between two successive frames in the sequence, *i.e.* if the consumer can consume as fast as the producer can produce, then consumption should be at least as fast as production.

We may represent these constraints generally as:

$$Constraints ::= thruPut\langle\langle maxThru\rangle\rangle \mid delay\langle\langle maxDelay\rangle\rangle$$

Thus from this, environment contracts may be represented as:

$$
\begin{array}{|l}
\hline
EnvCon \\
qosCons : \mathbb{F}\ Constraints \\
\hline
\end{array}
$$

5 Computational Interfaces in ODP

Computational interfaces in ODP have associated with them, an interface signature; a behaviour specification and an environment contract. The interfaces under consideration in this paper are control and stream interfaces. We may thus represent control interface templates as:

$$
\begin{array}{|l}
\hline
ctrIntTemp \\
\hline
ctrIntSigs : \mathbb{F}_1\ ctrIntSig \\
ctrlIntBS : behspec \\
ctrlEnvCon : EnvCon \\
\hline
\forall\, cis : ctrIntSig \mid cis \in ctrIntSigs \bullet \\
\quad (\textbf{let}\ annActs == \{ans : annSig \mid ans \in cis.anns \bullet isAnnAction(ans)\} \bullet \\
\quad (\textbf{let}\ internalActs == \{ia : Internal \mid \\
\qquad isIntAction(ia) \in \text{dom}\ ctrlIntBS \cup \text{ran}\ ctrlIntBS \bullet isIntAction(ia)\} \bullet \\
\quad \langle annActs, internalActs\rangle \text{ partition dom } ctrlIntBS \cup \text{ran } ctrlIntBS)) \\
\hline
\end{array}
$$

This states that the only actions that can be found in the behaviour specification associated with an operational interface template are either announcements or internal actions.

[5] A considerable simplification.

Stream interface templates may similarly be represented by:

$$
\begin{array}{|l}
\textit{strIntTemp} \\
\hline
\textit{streams} : \mathbb{F}_1\ \textit{strIntSig} \\
\textit{strIntBS} : \textit{behspec} \\
\textit{strEnvCon} : \textit{EnvCon} \\
\hline
\forall\ \textit{sts} : \textit{strIntSig} \mid \textit{sts} \in \textit{streams} \bullet \\
\quad (\textbf{let}\ \textit{flowActs} == \{\textit{fs} : \textit{flowSig} \mid \textit{fs} \in \textit{sts.flows} \bullet \textit{isFlowAction}(\textit{fs})\} \bullet \\
\quad (\textbf{let}\ \textit{otherActs} == \{\textit{ia} : \textit{Internal} \mid \\
\qquad \textit{isIntAction}(\textit{ia}) \in \operatorname{dom} \textit{strIntBS} \cup \operatorname{ran} \textit{strIntBS} \bullet \textit{isIntAction}(\textit{ia})\} \bullet \\
\quad \langle \textit{flowActs}, \textit{otherActs} \rangle \text{ partition } \operatorname{dom} \textit{strIntBS} \cup \operatorname{ran} \textit{strIntBS})) \\
\hline
\end{array}
$$

This states that the only actions that can be found in the behaviour specification associated with a stream interface template are either stream actions or internal actions.

Computational interfaces represented as streams or control (operational) interfaces generally may thus be represented by:

$$\textit{compIntTemp} ::= \textit{control}\langle\langle \textit{ctrIntTemp} \rangle\rangle \mid \textit{stream}\langle\langle \textit{strIntTemp} \rangle\rangle$$

6 Binding in ODP

The ODP-RM states that the interfaces supporting computational objects may be bound provided they satisfy certain criteria: they must have complementary signatures, *i.e.* identical apart from causality being reversed. We argue that this is overly restrictive and not prescriptive enough. That is, requiring interfaces to be complementary is too strong as it prohibits interfaces with more behaviours from being bound, *e.g.* a server producing audio and video flows may well be bound to a client of video only if the client has the ability to simply ignore the audio flows as is the case for example with television. Or, a client only invoking a subset of the operations offered by a server interface could not be bound according to these rules. Similarly, requiring flow types to be the same is overly restrictive. It might well be the case that a producer of video flow can be replaced by a producer of audio/video if the consumer understands [6] the flow of information and can extract the video flow.

The ODP-RM also states nothing about the effect of environment contracts associated with interfaces on the legality of bindings. Thus we propose extending the ODP notion of binding.

6.1 Syntactic Compatibility of Stream Interfaces

One stream interface is syntactically compatible to a second if: all of the consumer flows in the first are matched by a producer flow in the second, and all of the consumer flows in the second are matched by producer flows in the first. Here we treat matching as having the same names and same flow types. Ideally we should deal with subtyping issues of multimedia flows, however, generically there are no specific rules since typing depends very much on issues such as compression techniques etc, *e.g.* an audio/video flow may or may not be a subtype of a video flow. This can be formalised by:

[6] This may well be based on having the correct decompression/decoding techniques etc.

$$strSyntaxOk : strIntSig \leftrightarrow strIntSig$$

$$\begin{array}{l}
\forall\, x, y : strIntSig \mid (x, y) \in strSyntaxOk \bullet (\forall\, ax : flowSig \mid ax \in x.flows \bullet \\
\quad (ax.role = Consumer \Rightarrow (\exists\, by : flowSig \mid by \in y.flows \bullet by.role = Producer \wedge \\
\qquad ax.fType = by.fType \wedge ax.fName = by.fName)) \wedge \\
\quad ax.role = Producer \Rightarrow \neg\, (\exists\, by : flowSig \mid by \in y.flows \bullet (by.role = Consumer \wedge \\
\qquad ax.fType = by.fType \wedge ax.fName = by.fName)))
\end{array}$$

6.2 Syntactic Compatibility of Control Interfaces

One server control interface is syntactically compatible to a second client control interface if it provides all of the server operations requested by the client interface, *i.e.* it may have more operations. One client control interface is syntactically compatible to a second server control interface if it doesn't request anything other than those operations in the server control interface. There are subtyping rules associated with the parameters of these operations, however, for simplicity sake we ignore these for now. These rules can be formalised by:

$$ctrSyntaxOk : ctrIntSig \leftrightarrow ctrIntSig$$

$$\begin{array}{l}
\forall\, x, y : ctrIntSig \mid (x, y) \in ctrSyntaxOk \bullet \\
\quad x.role = Client \wedge y.role = Server \Rightarrow (\forall\, ax : annSig \bullet ax \in x.anns \Rightarrow ax \in y.anns) \wedge \\
\quad x.role = Server \wedge y.role = Client \Rightarrow (\forall\, ax : annSig \bullet ax \in x.anns \Rightarrow \\
\quad \neg\, (\exists\, by : annSig \mid by \in y.anns \bullet ax.invName = by.invName \wedge ax.inArgs = by.inArgs))
\end{array}$$

6.3 Satisfying Environment Contracts

An environment contract may be deemed as being satisfied when the interface with which it is associated does not exhibit behaviours that contradict it. In our example, we consider throughput and maximum delay of multimedia interfaces. This can be represented as:

$$SatsCons : Constraints \leftrightarrow Constraints$$

$$\begin{array}{l}
\forall\, mft_1, mft_2 : muFlowType;\ c_1, c_2 : Constraints \mid (c_1, c_2) \in SatsCons \bullet \\
\quad (c_1 = thruPut(mft_1, maxThru(mft_1)) \wedge c_2 = thruPut(mft_2, maxThru(mft_2)) \wedge \\
\qquad maxThru(mft_1) \geq maxThru(mft_2)) \vee \\
\quad (c_1 = delay(mft_1, maxDelay(mft_1)) \wedge c_2 = delay(mft_2, maxDelay(mft_2)) \wedge \\
\qquad maxDelay(mft_1) \leq maxDelay(mft_2))
\end{array}$$

Here one constraint satisfies another constraint when the flow with which it is associated has a higher throughput and smaller delay. Environment contracts are satisfied when all constraints associated with them are satisfied. This can be represented as:

$$SatsEnvCon : EnvCon \leftrightarrow EnvCon$$

$$\begin{array}{l}
\forall\, ec_1, ec_2 : EnvCon \mid (ec_1, ec_2) \in SatsEnvCon \bullet \\
\quad (\forall\, qosC_1 : Constraints \mid qosC_1 \in ec_1.qosCons \bullet (\exists\, qosC_2 : Constraints \bullet \\
\qquad qosC_2 \in ec_2.qosCons \wedge (qosC_1, qosC_2) \in SatsCons))
\end{array}$$

6.4 Primitive Binding of Interfaces

Primitive binding occurs provided the two interfaces to be bound are syntactically compatible and their environment contracts are satisfied. The result of a primitive binding is a collection

of actions with an ordering between them. This ordering is given by the transitive closure of the two partial orderings associated with the behaviour specifications of the interfaces being bound.

$$
\begin{array}{|l}
\hline
extPrimBind : compIntTemp \times compIntTemp \pfun (action \rel action) \\
\hline
\forall\, cit_1, cit_2 : compIntTemp;\ str_1, str_2 : strIntTemp;\ strs_1, strs_2 : strIntSig \mid \\
\quad stream(str_1) = cit_1 \land stream(str_2) = cit_2 \land \\
\quad strs_1 \in str_1.streams \land strs_2 \in str_2.streams \bullet \\
\qquad (str_1.strEnvCon, str_2.strEnvCon) \in SatsEnvCon \land (strs_1, strs_2) \in strSyntaxOk \land \\
\qquad extPrimBind(cit_1, cit_2) = (str_1.strIntBS \cup str_2.strIntBS)^+ \lor \\
(\forall\, ctrl_1, ctrl_2 : ctrIntTemp;\ cis_1, cis_2 : ctrIntSig \mid \\
\quad control(ctrl_1) = cit_1 \land control(ctrl_2) = cit_2 \land \\
\quad cis_1 \in ctrl_1.ctrIntSigs \land cis_2 \in ctrl_2.ctrIntSigs \bullet \\
\qquad (ctrl_1.ctrlEnvCon, ctrl_2.ctrlEnvCon) \in SatsEnvCon \land (cis_1, cis_2) \in ctrSyntaxOk \land \\
\qquad extPrimBind(cit_1, cit_2) = (ctrl_1.ctrlIntBS \cup ctrl_2.ctrlIntBS)^+) \\
\hline
\end{array}
$$

This thus enables interfaces to be bound provided they are syntactically compatible and do not have contradictory environment contracts. To attempt to establish that the two interfaces being bound are semantically compatible would require that the two sets of partial orderings associated with the interface behaviour specifications be known and they not be contradictory. That is, if (a_1, a_2) were associated with the partial ordering of one interface then (a_2, a_1) would not be associated with the other interface. The actual ordering of two non-contradictory partial orders is then given by their transitive closure. Determining whether partial orderings are contradictory is likely to be problematic in most non-trivial behaviours.

7 Binding Objects and Firing of Actions

The binding object template in our example consists of two control interface templates and two stream interface templates. It will also have some behaviour that relates these interfaces together. This may simplistically be represented as:

$$
\begin{array}{|l}
\hline
BindingObjectTemplate \\
\hline
ctrlProd, ctrlCon : ctrIntTemp \\
strProd, strCon : strIntTemp \\
bs : behspec \\
\hline
\end{array}
$$

We do not give the precise behaviour specification here, since to do so would require a high level of prescriptivity on specific behaviours. Rather, we highlight the sort of actions that a binding object might have as part of its behaviour specification. Specifically, we consider an announcement for increasing the (isochronous) flow rate of the producer with which it is associated. This might be represented by:

$$
\begin{array}{|l}
\textit{FireFasterAnnouncement} \\
\hline
as? : annSig \\
ctrlProd, ctrlProd' : ctrIntTemp \\
strProd, strProd' : strIntTemp \\
\hline
\exists\, botCtrl : ctrIntTemp \bullet \\
\quad (control(botCtrl), control(ctrlProd)) \in \mathrm{dom}\, extPrimBind \land \\
\quad isAnnAction(as?) \in \mathrm{dom}\, botCtrl.ctrlIntBS \cap \mathrm{dom}\, ctrlProd.ctrlIntBS \land \\
\quad ctrlProd'.ctrIntSigs = ctrlProd.ctrIntSigs \land \\
\quad ctrlProd'.ctrlEnvCon = ctrlProd.ctrlEnvCon \land \\
\quad ctrlProd'.ctrlIntBS = \{isAnnAction(as?)\} \lhd ctrlProd.ctrlIntBS \land \\
\quad strProd'.streams = strProd.streams \land \\
\quad strProd'.strEnvCon = strProd.strEnvCon \land \\
\quad \mathrm{ran}\, strProd'.strIntBS = \mathrm{dom}\, strProd.strIntBS \land \\
\quad (\textbf{let}\ fasterFlow == (\mu\, fs_1, fs_2 : flowSig \mid \\
\qquad isFlowAction(fs_1) \in \mathrm{dom}\, strProd.strIntBS \land fs_1.fName = fs_2.fName \land \\
\qquad fs_1.role = fs_2.role \land fs_1.fType.rate < fs_2.fType.rate \land \\
\qquad fs_2.fType.rate \leq maxThru(fs_1.fType) \bullet fs_2) \bullet \\
\qquad \mathrm{dom}\, strProd'.strIntBS = \mathrm{dom}\, strProd.strIntBS \cup \{isFlowAction(fasterFlow)\})
\end{array}
$$

Here the control interface of the producer object must be legally bound to some binding object and the announcement being fired must be in the domain of the behaviour specification of both these objects, *i.e.* it is an action that can be fired at that moment. After the firing of the announcement, the interface signature and environment contract of the control interface are unchanged, but the behaviour specification is modified with all relations mapping that announcement to another action being removed. The stream interface also has an unchanged signature and environment contract, but the behaviour is changed so that its domain contains a new flow action with a faster rate, provided this does not exceed the maximum permitted as given by the environment contract.

For simplicity sake, we do not consider where the frames are sent or the modifications to the behaviour specification associated with the binding object involved in the firing of the announcement. Similarly we have not shown how the particular flow is identified or what sort of increase in frames produced is expected. The latter case is necessary in order to satisfy the proof obligation associated with the definite description, *i.e.* many different rate increases are likely to satisfy the predicates given, hence uniqueness cannot be guaranteed. These informations are likely to be associated with the parameters of the announcement.

8 Conclusions

This paper has shown how powerful the Z language is for specifying systems. Through the elementary ideas given through set theory and first order predicate logic, complex reasoning about the behaviour of systems can be achieved. The structure of these systems can be specified within the currently existing Z notation, as opposed to extensions to the notation. That is, for reasoning about behaviours, Z is perfectly adequate, but for considering issues such as encapsulation and state, then it is likely that an object-oriented flavour of Z would be better suited. The modelling of real-time behavioural issues can be dealt with adequately in Z, as opposed to Z and different temporal logics. This includes the continuous real time

synchronisation of multimedia such as might be found in lip-synchronisation of audio and video for example.

The specification of aspects of binding objects that have to be considered for distributed interworking of computational objects has been shown. This has extended the basic ODP idea of checking by signature only to include environment contracts and necessary conditions for behavioural type checking, *i.e.* non-contradictory behaviours.

8.1 Acknowledgements

The first author is supported by a joint grant from the United Kingdom Engineering and Physical Sciences Research Council and the Department of Trade and Industry, as part of the project FORMOSA (The Formalisation of the ODP Systems Architecture). This is a joint project between the University of Stirling and British Telecommunications (BT).

The Z in this document has been type checked using *fuzz* (Spivey 1993).

References

Blair, G. S. & Lea, R. (1993), The impact of distribution on support for object-oriented software development, Technical Report MPG-93-25, University of Lancaster, England.

Coombes, A. (1990), An interval logic for modelling time in Z, Technical report, Department of Computing Science, University of York.

Delisle, N. & Garlan, D. (1989), 'Formally specifying electronic instruments', *ACM SIGSOFT Eng. Notes* **14**, 242–248.

ISO/IEC (1989*a*), *Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model*, ISO/IEC 9074, International Organization for Standardization, Geneva, Switzerland.

ISO/IEC (1989*b*), *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO/IEC 8807, International Organization for Standardization, Geneva, Switzerland.

ISO/IEC (1995*a*), *Basic Reference Model of ODP – Part 1: Overview and Guide to Use of the Reference Model*, Draft International Standard 10746-1, Draft ITU-T Recommendation X.901, ISO/IEC ITU-T, Geneva, Switzerland.

ISO/IEC (1995*b*), *Basic Reference Model of ODP – Part 2: Foundations*, International Standard 10746-2, ITU-T X.902, ISO/IEC ITU-T, Geneva, Switzerland.

ISO/IEC (1995*c*), *Basic Reference Model of ODP – Part 3: Architecture*, International Standard 10746-3, ITU-T X.903, ISO/IEC ITU-T, Geneva, Switzerland.

ISO/IEC (1995*d*), *Basic Reference Model of ODP – Part 4: Architectural Semantics*, Draft International Standard 10746-4, Draft ITU-T Recommendation X.904, ISO/IEC ITU-T, Geneva, Switzerland.

ISO/IEC (1995*e*), *Basic Reference Model of ODP – Part 4.1: Architectural Semantics Amendment*, ISO/IEC JTC1/SC21 Working Document N9818, ISO/IEC ITU-T, Geneva, Switzerland.

ITU-T (1992), *International Consultative Committee on Telegraphy and Telephony – SDL – Specification and Description Language*, CCITT Z.100, International Telecommunications Union, Geneva, Switzerland.

King, P. (1989), A formal specification of signalling system number 7 link layer, Technical Report TR-101, University of Queensland, Key Centre for Software Technology.

Sinnott, R. & Turner, K. (1996), Specifying ODP Computational Objects in Z, *in* E. Najm & J.-B. Stefani, eds, 'Proceedings of Formal Methods for Open Object-based Distributed Systems', Paris, France.

Spivey, J. (1992), *The Z Notation: A Reference Manual*, Prentice-Hall International Series in Computing Science: C.A.R. Hoare Series Editor, second edn, Prentice-Hall International.

Spivey, J. (1993), *The Fuzz Manual*, Computing Science Consultancy. Second Printing.

Turner, K. J. (1996), 'Relating architecture and specification', *Computer Networks and ISDN Systems*. Accepted for publication in Special Edition on Specification Architecture.