Specifying ODP Computational Objects in Z

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

Abstract

The computational viewpoint contained within the Reference Model of Open Distributed Processing (RM-ODP) shows how collections of objects can be configured within a distributed system to enable interworking. It prescribes certain capabilities that such objects are expected to possess and structuring rules that apply to how these objects can be configured with one another. This paper highlights how the specification language Z can be used to formalise these capabilities and the associated structuring rules, thereby enabling specifications of ODP systems from the computational viewpoint to be achieved.

Keywords: Z; Open Distributed Processing; Architectural Semantics

1 Introduction

The current standardisation initiative of ODP (ISO/IEC 1995*a*, ISO/IEC 1995*b*, ISO/IEC 1995*c*, ISO/IEC 1995*d*, ISO/IEC 1995*e*) has advocated the use of formal methods. Indeed, one whole part of the standardisation work is devoted entirely to how formal methods can be used to enhance understanding of the reference model: the architectural semantics of ODP (ISO/IEC 1995*d*, ISO/IEC 1995*e*).

The architectural semantics 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 1995).

Initially, the scope of the architectural semantics work was concerned with the more fundamental concepts, *e.g.* object, interface, action. However, it was recognised that the scope of the work could be extended further through attempting to formalise the more prescriptive concepts of the RM-ODP: those concerned with the viewpoint languages. This paper provides the basis for the formalisation of one such viewpoint language in Z: the computational viewpoint language.

The rest of the paper is structured as follows. Section 2 gives a brief overview of the RM-ODP and introduces the viewpoint languages. It also considers the architectural semantics work and the different approaches that can be taken to it. Section 3 considers in detail the formalisation of the syntactic features of the computational language. Section 4 introduces a framework for consideration of the expected behaviour of computational objects. Section 5 considers how computational interface templates can be represented in Z. Section 6 shows how binding of computational interfaces can be achieved. Section 7 shows how computational objects can be represented in Z. Finally, section 8 draws some conclusions on the work done.

2 Overview of ODP

The RM-ODP 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 RM-ODP uses an object-oriented approach, where an object may be regarded as an identifiable, encapsulated aspect of some real world entity. The advantages of this with regard to systems development generally are well documented in the literature, *e.g.* (Meyer 1988). The advantages of this approach for distributed systems development are presented in some detail in (Blair & Lea 1993). The RM-ODP itself is divided into four main parts:

Part 1 - Overview and Guide to Use (ISO/IEC 1995*a*): contains an overview and guide to use of the RM-ODP.

Part 2 - Foundations (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 RM-ODP.

Part 3 - Architecture (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.

ODP uses the notion of a viewpoint as it recognises that it is not possible to capture effectively all aspects of design in a single description. A given viewpoint captures certain design facets of concern to a particular group involved in the design process. In doing so the complexity involved in considering the system is reduced. ODP recognises five viewpoints, each with its own associated language:

Enterprise Viewpoint: this focuses on the expression of purpose, policy and boundary for a given ODP system.

Information Viewpoint: this focuses on the information and information processing functions in a given ODP system.

Computational Viewpoint: this focuses on the functional decomposition of a given ODP system, and on the interworking and portability of ODP functions.

Engineering Viewpoint: this focuses on the infrastructure required to support distributed processing.

Technology Viewpoint: this focuses on suitable technologies to support distributed processing.

Part 4 - Architectural Semantics (ISO/IEC 1995*d*, ISO/IEC 1995*e*): 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.

Several approaches have been put forward to formalise the concepts of ODP, each with their own advantages and disadvantages as discussed in (Sinnott & Turner 1995). Briefly these are:

formalisation in natural language: here the approach is to write in English how the concept might best be modelled in a given formal language. The advantages of this approach are that it brings most understanding of the concept under consideration and it can be applied to all concepts. The disadvantage is that it is not directly applicable to writing specifications, *i.e.* no library of specification fragments has been built that can be used directly.

direct formalisation in mathematics: this approach models the concepts directly in mathematics, as opposed to via a formal language, *e.g.* Z. The advantage of this approach is that it is possible to represent precisely what is meant by the concepts considered. The disadvantages are that it is not particularly easy to understand and that it has little or no relation to the formal methods currently being considered in the architectural semantics work.

formalisation through specification templates: this approach develops explicit specification fragments that can be used to build ODP specifications. The advantages of this is that would-be specifiers can use the specification fragments directly to build their specifications. The disadvantage is that it is not always possible to give particular specification fragments due to being over-prescriptive. For example, it is not possible to give a precise specification fragment for an object since all objects are likely to have their own particular behaviours.

Thus each approach has certain advantages and disadvantages. This paper provides a formalisation based upon providing specification templates in Z — arguably the most useful of the three approaches for specifiers of ODP systems. In particular it provides specific behavioural fragments that have been identified as necessary from the computational viewpoint.

3 Syntactic Aspects of Computational Objects

The computational viewpoint contains the concepts and rules associated with objects and their associated interfaces. It prescribes the sort of interfaces that are found in ODP and the rules that apply to them, *e.g.* only interfaces in a specific relationship (such as subtype) can be connected. These rules are given in terms of syntactic aspects of the interface, *i.e.* its signature, as opposed to behavioural aspects. Thus all messages passed to objects will at least have an understood format.

To formalise the concepts associated with the computational viewpoint, it is necessary to introduce labels (*Name*) for things, *e.g.* names of operations and types. The types existing in the system are denoted by *TypeIdentifier*.

The parameters that are associated with interfaces to computational objects consist of a name and a type. It should always be possible to determine the type of a parameter in a given system. Thus *Parameter* is introduced as an injective function from names to types.

$$\begin{array}{|l} \mathit{Parameter} : \mathit{Name} ⤔ \mathit{TypeIdentifier} \end{array}$$

It is also useful to introduce sequences of these parameters.

$$\mathit{ParameterList} == \mathrm{seq}\, \mathit{Parameter}$$

Interfaces in the computational viewpoint can be stream, operational or signal. These have a causality either associated with the interface as a whole (operational) or with the individual actions associated with the interfaces (stream, signal). The different causalities may be represented by:

$$\mathit{Causality} ::= \mathit{Producer} \mid \mathit{Consumer} \mid \mathit{Initiator} \mid \mathit{Responder} \mid \mathit{Client} \mid \mathit{Server}$$

There are two basic kinds of operation in RM-ODP: interrogations and announcements. Interrogations consist of an invocation action followed by a non-empty finite set of termination actions. Announcements consist of only an invocation action.

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 invocation may thus be represented by the following schema:

$$\begin{array}{|l} \hline \mathit{InvocationTemplate} \\ \mathit{invocationName} : \mathit{Name} \\ \mathit{inArgs} : \mathit{ParameterList} \\ \hline \end{array}$$

A termination action is similar to an invocation, *i.e.* it has a name, number, names and types of result parameters.

$$\begin{array}{|l} \hline \mathit{TerminationTemplate} \\ \mathit{terminationName} : \mathit{Name} \\ \mathit{outArgs} : \mathit{ParameterList} \\ \hline \end{array}$$

Having defined invocation and termination actions, interrogations and announcements can be specified. An interrogation is a single invocation followed by a non-empty finite set of terminations:

$$
\begin{array}{|l}
\textit{InterrogationSignature} \\
\hline
\textit{invocation} : \textit{InvocationTemplate} \\
\textit{terminations} : \mathbb{F}_1\ \textit{TerminationTemplate} \\
\hline
\end{array}
$$

An announcement consists of a single invocation:

$$
\begin{array}{|l}
\textit{AnnouncementSignature} \\
\hline
\textit{invocation} : \textit{InvocationTemplate} \\
\hline
\end{array}
$$

Operational interface signatures consist of sets of announcements and interrogations, 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 invocation names in the interface are required to be unique. All termination names associated with a given invocation are required to be unique, and the parameter names associated with invocations and terminations are required to be unique. This can be represented as:

$$
\begin{array}{|l}
\textit{OperationInterfaceSignature} \\
\hline
\textit{anns} : \mathbb{P}\ \textit{AnnouncementSignature} \\
\textit{ints} : \mathbb{P}\ \textit{InterrogationSignature} \\
\textit{role} : \textit{Causality} \\
\hline
\textit{role} \in \{\textit{Client}, \textit{Server}\} \\
(\forall\ \textit{as}_1, \textit{as}_2 : \textit{AnnouncementSignature};\ \textit{is}_1, \textit{is}_2 : \textit{InterrogationSignature}; \\
\quad t_1, t_2 : \textit{TerminationTemplate};\ p_1, p_2 : \textit{Parameter} \bullet \\
\qquad (\textit{as}_1 \in \textit{anns} \land \textit{as}_2 \in \textit{anns} \land \textit{as}_1 \neq \textit{as}_2 \Rightarrow \\
\qquad\qquad \textit{as}_1.\textit{invocation}.\textit{invocationName} \neq \textit{as}_2.\textit{invocation}.\textit{invocationName}) \land \\
\qquad (\textit{is}_1 \in \textit{ints} \land \textit{is}_2 \in \textit{ints} \land \textit{is}_1 \neq \textit{is}_2 \Rightarrow \\
\qquad\qquad \textit{is}_1.\textit{invocation}.\textit{invocationName} \neq \textit{is}_2.\textit{invocation}.\textit{invocationName}) \land \\
\qquad (\textit{is}_1 \in \textit{ints} \land \textit{as}_1 \in \textit{anns} \Rightarrow \\
\qquad\qquad \textit{is}_1.\textit{invocation}.\textit{invocationName} \neq \textit{as}_1.\textit{invocation}.\textit{invocationName}) \land \\
\qquad (\textit{is}_1 \in \textit{ints} \land t_1 \in \textit{is}_1.\textit{terminations} \land t_2 \in \textit{is}_1.\textit{terminations} \land t_1 \neq t_2 \Rightarrow \\
\qquad\qquad t_1.\textit{terminationName} \neq t_2.\textit{terminationName}) \land \\
\qquad ((\textit{as}_1 \in \textit{anns} \land \langle p_1 \rangle \ \mathsf{in}\ \textit{as}_1.\textit{invocation}.\textit{inArgs} \land \langle p_2 \rangle \ \mathsf{in}\ \textit{as}_1.\textit{invocation}.\textit{inArgs}) \lor \\
\qquad\ (\textit{is}_1 \in \textit{ints} \land \langle p_1 \rangle \ \mathsf{in}\ \textit{is}_1.\textit{invocation}.\textit{inArgs} \land \langle p_2 \rangle \ \mathsf{in}\ \textit{is}_1.\textit{invocation}.\textit{inArgs}) \lor \\
\qquad\ (\textit{is}_1 \in \textit{ints} \land t_1 \in \textit{is}_1.\textit{terminations} \land \langle p_1 \rangle \ \mathsf{in}\ t_1.\textit{outArgs} \land \\
\qquad\qquad\qquad \langle p_2 \rangle \ \mathsf{in}\ t_1.\textit{outArgs}) \land p_1 \neq p_2) \Rightarrow \textit{first}\ p_1 \neq \textit{first}\ p_2) \\
\hline
\end{array}
$$

Signals represent the most basic unit of interaction in the computational viewpoint. They may be considered as single, atomic actions between computational objects. They have associated with them a name, the number, names and types of parameters, and a causality. A signal signature may thus be represented by:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SignalSignature}} \\ \hline
\mathit{signalName} : \mathit{Name} \\
\mathit{args} : \mathit{ParameterList} \\
\mathit{role} : \mathit{Causality} \\ \hline
\mathit{role} \in \{\mathit{Initiator}, \mathit{Responder}\} \\ \hline
\end{array}$$

A signal interface signature consists of a set of signal signatures. Each signal name associated with a given signal interface signature is required to be unique, and the parameters names associated with signals are required to be unique.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SignalInterfaceSignature}} \\ \hline
\mathit{signals} : \mathbb{P}\, \mathit{SignalSignature} \\ \hline
\forall\, ss_1, ss_2 : \mathit{SignalSignature};\ p_1, p_2 : \mathit{Parameter} \bullet \\
\qquad (ss_1 \in \mathit{signals} \wedge ss_2 \in \mathit{signals} \wedge ss_1 \neq ss_2 \Rightarrow \\
\qquad\qquad ss_1.\mathit{signalName} \neq ss_2.\mathit{signalName}) \wedge \\
\qquad (ss_1 \in \mathit{signals} \wedge \langle p_1 \rangle \ \mathsf{in}\ ss_1.\mathit{args} \wedge \langle p_2 \rangle \ \mathsf{in}\ ss_1.\mathit{args} \wedge p_1 \neq p_2 \Rightarrow \\
\qquad\qquad \mathit{first}\ p_1 \neq \mathit{first}\ p_2) \\ \hline
\end{array}$$

The computational viewpoint also considers interfaces concerned with the continuous flow of data, *e.g.* multimedia. The exact nature of the flow of information is abstracted away from; it is represented here simply as a type[1]. Flow signatures also contain a name for the flow and an indication of the causality of the flow. This may be represented as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{FlowSignature}} \\ \hline
\mathit{flowName} : \mathit{Name} \\
\mathit{flowType} : \mathit{TypeIdentifier} \\
\mathit{role} : \mathit{Causality} \\ \hline
\mathit{role} \in \{\mathit{Producer}, \mathit{Consumer}\} \\ \hline
\end{array}$$

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:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StreamInterfaceSignature}} \\ \hline
\mathit{flows} : \mathbb{P}\, \mathit{FlowSignature} \\ \hline
\forall\, fs_1, fs_2 : \mathit{FlowSignature} \bullet \\
\qquad fs_1 \in \mathit{flows} \wedge fs_2 \in \mathit{flows} \wedge fs_1 \neq fs_2 \Rightarrow fs_1.\mathit{flowName} \neq fs_2.\mathit{flowName} \\ \hline
\end{array}$$

[1] In reality it is likely to be a more complex type due to the properties associated with it, *e.g.* temporal aspects of the flow.

Before proceeding to show how these syntactic structures can be used to build computational interfaces and computational objects, it is necessary to consider issues of behaviour, *i.e.* the behaviour specification associated with the object and interface templates.

4 Introducing Behavioural Considerations

Computational interfaces and the objects that they support can be considered as consisting of signatures that are offered to the environment, *i.e.* the other objects in the system, and a behaviour specification[2] that corresponds to what occurs when the operations associated with these signatures are invoked from the environment.

A behaviour specification consists of a (possibly infinite) set of distinct[3] actions with constraints on their occurrence. These constraints impose a partial ordering on the set of actions. The actions themselves can be internal to the object or observable to the environment, *i.e.* require participation (synchronisation) with the environment to occur. We introduce the basic type

$$[action]$$

to denote the set of all possible actions. This will later (section 7) be replaced by another type related to the specific actions that can be associated with computational object templates once these action templates have been developed.

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

$$\begin{aligned} behspec == \{ & ar_1, ar_2 : action \leftrightarrow action \mid \\ & \quad ar_1 = ar_1^+ \land ar_1 \cap ar_1^\sim = \varnothing \land ar_2 = ar_1^* \bullet ar_2\} \end{aligned}$$

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 transitive closure, which is the same as saying that it is transitive. The expression $ar_1 \cap ar_1^\sim = \varnothing$ ensures that the relation is anti-symmetric, *i.e.* no two actions in the relation are related by the inverse of the relation also. Finally, the expression $ar_2 = ar_1^*$ states that ar_2 is ar_1 with the addition of all the reflexive pairs. Thus relation ar_2 is a relation that is transitive, anti-symmetric and reflexive, *i.e.* a partial order.

4.1 Consideration of Interface Templates

In order to consider computational interfaces, it is necessary to introduce some functions that map action signatures to actions, *i.e.* invocation templates to invocation actions, etc.

[2] Computational objects and interfaces are also required to possess environment contracts; however, consideration of these is outside the scope of this paper.

[3] 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.

As will be seen in section 7, these represent only a subset of the possible action templates that can be associated with computational objects. We also introduce the special action *Internal*.

$$\begin{array}{l}
Internal == action \\
Fail == action
\end{array}$$

Internal corresponds to an action in the behaviour specification of an interface or object that does not require synchronisation with the environment to occur. *Fail* is a special action that models the failure (or non-occurrence) of some other action.

$$\begin{array}{|l}
isInternalAction : Internal \rightarrowtail action \\
isFailAction : Fail \rightarrowtail action \\
isInvocationAction : InvocationTemplate \rightarrowtail action \\
isTerminationAction : TerminationTemplate \rightarrowtail action \\
isSignalAction : SignalSignature \rightarrowtail action \\
isFlowAction : FlowSignature \rightarrowtail action \\
\hline
\langle \mathrm{ran}\ isInvocationAction, \mathrm{ran}\ isTerminationAction, \mathrm{ran}\ isSignalAction, \\
\quad \mathrm{ran}\ isFlowAction, \mathrm{ran}\ isFailAction, \mathrm{ran}\ isInternalAction \rangle \ \mathsf{partition}\ action
\end{array}$$

Computational interfaces are represented by a signature and a behaviour specification. Operational interface templates may thus be represented by:

$$\begin{array}{|l}
\multicolumn{1}{l}{OperationalInterfaceTemplate} \\
\hline
operations : \mathbb{F}_1\ OperationInterfaceSignature \\
opIntTempBehSpec : behspec \\
\hline
\forall\, ois : OperationInterfaceSignature \mid ois \in operations \bullet \\
\quad (\mathbf{let}\ invActs == \{ invAct : InvocationTemplate \mid \\
\qquad (\exists\, as : AnnouncementSignature; is : InterrogationSignature \mid \\
\qquad\ as \in ois.anns \lor is \in ois.ints \bullet \\
\qquad\qquad invAct \in \{as.invocation\} \lor invAct \in \{is.invocation\}) \bullet \\
\qquad\qquad\quad isInvocationAction(invAct)\} \bullet \\
\quad (\mathbf{let}\ termActs == \{ termAct : TerminationTemplate \mid \\
\qquad (\exists\, is : InterrogationSignature \mid is \in ois.ints \bullet termAct \in is.terminations) \bullet \\
\qquad\qquad\quad isTerminationAction(termAct)\} \bullet \\
\quad (\mathbf{let}\ otherActs == \{ ia : Internal \mid \\
\qquad isInternalAction(ia) \in \mathrm{dom}\ opIntTempBehSpec \cup \mathrm{ran}\ opIntTempBehSpec \bullet \\
\qquad\qquad\qquad isInternalAction(ia)\} \bullet \\
\qquad\qquad \langle invActs, termActs, otherActs \rangle\ \mathsf{partition} \\
\qquad\qquad\qquad \mathrm{dom}\ opIntTempBehSpec \cup \mathrm{ran}\ opIntTempBehSpec))) \\
\hline
\end{array}$$

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

Stream interface templates may be represented by:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;StreamInterfaceTemplate\;}} \\
streams : \mathbb{F}_1\, StreamInterfaceSignature \\
strIntTempBehSpec : behspec \\
\hline
\forall\, sts : StreamInterfaceSignature \mid sts \in streams \bullet \\
\quad (\mathbf{let}\ flowActs == \{fs : FlowSignature \mid fs \in sts.flows \bullet isFlowAction(fs)\} \bullet \\
\quad (\mathbf{let}\ otherActs == \{ia : Internal \mid \\
\qquad\quad isInternalAction(ia) \in \operatorname{dom} strIntTempBehSpec \cup \operatorname{ran} strIntTempBehSpec \bullet \\
\qquad\qquad\qquad\qquad\qquad isInternalAction(ia)\} \bullet \\
\qquad\qquad\qquad \langle flowActs, otherActs\rangle \ \mathsf{partition} \\
\qquad\qquad\qquad\quad \operatorname{dom} strIntTempBehSpec \cup \operatorname{ran} strIntTempBehSpec)) \\
\hline
\end{array}$$

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.

Signal interface templates may be represented by:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;SignalInterfaceTemplate\;}} \\
signals : \mathbb{F}_1\, SignalInterfaceSignature \\
sigIntTempBehSpec : behspec \\
\hline
\forall\, sis : SignalInterfaceSignature \mid sis \in signals \bullet \\
\quad (\mathbf{let}\ sigActs == \{ss : SignalSignature \mid \\
\qquad\quad ss \in sis.signals \bullet isSignalAction(ss)\} \bullet \\
\quad (\mathbf{let}\ otherActs == \{ia : Internal \mid \\
\qquad\quad isInternalAction(ia) \in \operatorname{dom} sigIntTempBehSpec \cup \operatorname{ran} sigIntTempBehSpec \bullet \\
\qquad\qquad\qquad\qquad\qquad isInternalAction(ia)\} \bullet \\
\qquad\qquad\qquad \langle sigActs, otherActs\rangle \ \mathsf{partition} \\
\qquad\qquad\qquad\quad \operatorname{dom} sigIntTempBehSpec \cup \operatorname{ran} sigIntTempBehSpec)) \\
\hline
\end{array}$$

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

Computational interface templates can be operational, signal or stream interface templates. This can be represented by:

$$\begin{array}{lll}
ComputationalInterfaceTemplate & ::= & operational\langle\!\langle OperationalInterfaceTemplate\rangle\!\rangle \mid \\
 & & stream\langle\!\langle StreamInterfaceTemplate\rangle\!\rangle \mid \\
 & & signal\langle\!\langle SignalInterfaceTemplate\rangle\!\rangle
\end{array}$$

Before proceeding to show how these computational interface templates can be used to build computational object templates, it is necessary to consider other action templates that can be associated with computational objects. Specifically, the computational viewpoint identifies behaviours related to the forking, joining and spawning of activities, and the binding of interfaces.

5 Behavioural Activity Considerations

An activity may be regarded as a single headed, directed acyclic graph of actions where occurrence of actions is made possible by the occurrence of all immediately preceding actions, *i.e.* by all adjacent actions closer to the head.

In order to consider the activities that can be associated with an object, it is necessary to consider the actions associated with an object and the constraints on their occurrence as a directed graph (*digraph*) of actions. A digraph is a set of actions (*as*) with an ordering relation (*or*) between them. This can be represented as:

$$digraph == \{as : \mathbb{P}\, action; or : action \leftrightarrow action \mid (\mathrm{dom}\, or \cup \mathrm{ran}\, or) \subseteq as\}$$

A directed acyclic graph (*dag*) is a directed graph that contains no cycles.

$$dag == \{as : \mathbb{P}\, action; or : action \leftrightarrow action \mid (as, or) \in digraph \land \mathsf{disjoint}\ \langle or^+, \mathrm{id}\, action\rangle\}$$

Here or^+ represents the transitive closure of the ordering relation and *id* is the identity relation on a set. The above states that no node can be reached in one or more steps from itself. Thus there are no cycles in the graph.

A connected directed acyclic graph (*condag*) is a directed acyclic graph that does not have separate subgraphs. This can be represented as:

$$condag == \{as : \mathbb{P}\, action; or : action \leftrightarrow action \mid (as, or) \in dag \land (or \cup or^{\sim})^* = action \times action\}$$

Here $or^{\sim}$ represents the relational inverse of the directed edge relation. Thus $or \cup or^{\sim}$ describes edges where the nodes are joined in both directions and $(or \cup or^{\sim})^*$ is the reflexive transitive closure of the edge relation, *i.e.* it relates all nodes reachable by zero or more steps along the edges. $action \times action$ is the set of all pairs of sets of actions. The condition therefore states that all nodes can be reached from all others in zero or more steps, hence the graph is connected.

Finally an activity corresponds to a connected acyclic direct graphs of actions that is single headed.

$$activity == \{as : \mathbb{P}\, action; or : action \leftrightarrow action \mid (as, or) \in condag \land (\exists\, a : action \bullet \{a\} = as \setminus \mathrm{ran}\, or)\}$$

Computational objects may be associated with specific forms of activities. Of particular importance are those activities connected with chains, where a chain (*Chain*) may be regarded as a sequence of actions within an activity where for each adjacent pair of actions, occurrence of the first is necessary for the occurrence of the second action. A chain (*Chain*) may thus be represented by:

$$Chain == \{sa : \mathrm{seq}\, action \mid (\exists\, act : activity \bullet (\forall\, a_1, a_2 : action \mid \langle a_1, a_2\rangle\ \mathsf{in}\ sa \bullet \{a_1, a_2\} \subseteq first\ act \land (a_1 \mapsto a_2) \in second\ act))\}$$

Through considering chains, objects having their own separate behaviours can be reasoned about. That is, dividing and joining actions can be considered. A joining action (*Join*) is an action that is shared between two or more chains that results in a single chain. This can be represented as:

JoinAction

$$join : Chain \times Chain \rightarrowtail\!\!\!\!\!\!\;\; + \; Chain$$

$$\begin{array}{l}
\forall\, c_1, c_2, c_3 : Chain \mid c_1 \neq c_2 \neq c_3 \bullet join(c_1, c_2) = c_3 \Rightarrow \\
\qquad (\exists\, a : action \bullet \langle a \rangle \ \mathsf{in}\ c_1 \wedge \langle a \rangle \ \mathsf{in}\ c_2 \wedge last\ c_1 \neq last\ c_2 \wedge \\
\qquad\qquad (a = last\ c_1 \Rightarrow c_3 = tail\,(SeqRestrict(a, c_2))) \vee \\
\qquad\qquad (a = last\ c_2 \Rightarrow c_3 = tail\,(SeqRestrict(a, c_1))))
\end{array}$$

Here *SeqRestrict* is a function that takes an action and sequence of actions as arguments and produces a subsequence of the sequence argument. This subsequence is given by the sequence of actions following the action argument.

$$SeqRestrict : action \times \operatorname{seq} action \rightarrowtail\!\!\!\!\!\!\;\; + \; \operatorname{seq} action$$

$$\begin{array}{l}
\forall\, a : action;\ sa_1 : \operatorname{seq} action \bullet \\
\qquad SeqRestrict(a, sa_1) = \mathbf{if}\ a = head\ sa_1\ \mathbf{then}\ sa_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ SeqRestrict(a, tail\,(sa_1))
\end{array}$$

It should be noted here that this recursive definition has no base case since its usage requires that the action is in the sequence as a precondition in *JoinAction*. See section 8 for the repercussions of this formalisation of *JoinAction*.

Dividing actions are actions that enable two or more chains. There are two cases of dividing action: forking actions (*Fork*) in which the enabled chains eventually join each other, and spawning actions (*Spawn*) in which the enabled chains do not join each other. These can be represented by:

ForkAction

$$fork : Chain \rightarrowtail\!\!\!\!\!\!\;\; + \; Chain \times Chain$$

$$\begin{array}{l}
\forall\, c_1, c_2, c_3 : Chain \mid c_1 \neq c_2 \neq c_3 \bullet fork(c_1) = (c_2, c_3) \Rightarrow \\
\qquad (\exists\, a : action;\ j : JoinAction \bullet \\
\qquad\qquad \langle a \rangle \ \mathsf{in}\ c_1 \wedge c_2 = SeqRestrict(a, c_1) \wedge (c_2, c_3) \in \operatorname{dom} j.join)
\end{array}$$

SpawnAction

$$spawn : Chain \rightarrowtail\!\!\!\!\!\!\;\; + \; Chain \times Chain$$

$$\begin{array}{l}
\forall\, c_1, c_2, c_3 : Chain \mid c_1 \neq c_2 \neq c_3 \bullet spawn(c_1) = (c_2, c_3) \Rightarrow \\
\qquad (\exists\, a : action;\ j : JoinAction \bullet \\
\qquad\qquad \langle a \rangle \ \mathsf{in}\ c_1 \wedge c_2 = SeqRestrict(a, c_1) \wedge (c_2, c_3) \notin \operatorname{dom} j.join)
\end{array}$$

The formalisation of these actions also have repercussions that are considered in more detail in section 8.

6 Binding Computational Interfaces

The interfaces supporting computational objects may be bound provided they satisfy certain criteria: they must have complementary signatures. A signature is complementary to another one if it is identical apart from its causality being reversed. For operational interface signatures this requires that one interface has client and the other server causality.

$$\begin{array}{|l}
\mathit{OpIntComp} : \mathit{OperationInterfaceSignature} \leftrightarrow \mathit{OperationInterfaceSignature} \\
\hline
\forall x, y : \mathit{OperationInterfaceSignature} \mid (x, y) \in \mathit{OpIntComp} \bullet \\
\quad ((x.\mathit{role} = \mathit{Client} \land y.\mathit{role} = \mathit{Server}) \lor (x.\mathit{role} = \mathit{Server} \land y.\mathit{role} = \mathit{Client})) \land \\
\quad (\forall \mathit{int} : \mathit{InterrogationSignature}; \mathit{ann} : \mathit{AnnouncementSignature} \bullet \\
\qquad (\mathit{int} \in x.\mathit{ints}) \Leftrightarrow (\mathit{int} \in y.\mathit{ints}) \land (\mathit{ann} \in x.\mathit{anns}) \Leftrightarrow (\mathit{ann} \in y.\mathit{anns}))
\end{array}$$

For stream interface signatures complementarity requires that one interface has consumer causality and the other producer causality.

$$\begin{array}{|l}
\mathit{StrIntComp} : \mathit{StreamInterfaceSignature} \leftrightarrow \mathit{StreamInterfaceSignature} \\
\hline
\forall x, y : \mathit{StreamInterfaceSignature} \mid (x, y) \in \mathit{StrIntComp} \bullet \\
\quad (\forall \mathit{ax} : \mathit{FlowSignature} \mid \mathit{ax} \in x.\mathit{flows} \bullet \\
\qquad (\exists \mathit{by} : \mathit{FlowSignature} \bullet \mathit{by} \in y.\mathit{flows} \land \\
\qquad\quad ((\mathit{ax}.\mathit{role} = \mathit{Producer} \land \mathit{by}.\mathit{role} = \mathit{Consumer}) \lor \\
\qquad\quad (\mathit{ax}.\mathit{role} = \mathit{Consumer} \land \mathit{by}.\mathit{role} = \mathit{Producer})) \land \\
\qquad\qquad \mathit{ax}.\mathit{flowType} = \mathit{by}.\mathit{flowType}))
\end{array}$$

For signal interface signatures complementarity requires that one interface has initiator causality and the other responder causality.

$$\begin{array}{|l}
\mathit{SigIntComp} : \mathit{SignalInterfaceSignature} \leftrightarrow \mathit{SignalInterfaceSignature} \\
\hline
\forall x, y : \mathit{SignalInterfaceSignature} \mid (x, y) \in \mathit{SigIntComp} \bullet \\
\quad (\forall \mathit{ax} : \mathit{SignalSignature} \mid \mathit{ax} \in x.\mathit{signals} \bullet \\
\qquad (\exists \mathit{by} : \mathit{SignalSignature} \bullet \mathit{by} \in y.\mathit{signals} \land \\
\qquad\quad ((\mathit{ax}.\mathit{role} = \mathit{Initiator} \land \mathit{by}.\mathit{role} = \mathit{Responder}) \lor \\
\qquad\quad (\mathit{ax}.\mathit{role} = \mathit{Responder} \land \mathit{by}.\mathit{role} = \mathit{Initiator})) \land \\
\qquad\quad \mathit{ax}.\mathit{args} = \mathit{by}.\mathit{args} \land \mathit{ax}.\mathit{signalName} = \mathit{by}.\mathit{signalName}))
\end{array}$$

Binding actions can be implicit, compound or primitive. Implicit binding is used in notations with no explicit terms that can be used to express the binding action. It is defined only for server operational interfaces, since it is not known where the initiative on subsequent interactions is to be placed following binding. Compound binding enables sets of interfaces to be bound through a binding object. Primitive binding simply binds an interface of an object to another interface. It is primitive binding that is considered here.

Primitive binding occurs provided the two interfaces to be bound are complementary. The result of this 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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BindAction}} \\
\mathit{cit}_1, \mathit{cit}_2 : \mathit{ComputationalInterfaceTemplate} \\
\mathit{res}! : \mathit{action} \leftrightarrow \mathit{action} \\
\hline
(\forall \mathit{sit}_1, \mathit{sit}_2 : \mathit{SignalInterfaceTemplate};\ \mathit{sis}_1, \mathit{sis}_2 : \mathit{SignalInterfaceSignature} \mid \\
\quad \mathit{signal}(\mathit{sit}_1) = \mathit{cit}_1 \land \mathit{signal}(\mathit{sit}_2) = \mathit{cit}_2 \land \\
\quad \mathit{sis}_1 \in \mathit{sit}_1.\mathit{signals} \land \mathit{sis}_2 \in \mathit{sit}_2.\mathit{signals} \bullet (\mathit{sis}_1, \mathit{sis}_2) \in \mathit{SigIntComp} \land \\
\quad\ \mathit{res}! = (\mathit{sit}_1.\mathit{sigIntTempBehSpec} \cup \mathit{sit}_2.\mathit{sigIntTempBehSpec})^{+}) \lor \\
(\forall \mathit{str}_1, \mathit{str}_2 : \mathit{StreamInterfaceTemplate};\ \mathit{strs}_1, \mathit{strs}_2 : \mathit{StreamInterfaceSignature} \mid \\
\quad \mathit{stream}(\mathit{str}_1) = \mathit{cit}_1 \land \mathit{stream}(\mathit{str}_2) = \mathit{cit}_2 \land \\
\quad \mathit{strs}_1 \in \mathit{str}_1.\mathit{streams} \land \mathit{strs}_2 \in \mathit{str}_2.\mathit{streams} \bullet (\mathit{strs}_1, \mathit{strs}_2) \in \mathit{StrIntComp} \land \\
\quad\ \mathit{res}! = (\mathit{str}_1.\mathit{strIntTempBehSpec} \cup \mathit{str}_2.\mathit{strIntTempBehSpec})^{+}) \lor \\
(\forall \mathit{oit}_1, \mathit{oit}_2 : \mathit{OperationalInterfaceTemplate};\ \mathit{ois}_1, \mathit{ois}_2 : \mathit{OperationInterfaceSignature} \mid \\
\quad \mathit{operational}(\mathit{oit}_1) = \mathit{cit}_1 \land \mathit{operational}(\mathit{oit}_2) = \mathit{cit}_2 \land \\
\quad \mathit{ois}_1 \in \mathit{oit}_1.\mathit{operations} \land \mathit{ois}_2 \in \mathit{oit}_2.\mathit{operations} \bullet (\mathit{ois}_1, \mathit{ois}_2) \in \mathit{OpIntComp} \land \\
\quad\ \mathit{res}! = (\mathit{oit}_1.\mathit{opIntTempBehSpec} \cup \mathit{oit}_2.\mathit{opIntTempBehSpec})^{+}) \\
\hline
\end{array}
$$

This thus enables interfaces to be bound provided they are syntactically compatible. 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 Consideration of Object Templates

As stated, computational object templates consist of collections of actions with constraints on their occurrence. The specific actions related to computational objects include those associated with their interfaces, *e.g.* invocations and terminations, and those related to binding and dividing and joining actions. This can be represented by the parameterised free type definition *Action*, where the function *CompAct* relates the basic type *action* to the new type *Action*, and the other functions represent mappings from action signatures to actions. Thus a computational action may be represented by:

$$
\begin{array}{ll}
\mathit{Action} ::= & \mathit{CompAct}\langle\langle \mathit{action} \rangle\rangle \mid \\
& \mathit{isForkAction}\langle\langle \mathit{ForkAction} \rangle\rangle \mid \\
& \mathit{isSpawnAction}\langle\langle \mathit{SpawnAction} \rangle\rangle \mid \\
& \mathit{isJoinAction}\langle\langle \mathit{JoinAction} \rangle\rangle \mid \\
& \mathit{isBindAction}\langle\langle \mathit{BindAction} \rangle\rangle
\end{array}
$$

As before, a behaviour specification may be given as a set of actions with a partial ordering relation between them. Here the actions under consideration must be of the kind *Action*.

$$\begin{array}{l}
BehSpec == \{AR_1, AR_2 : Action \leftrightarrow Action \mid \\
\qquad\qquad AR_1 = AR_1^{+} \land AR_1 \cap AR_1^{\sim} = \emptyset \land AR_2 = AR_1^{*} \bullet AR_2\}
\end{array}$$

Computational object templates consist of a collection of interface templates and a behaviour specification. The exact kinds of actions that can be associated with computational objects can now be prescribed however. These must be one of those that make up the parameterised free type *Action*.

$$\begin{array}{|l}
\hline
\textit{ComputationalObjectTemplate} \\
ints : \mathbb{F}_1\ ComputationalInterfaceTemplate \\
bs : BehSpec \\
\hline
\forall a : Action \mid a \in \mathrm{dom}\ bs \cup \mathrm{ran}\ bs \bullet \\
\qquad (\exists i : Internal \bullet CompAct(isInternalAction(i)) = a) \lor \\
\qquad (\exists f : Fail \bullet CompAct(isFailAction(f)) = a) \lor \\
\qquad (\exists i : InvocationTemplate \bullet CompAct(isInvocationAction(i)) = a) \lor \\
\qquad (\exists t : TerminationTemplate \bullet CompAct(isTerminationAction(t)) = a) \lor \\
\qquad (\exists s : SignalSignature \bullet CompAct(isSignalAction(s)) = a) \lor \\
\qquad (\exists f : FlowSignature \bullet CompAct(isFlowAction(f)) = a) \lor \\
\qquad (\exists j : JoinAction \bullet isJoinAction(j) = a) \lor \\
\qquad (\exists f : ForkAction \bullet isForkAction(f) = a) \lor \\
\qquad (\exists s : SpawnAction \bullet isSpawnAction(s) = a) \lor \\
\qquad (\exists b : BindAction \bullet isBindAction(b) = a) \\
\hline
\end{array}$$

8 Conclusions and Acknowledgements

This paper has shown how the formal language Z can be used to model ODP computational object templates in Z. A re-usable library of specification fragments has been developed along with structuring rules that apply to them, thus enabling production of ODP-compliant specifications from the computational viewpoint. Amongst the advantages in doing this, as opposed to some other formal technique, such as LOTOS, are that many of the structuring rules can be specified directly within the Z text. For example, ensuring naming rules are not violated in interface signatures in a LOTOS specification requires the specifier follows an informal modelling style, *i.e.* they have to specify unique names themselves when writing their specifications. In Z, however, these rules can be enforced through having explicit predicates associated with the interface signatures and the names contained within them, as given here.

This paper also presents issues that need to be dealt with when considering type management issues. For example, as opposed to dealing with the predominantly syntactic considerations involved in signature type checking as presented in Part 3 of the reference model and in work such as (Brookes & Indulska 1994), this paper attempts to highlight issues that need to be dealt with when attempting behavioural type checking. Examples of this involve determining that partial orders in the behaviour specifications associated with interfaces are not contradictory. It should be noted that it is possible to specify such things quite readily in Z, however it is likely that this will be undecidable in general.

This paper has deliberately avoided dealing with issues that Z does not handle adequately. For example, whilst Z is good at modelling static views of possible behaviours, it is poor at modelling dynamic behaviours that actually occur. It is for this reason that instantiation of interface and object templates and the actual occurrence of their actions has been avoided. As a result, attempts at modelling collections of interacting computational objects has not been made. This is made especially difficult due to the lack of encapsulation and interaction semantics in Z. It is for further consideration how object-oriented versions of Z can be used to model such configurations. Certainly, encapsulation and forms of interaction can made possible through the use of object-oriented versions of Z (Stepney, Barden & Cooper 1992).

Finally, this paper further enforces the advantages to be gained from developing an architectural semantics. Not only were numerous ambiguities identified in the reference model of ODP, but also several areas were identified where clarification was necessary. For example, the reference model describes the form of the interfaces to computational objects, *e.g.* their signature structures, and then proceeds to detail what a computational object template should be able to do. The relation between computational object templates and the structure of their interfaces is somewhat vague. For example, computational object templates should be able to spawn, fork and join activities but there is no mention as to how this can be achieved. That is, are these all possible through operations, signals and streams, or are they somehow different? Similarly, the definitions of dividing and joining actions are imprecise. In joining say, do both of the joining chains terminate with the production of a new chain, or does one chain terminate and the other carry on. Likewise for forking and spawning actions, there is no mention about the continuation of the existing chain. The approach taken here has been to assume the chains continue to exist. These issues are typical of those that remain hidden without developing an architectural semantics.

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 authors would thus like to thank the chief collaborators in this project from BT, namely Steve Rudkin and Pete Young. Special thanks also go to Pete for reading through the Z of earlier drafts of this paper.

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.

Brookes, W. & Indulska, J. (1994), ODP Types and their Management: An Object-Z Specification, *in* K. Raymond & E. Armstrong, eds, 'Open Distributed Processing: Experiences with Distributed Envirnonments', Chapman and Hall, pp. 425–437.

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.

Meyer, B. (1988), *Object Oriented Software Construction*, Prentice-Hall International Series in Computing Science: C.A.R. Hoare Series Editor, Prentice-Hall International.

Sinnott, R. & Turner, K. J. (1995), 'Applying formal methods to standard development: The open distributed processing experience', *Computer Standards & Interfaces* **17**, 615–630.

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.

Stepney, S., Barden, R. & Cooper, D., eds (1992), *Object Orientation in Z*, Springer-Verlag.

Turner, K. J. (1995), 'Relating architecture and specification', *Computer Networks and ISDN Systems*. Accepted for publication in Special Edition on Specification Architecture.