Rolia, Jacob Slonim and John Botsford, editors, Open Distributed
Processing and Distributed Platforms, pages 85-96, Chapman and Hall, May 1997.

Type Checking in Open Distributed Systems: a Complete Model and its Z Specification

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

January 15, 1997

Abstract

Type checking is at the heart of distributed systems. The ability to be able to configure objects and have them interwork correctly may well be regarded as the fundamental issue in the development of reliable distributed systems. The type system put forward in the current standardisation activity of Open Distributed Processing (ODP), however, is both *inadequate* and *incorrect*. The inadequacy is due to the scope of the type system being based entirely on syntactic issues. To achieve reliable interoperability between systems, a type system should deal with behavioural (semantic) issues as well as non-functional issues, *i.e.* aspects of the type that its signature and behaviour do not capture. The incorrectness is due to the syntactic issues not being dealt with correctly. That is, clients and servers have fundamentally different type rules that apply to them. We provide a Z specification of a robust type system that deals with the syntactic aspects of types (correctly) as well as a treatment of the behavioural and non-functional aspects of types.

Keywords: Z; Types; Open Distributed Processing.

1 Introduction

It could be argued that the ability to develop sub-systems of some larger more complex system and compose them with a guarantee that they will interwork successfully is the most important consideration in software engineering today. Object-orientation helps in simplifying this task to some extent through introducing the idea of objects (sub-systems). Objects interact with the outside world through a clearly defined interface and they hide (encapsulate) information internal to themselves. Through consideration of the interfaces of objects, composition of sub-systems can be reasoned about more simply. This is possible if interface types are introduced into the system. That is, sub-systems can be composed with one another provided they have compatible interfaces, where compatibility is based on some form of interface type equivalence.

We investigate this notion of type equivalence using the work on the current standardisation of Open Distributed Processing (ODP) (ISO/IEC 1995*a*, ISO/IEC 1995*b*, ISO/IEC 1995*c*, ISO/IEC 1995*d*, ISO/IEC 1995*e*). We argue that the types found in distributed systems are required to satisfy more complex equivalence relations than has so far been suggested if successful interworking of sub-systems is to be achieved. We also show how the

type rules associated with distributed systems in general are different to those that might be found elsewhere. We present a formalisation of these new considerations in the language Z (ISO/IEC 1995*f*). This formalisation is particularly powerful in that it deals with all issues in checking compatibility of interfaces: signature compatibility, behavioural compatibility, and non-functional aspects of the interface such as performance issues.

The rest of the paper is structured as follows. Section 2 provides a brief overview of ODP. Section 3 focuses on the areas of ODP where types are introduced. Section 4 highlights the problems with types as they are presented in ODP. Section 5 puts forward a new and improved type model for composing systems. Section 6 shows how Z can be used to specify the type model suggested. Finally, section 7 draws some conclusions from the work and gives some acknowledgements. We assume the reader is familiar with the Z notation.

2 Overview of ODP

The ODP-RM (Reference Model) 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 uses an object-oriented approach. 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 ODP-RM itself is divided into four main parts:

Part 1 - Overview and Guide to Use : contains an overview and guide to use of the ODP-RM.

Part 2 - Foundations : contains the definition of concepts and gives the framework for description of distributed systems. It is here that the concept of type, and associated concept of subtype, are introduced.

Part 3 - Architecture : contains the specification of the required characteristics that qualify distributed system as open, *i.e.* constraints to which ODP systems must conform. One feature of Part 3 of concern to this paper is the idea of viewpoint languages. These represent abstractions of the system as a whole that focus on a particular concern, the intention being to simplify reasoning about the system as a whole. Of relevance to this paper is the computational viewpoint since it is here that types are used as a basis for composing (binding) sub-systems together.

Part 4 - Architectural Semantics : 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 paper represents ongoing work in this area.

3 Types in ODP

ODP introduces the concept of a type as a fundamental concept in Part 2 (ISO/IEC 1995*b*). It is defined as:

A predicate characterising a collection of $< X >$*'s. An* $< X >$ *is of the type or satisfies the type if the predicate holds for that* $< X >$*. In ODP types are needed for at least objects, interfaces and actions.*

A subtype is defined in Part 2 (ISO/IEC 1995*b*) as:

A type A is a subtype of a type B, and B is a supertype of type A, if every $< X >$ *which satisfies A also satisfies B.*

These elementary concepts are then used in the computational viewpoint to restrict interactions between objects. That is, they are used to enforce the legal bindings that can occur between objects, where a binding may loosely be considered as the composition of two (or more) interfaces. This is achieved through denoting interfaces with a type. This type captures (characterises) the features that manifest themselves in that interface. Features are manifest in ODP through syntactic aspects only. We shall show the limitations of this approach and the incorrect way it is applied in ODP.

3.1 Inadequacy of Current Model

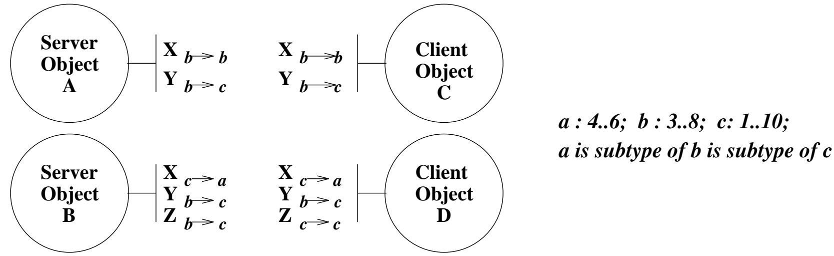
Basing interface type equivalence on structural and name equivalence of signatures results only in a "message is understood" policy of type checking. That is, if two signatures are type equivalent under these conditions then any invocation of an operation in an interface will at least be understood. The semantics of the operation might well be entirely different from that expected however. For example, the types **cowboy** and **rectangle** could be regarded as type equivalent if they both supported the single operation **.draw()**, even though the semantics (behaviour) of this operation is totally different in each case.

Thus structural and name equivalence of signatures only is not a sufficiently strong type checking policy to enable separate sub-systems (objects) to be composed with a guarantee of correct interworking.

3.2 Incorrectness of Current Model

Interfaces in the computational viewpoint can be operational, signal or stream. These interfaces have associated with them causalities. An operational interface possesses either a client or a server causality. This designation of causality represents an implicit assumption on the expected behaviour of that interface. A server performs functions requested by a client; a client requests functions to be performed by the server.

ODP does not distinguish between the clients and servers when establishing type relationships however. This is incorrect. Clause 7.2.4.3 of (ISO/IEC 1995*c*) states informally: "operation interface X is a signature subtype of operation interface Y when X possesses all of the operations of Y (and possibly more); the parameters of the operations in Y should be subtypes of the corresponding operation in X; and the results of operations of Y should be supertypes of the corresponding operations in X." The problems of this definition are illustrated using the following diagram.



The notation of **a⟶b** for servers means that the server accepts an input of type **a** and provides an output of type **b**. For clients **a⟶b** means that the client sends an output of type **a** and receives one of type **b**. We state that these types are ranges and classical type rules (Cardelli & Wegner 1985) apply to them, *i.e.* a subrange is a subtype.

An interface type relation is used in ODP for two different, but related, reasons: to substitute one interface for another and to bind interfaces together. We consider the problems in the ODP definition with regard to these two aspects.

3.2.1 Incorrectness in Substitution

Substitutability should be based on the environment being unaware of any difference in the substitute. Thus causality is essential here. It would be incorrect to substitute server object **A** with client object **C**, even though they support the same operations, since the client expects functions to be performed by other objects. Thus any clients bound to server object **A** initially, would not be serviced by client object **C**. Thus as far as substitutability is concerned, type checking requires like causalities. We thus propose extending the signature of interfaces to include a causality label.

Having established like causality, substitutability between clients and clients and servers and servers is also different. A client is an acceptable substitute for a second client when the operations it requests are a subset of the operations of the original client. The input parameters associated with the request are subtypes of the original parameters, and all result parameters of the substitute are supertypes of the original parameters. Thus it does not ask for anything the old client never asked for and everything it does ask for is understood by the server. In the diagram **C** is thus an acceptable substitute for **D** since it has less operations; the operation parameters of **C** are subtypes of the corresponding operations in **D**; and the results of those operations in **C** are supertypes of those in **D**.

A server is an acceptable substitute for another server when it possesses all the operations of the former and possibly more, and any request that could be dealt with by the original server may be dealt with by the substitute. This requires that the input parameters of the substitute server are supertypes of the original server and the return parameters of the substitute server are subtypes of the original server result parameters. In the diagram **B** is thus an acceptable substitute for **A** since it has more operations; the operation parameters of **A** are subtypes of the corresponding operations in **B**; and the results of those operations in **B** are subtypes of those in **A**.

Thus in effect the type relations between client substitutability and server substitutability are opposite with subsetting and supersetting of operations respectively. Hence in our example, **B** is type equivalent (a subtype) to **A** and **C** type equivalent (a subtype) to **D** based on a syntactic substitutability relationship. As can be seen, ODP only deals with the case of server interfaces and does not make a distinction between client and server substitutions.

3.2.2 Incorrectness in Binding

With regard to type relationships and binding, opposite causality is required between the interfaces. It would be meaningless to bind two clients together (or two servers) since they would both either expect to be served or expect to serve one another respectively. Given that opposite causality has been established, type relationship for binding requires that:

- the client asks for only a subset of the operations of the server;
- the requests made by the client are understood by the server;
- the responses issued by the server are understood by the client.

In the most simple form of binding (primitive) as given in clause 7.2.3.2 of (ISO/IEC 1995*c*), it is stated that interfaces can be bound provided they have complementary interface signatures. Complementary is defined as identical except for opposite causality. This is too strong a signature compatibility relation since it prohibits the interface of **C** being bound to the interface of **B** above, despite all possible client requests being catered for (understood) by the server and the server responses being understood by the client.

4 Type Equivalence

As a result of these considerations, it is apparent that the notion of causality is fundamental to the different aspects that have to be dealt with in establishing type equivalence. However, as shown in section 3.1, type equivalence based on signatures only is inadequate. We argue that type equivalence should be separated into three areas. Firstly, syntactic equivalence of signatures should be achieved, as done in most type systems currently used in distributed systems, *e.g.* Emerald (Black, Hutchinson, Jul, Levy & L.Carter 1987), CORBA (*The Common Object Request Broker Architecture and Specification: Revision 2.0* July 1995), DCE (Shirley 1993). As such, it is a well investigated area.

Once signature compatibility has been established, type checking should be extended to deal with behavioural aspects of types. This alone may not be enough to guarantee successful sub-system interworking, so it should be augmented with checks related to the non-functional equivalence of types. For example, two syntactically compatible interfaces that do not offer contradictory behaviours may not always be successfully bound together, *e.g.* if one produces outputs faster than the other can consume inputs then their interworking will be effected. To show how these three areas can be dealt with, we provide a Z specification of each area and how they can be unified in the context of the computational viewpoint of ODP. In particular we focus on operational interfaces of the computational viewpoint, specifically those dealing with interrogations, and non-functional aspects of these interfaces related to time.

4.1 Syntactic Aspects of Operational Interfaces

ODP prescribes rules for correct structuring of operational interfaces. These rules require certain naming rules to be satisfied for the labelling of operation and parameters of interface signatures. We firstly introduce a [*Name*] for things. The types existing in the system are denoted by [*TypeId*]. It should be pointed out that *TypeId* is likely to be more complex since there are many types that might exist in a given system, *e.g.* actions, interfaces and objects.

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 *Param* is introduced as an injective function from names to types.

$$| \; Param : Name \rightarrowtail\!\!\!\!\!\!\rightarrow TypeId$$

It is also useful to introduce sequences of these parameters.

$$PList == \mathrm{seq}\, Param$$

As stated earlier, interfaces in the computational viewpoint can be stream, operational or signal. These have a causality associated with the interface as a whole (operational) or with the individual actions associated with the interfaces (stream, signal). The causalities needed for consideration of the interfaces in this paper may be represented by:

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

An interrogation is represented by an invocation followed by a non-empty set of terminations. An invocation is represented by a name and the number, name and type of the argument parameters associated with the invocation.

```
┌─ InvTemplate ─────────────────────────────
│ invName : Name
│ inArgs : PList
└───────────────────────────────────────────
```

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

```
┌─ TermTemplate ────────────────────────────
│ termName : Name
│ outArgs : PList
└───────────────────────────────────────────
```

Having defined invocation and termination actions, an interrogation signature may be represented by:

```
┌─ IntSig ──────────────────────────────────
│ inv : InvTemplate
│ terms : 𝔽₁ TermTemplate
└───────────────────────────────────────────
```

The operational interface signatures under consideration here consist of sets of interrogations, and the interface as a whole is given a causality of 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:

$$
\begin{array}{l}
\textbf{OpIntSig} \\
\hline
ints : \mathbb{P}\, IntSig \\
role : Causality \\
\hline
role \in \{Client, Server\} \\
(\forall\, is_1, is_2 : IntSig; \\
\quad t_1, t_2 : TermTemplate; p_1, p_2 : Param \bullet \\
\qquad (is_1 \in ints \land is_2 \in ints \land is_1 \neq is_2 \Rightarrow \\
\qquad\qquad is_1.inv.invName \neq is_2.inv.invName) \land \\
\qquad (is_1 \in ints \land t_1 \in is_1.terms \land t_2 \in is_1.terms \land t_1 \neq t_2 \Rightarrow \\
\qquad\qquad t_1.termName \neq t_2.termName) \land \\
\qquad (is_1 \in ints \land \langle p_1 \rangle \text{ in } is_1.inv.inArgs \land \langle p_2 \rangle \text{ in } is_1.inv.inArgs \\
\qquad\qquad \land\ p_1 \neq p_2) \Rightarrow first\, p_1 \neq first\, p_2)
\end{array}
$$

4.2 Syntactic Compatibility for Binding

Two operational interfaces are syntactically equivalent and can be bound when they have opposite causalities and they do not issue requests or responses that are not understood. We thus introduce an elementary relationship between types based on substitution. For simplicity, we assume this relationship is based on sequences of parameters (types). The extension from type relationships to relationships between sequences of types is straightforward (Cardelli & Wegner 1985).

$$
isSubtype : PList \leftrightarrow PList
$$

Two interfaces with opposite causality can be bound successfully with regard to messages being understood when the invocation names of the client exist in the server and the parameters associated with the request are subtypes of those in the server interface. In addition, the termination names of the server must exist in the client and the parameters associated with the response are supertypes of those in the server interface. This can be represented by:

$$
\begin{array}{l}
BindSyntaxOk : OpIntSig \leftrightarrow OpIntSig \\
\hline
\forall\, ois_1, ois_2 : OpIntSig \mid (ois_1, ois_2) \in BindSyntaxOk \bullet \\
\quad ois_1.role = Client \land ois_2.role = Server \Rightarrow \\
\quad (\forall\, int_1 : IntSig \mid int_1 \in ois_1.ints \bullet (\exists\, int_2 : IntSig \bullet int_2 \in ois_2.ints \land \\
\qquad int_1.inv.invName = int_2.inv.invName \land (int_1.inv.inArgs, int_2.inv.inArgs) \in isSubtype \land \\
\qquad (\forall\, tt_2 : TermTemplate \mid tt_2 \in int_2.terms \bullet (\exists\, tt_1 : TermTemplate \bullet tt_1 \in int_1.terms \land \\
\qquad\quad tt_2.termName = tt_1.termName \land (tt_2.outArgs, tt_1.outArgs) \in isSubtype))))
\end{array}
$$

For brevity, we do not make this a symmetric relationship, *i.e.* we do not give here the case where the role of ois_1 is a server and ois_2 is a client. We also do not deal with higher order type systems, where the parameters of invocations can reference other interfaces. To specify this would require a check on the parameters and whether they were basic or referenced other interfaces. If they were basic then our simple subtype relation would suffice. If they referenced other interfaces, a recursive definition would be needed.

4.3 Syntactic Compatibility for Substitution

One server operational interface can be substituted for another when it understands all messages that could be sent to the original server, and the responses it gives will be understood by the recipient, *i.e.* the client.

$$SubstituteSyntaxOk : OpIntSig \leftrightarrow OpIntSig$$

$$\begin{array}{l}
\forall\, ois_1, ois_2 : OpIntSig \mid (ois_1, ois_2) \in SubstituteSyntaxOk \bullet \\
\quad ois_1.role = Server \land ois_2.role = Server \Rightarrow \\
\quad\; (\forall\, int_2 : IntSig \mid int_2 \in ois_2.ints \bullet (\exists\, int_1 : IntSig \bullet\; int_1 \in ois_1.ints \land \\
\qquad int_1.inv.invName = int_2.inv.invName \land (int_1.inv.inArgs, int_2.inv.inArgs) \in isSubtype \land \\
\qquad (\forall\, tt_2 : TermTemplate \mid tt_2 \in int_2.terms \bullet (\exists\, tt_1 : TermTemplate \bullet\; tt_1 \in int_1.terms \land \\
\qquad\; tt_2.termName = tt_1.termName \land (tt_2.outArgs, tt_1.outArgs) \in isSubtype))))
\end{array}$$

For brevity, we do not give here the conditions for client substitutions. As previously we deal with first order type systems where parameters do not reference other interfaces. The extension to deal with higher order cases is possible to specify, but omitted for brevity.

4.4 Behavioural Equivalence

In general, operation schemas are used to model actions in Z, *i.e.* an operation schema captures a before state and a state that must exist following the occurrence of this operation schema. Preconditions (the state before) and postconditions (the state after) can be calculated for the occurrence of a given operation schema.

In Z it is not normally the case that collections of actions are composed to form behaviours, where a behaviour here represents some sort of ordering relation between the operation schemas (actions). Rather, a Z specification is typically represented by a collection of unrelated[1] schemas acting upon some state, often represented itself by a schema.

This is purely a specification style, however, and Z due to its expressiveness can be used to model behaviours as might be given with other less abstract formal languages such as LOTOS (ISO/IEC 1989). With regard to this paper, the actions we consider can be either invocations or terminations. An action is more than simply a signature however. It is also a relation between a before and an after [*State*]. For simplicity, we model the state as a basic type, however it might well be represented by a schema containing the information that the object encapsulates, *e.g.* local variables and the predicates that apply to them. We may thus model actions generically by a parameterised free type definition which takes a signature, *e.g.* an invocation template, and two states as parameters. The first of these states represents the state that must exist for that action to fire, *i.e.* its precondition, and the second represents the state resulting from the firing of that action, *i.e.* the postcondition.

$$action ::= Inv\langle\langle InvTemplate \times State \times State\rangle\rangle \mid Term\langle\langle TermTemplate \times State \times State\rangle\rangle$$

Now a behaviour in effect is a collection of actions with a set of constraints on when they may occur. These constraints impose an irreflexive partial ordering on the set of actions. That is, the constraints form a transitive, irreflexive and anti-symmetric ordering relation on the set of actions. This may be represented by:

[1] In the sense that they do not prescribe an ordering relation between the operation schemas.

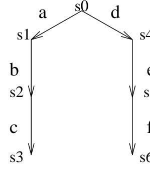
$$behspec == \{ar : action \leftrightarrow action \mid ar = ar^{+} \wedge ar \cap ar^{\sim} = \varnothing \wedge ar \cap \mathrm{id}\, action = \varnothing\}$$

Through reasoning about behaviour in this abstract manner, it is possible to consider compositions of behaviours more simply. This includes composition of interfaces and their associated behaviours.

The state of an object at a given time may then be represented by a sequence containing sets of state sequences, *i.e.* those states that the object has already been in.

$$history == \mathrm{seq}(\mathbb{F}_1(\mathrm{seq}\, State))$$

We illustrate briefly how the history is related to the behaviour specification through the following simple graph of actions with associated states.



Here the behaviour specification is given by $\{(a,b),(a,c),(b,c),(d,e),(d,f),(e,f)\}$ and the history is given by $\langle\{\langle s_0\rangle\},\{\langle s_0,s_1\rangle,\langle s_0,s_4\rangle\},\{\langle s_0,s_1,s_2\rangle,\langle s_0,s_4,s_5\rangle\},\{\langle s_0,s_1,s_2,s_3\rangle,\langle s_0,s_4,s_5,s_6\rangle\}\rangle$.

We may now represent the interface to a server object as a schema consisting of an operational interface signature, a behaviour specification and a history of states:

ServerInterface

$ois : OpIntSig$
$sbs : behspec$
$shistory : \mathrm{seq}(\mathbb{F}_1(\mathrm{seq}\, State))$

$ois.role = Server \wedge$
$(\forall is : IntSig; it : InvTemplate; tt : TermTemplate; s_1, s_2, s_3, s_4 : State \mid$
$\quad is \in ois.ints \wedge it = is.inv \wedge tt \in is.terms \wedge (Inv(it, s_1, s_2), Term(tt, s_3, s_4)) \in sbs \bullet$
$\qquad (\forall ss : \mathbb{F}_1(\mathrm{seq}\, State) \mid ss \in \mathrm{ran}\, shistory \bullet \langle s_1\rangle \in ss \wedge$
$\qquad (\forall ss_2 : \mathrm{seq}\, State \mid \langle s_1, s_2\rangle \frown ss_2 \in ss \bullet$
$\qquad\quad (\exists ss_3 : \mathrm{seq}\, State \mid ss_3 \in \bigcup(\mathrm{ran}\, shistory) \bullet ss_3 = \langle s_1, s_2\rangle \frown ss_2 \frown \langle s_3, s_4\rangle))))$

Here we state that the object should have a server role. It should always accept invocations, *i.e.* the precondition for invocations are always satisfied. We also state that for all invocations there must exist a finite sequence of states before the termination associated with that invocation can be issued, *i.e.* sequences of states that result in the state for the termination not being reachable are prohibited. It is possible to write more predicates here relating states to actions but for simplicity we do not do so. We model the structure of client objects similarly. For brevity, we do not include their associated predicates here.

ClientInterface

$ois : OpIntSig$
$cbs : behspec$
$chistory : \mathrm{seq}(\mathbb{F}_1(\mathrm{seq}\, State))$

$ois.role = Client$

A server and a client object may be bound successfully with regard to behavioural issues, when all requests a client might make are acceptable to the server and all responses issued by the server are understood by the client. The resultant behaviour is given by the transitive closure of the two behaviour specifications.

$$
\begin{array}{l}
\mathit{ClientServerBindBehOk} : \mathit{ServerInterface} \leftrightarrow \mathit{ClientInterface} \\
\hline
\forall\, si : \mathit{ServerInterface};\ ci : \mathit{ClientInterface} \mid (si, ci) \in \mathit{ClientServerBindBehOk} \bullet \\
\quad (si.ois, ci.ois) \in \mathit{BindSyntaxOk} \land (\exists\, res! : behspec \bullet res! = (si.sbs \cup ci.cbs)^{+})
\end{array}
$$

This axiomatic description shows the power of the Z language as opposed to other formal languages such as LOTOS. Here for a server and client to be bound together successfully, there must not exist contradictory behaviours. That is, if the server interface can perform action **a** followed by action **b** then the client interface cannot require action **b** to be performed before **a**. In LOTOS for example, this cannot be achieved within the specification itself. That is, such issues are considered when testing of the specification is done.

A server interface is an acceptable substitute for another server interface when it is syntactically compatible, *i.e.* it extends the signature. We also require that whatever sequence of states the original server could produce between an invocation and its associated termination then the substitute server cannot have a sequence of states that does not lead to these sequences arising. That is, it will not deadlock if the former does not deadlock. This in effect is the basis of the process algebra theory of **extension** (Brinksma & Scollo December 1986). This model of a history preserving extension is also consistent with the work presented in (Liskov & Wing 1995).

$$
\begin{array}{l}
\mathit{ServerServerSubBehOk} : \mathit{ServerInterface} \leftrightarrow \mathit{ServerInterface} \\
\hline
\forall\, si_1, si_2 : \mathit{ServerInterface} \mid (si_1, si_2) \in \mathit{ServerServerSubBehOk} \bullet \\
\quad (si_1.ois, si_2.ois) \in \mathit{SubstituteSyntaxOk} \land \\
\quad (\forall\, is : \mathit{IntSig};\ it : \mathit{InvTemplate};\ tt : \mathit{TermTemplate};\ s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8 : \mathit{State} \mid \\
\quad is \in si_1.ois.ints \cap si_2.ois.ints \land it = is.inv \land tt \in is.terms \land \\
\quad (\mathit{Inv}(it, s_1, s_2), \mathit{Term}(tt, s_3, s_4)) \in si_2.sbs \land (\mathit{Inv}(it, s_5, s_6), \mathit{Term}(tt, s_7, s_8)) \in si_1.sbs \bullet \\
\qquad (\forall\, ss : \mathrm{seq}\, \mathit{State} \mid \langle s_1, s_2 \rangle \frown ss \frown \langle s_3, s_4 \rangle \in \bigcup(\mathrm{ran}\, si_2.shistory) \bullet \\
\qquad\ \neg\, (\exists\, ss_2 : \mathrm{seq}\, \mathit{State} \mid \langle s_5, s_6 \rangle \frown ss_2 \in \bigcup(\mathrm{ran}\, si_1.shistory) \bullet \\
\qquad\quad (\forall\, ss_3 : \mathrm{seq}\, \mathit{State} \mid \langle s_5, s_6 \rangle \frown ss_3 \frown \langle s_7, s_8 \rangle \in \bigcup(\mathrm{ran}\, si_1.shistory) \bullet \neg\, (ss_2 \ \mathsf{in}\ ss_3)))))
\end{array}
$$

4.5 Non-functional Equivalence

There are many different sorts of non-functional equivalences that we might be interested in. For simplicity, we focus on the maximum number of state changes associated with a server interface following an invocation. This can then be used for example, to calculate the maximum time duration from the firing of an invocation to the firing of its associated termination, using the loose approximation that more state changes mean longer durations.

It might well be the case that the different interrogations associated with a server interface will have different prescriptions on the maximum non-functional aspects, *e.g.* state changes and hence timeliness requirements. For simplicity, we consider all of them at once.

$$
\begin{array}{|l}
maximumStateChanges : ServerInterface \times IntSig \rightarrow\!\!\!\!\!\!+ \mathbb{N} \\
\hline
\forall\, si : ServerInterface;\ is : IntSig \mid is \in si.ois.ints \bullet \\
\quad (\forall\, it : InvTemplate;\ tt : TermTemplate;\ s_1, s_2, s_3, s_4 : State \mid \\
\quad\ it = is.inv \land tt \in is.terms \land (Inv(it, s_1, s_2), Term(tt, s_3, s_4)) \in si.sbs \bullet \\
\qquad maximumStateChanges(si, is) = max\ \{ss_1 : \operatorname{seq} State \mid \\
\qquad\qquad head\ ss_1 = s_2 \land last\ ss_1 = s_4 \land \{ss_1\} \in \operatorname{ran} si.shistory \bullet \#ss_1\})
\end{array}
$$

It should be noted here that we take the sequence to start at s_2 since the invocation starts once state s_1 has been exited. We also assume that there is some form of fairness condition in the system. That is, if the precondition for an action is satisfied then it will eventually occur. This is necessary since invocations are always possible. It is also necessary to satisfy the proof obligation that the set cardinality operator is applied to finite sets.

Extending this to cover minimum state changes, and hence lower time duration limits requires only that the final predicate returns the minimum sequence. From this, upper and lower limits for the number of state changes from invocation to termination can be established. This information can subsequently be used to calculate time windows for interrogations, *i.e.* the upper and lower time limits for reception of a termination following an invocation.

5 Conclusions

This paper has shown the flaws in the current type system adopted by ODP and the limitations in type checking based on signatures only as found in most current distributed system approaches. Causality plays a central role in type checking at the signature level that has been glossed over in most work. The need to consider behavioural aspects of types is critical to achieving reliable interworking in distributed systems, since a signature is only a very weak substitute for behaviour. We have modelled behaviour here in a novel way that is not the usual Z approach and might initially be open to question. In effect we have a form of labelled transition system behaviour *ala* LOTOS. So are we simply re-inventing LOTOS semantics in Z? To answer this question, we list the advantages put forward in our work:

- we separate signature from behaviour (unlike in process algebras);
- we can introduce behavioural relations inside the specification instead of outside, *e.g.* when testing for conformance;
- we can also specify powerful relations that labelled transition systems do not traditionally offer, *e.g.* transitive closure.

This paper has also recognised that behavioural and syntactic considerations may not be enough to build reliable interworking systems. Non-functional aspects of types should also be checked, *e.g.* quality of service. We have shown how Z can be used to specify type checking at all three levels, *i.e.* syntactic, behavioural and non-functional.

It is likely that the issues surrounding this work will have to be addressed by the current standardisation activity of type management (ISO/IEC May 1996) so that interworking of distributed systems can be achieved successfully.

References

Black, A., Hutchinson, N., Jul, E., Levy, H. & L.Carter (1987), 'Distribution and abstract data types in Emerald', *IEEE Transactions on Software Engineering* **SE-13(1)**, 65–76.

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.

Brinksma, E. & Scollo, G. (December 1986), Formal notions of implementation and conformance in LOTOS, Technical Report INF-86-13, Dept. of Informatics, Twente University.

Cardelli, L. & Wegner, P. (1985), 'On understanding types, data abstraction, and polymorphism', *ACM Computing Surveys* **17**(4), 471–522.

ISO/IEC (1989), *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, ISO/IEC 8807, 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 Rec. X.904, ISO/IEC ITU-T, 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, Switzerland.

ISO/IEC (1995*f*), *Z Notation version 1.2*, ISO/IEC JTC1/SC22/WG19 CD 13568, ISO/IEC, Geneva, Switzerland.

ISO/IEC (May 1996), *Open Distributed Processing — Type Repository Function*, ISO/IEC JTC1/SC21 Working Document N10389, ISO/IEC ITU-T, Geneva, Switzerland.

Liskov, B. & Wing, J. (1995), Specifications and their use in defining subtypes, *in* J. P. Bowen & M. G. Hinchey, eds, 'Proceedings of the 9th International Conference of Z Users', Vol. 967 of *Lecture Notes in Computer Science*, Springer-Verlag.

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

Shirley, J. (1993), *Guide to Writing DCE Applications*, O'Reilly and Associates, Inc., Sebastopol CA, USA.

The Common Object Request Broker Architecture and Specification: Revision 2.0 (July 1995), Object Management Group, Inc., Framingham, MA.

University Library

MINERVA ACCESS

A gateway to Melbourne's research publications

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Sinnott, R. O.; Turner, K. J.

Title:
Type checking in open distributed system: a complete model and its Z specification

Date:
1997

Citation:
Sinnott, R. O., & Turner, K. J. (1997). Type checking in open distributed system: a complete model and its Z specification. In Open Distributed Processing and Distributed Platforms : proceedings of the IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms, Toronto, Canada.

Publication Status:
Published

Persistent Link:
http://hdl.handle.net/11343/28813

File Description:
Type checking in open distributed system: a complete model and its Z specification