INTERNATIONAL STANDARD

ISO/IEC 10746-4

First edition
1998-12-15

AMENDMENT 1
2001-12-15

Information technology — Open Distributed Processing — Reference Model: Architectural semantics

AMENDMENT 1: Computational formalization

Technologies de l'information — Traitement réparti ouvert — Modèle de référence: Sémantique architecturale

AMENDEMENT 1: Formalisation informatique

Reference number
ISO/IEC 10476-4:1998/Amd.1:2001(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Published by ISO in 2002

Printed in Switzerland

CONTENTS

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this Amendment may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Amendment 1 to International Standard ISO/IEC 10746-4:1998 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 7, *Software engineering*, in collaboration with ITU-T. The identical text is published as ITU-T Rec. X.904/Amd.1.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

INFORMATION TECHNOLOGY – OPEN DISTRIBUTED PROCESSING – REFERENCE MODEL: ARCHITECTURAL SEMANTICS

AMENDMENT 1

Computational formalization

1) Introduction

Replace the lst paragraph of the introduction

This Recommendation | International Standard is an integral part of the ODP Reference Model. It contains a formalisation of the ODP modelling concepts defined in ITU-T Rec. X.902 | ISO/IEC 10746-2, clauses 8 and 9. The formalisation is achieved by interpreting each concept in terms of the constructs of the different standardised formal description techniques.

with

This Recommendation | International Standard is an integral part of the ODP Reference Model. It contains a formalization of the ODP modelling concepts defined in ITU-T Rec. X.902 | ISO/IEC 10746-2, clauses 8 and 9 and in ITU-T Rec. X.903 | ISO/IEC 10746-3, clause 7 (Computational Language). The formalization is achieved by interpreting each concept in terms of the constructs of the different standardized formal description techniques.

2) Clause 1 – Scope

Replace the fourth bullet under The RM-ODP consists of

ITU-T Rec. X.904 | ISO/IEC 10746-4: **Architectural Semantics**: contains a formalisation of the ODP modelling concepts defined in ITU-T Rec. X.902 | ISO/IEC 10746-2, clauses 8 and 9, and a formalisation of the viewpoint languages of ITU-T Rec. X.903 | ISO/IEC 10746-3. The formalisation is achieved by interpreting each concept in terms of the constructs of the different standardised formal description techniques. This text is normative.

with

ITU-T Rec. X.904 | ISO/IEC 10746-4: **Architectural Semantics**: contains a formalization of the ODP modelling concepts defined in ITU-T Rec. X.902 | ISO/IEC 10746-2, clauses 8 and 9, and a formalization of the computational viewpoint language of ITU-T Rec. X.903 | ISO/IEC 10746-3. The formalization is achieved by interpreting each concept in terms of the constructs of the different standardized formal description techniques. This text is normative.

Replace the fourth paragraph

The purpose of this Recommendation | International Standard is to provide an architectural semantics for ODP. This essentially takes the form of an interpretation of the basic modelling and specification concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and the viewpoint languages of ITU-T Rec. X.903 | ISO/IEC 10746-3, using the various features of different formal specification languages. An architectural semantics is developed in four different formal specification languages: LOTOS, ESTELLE, SDL and Z. The result is a formalisation of ODP's architecture. Through a process of iterative development and feedback, this has improved the consistency of ITU-T Rec. X.902 | ISO/IEC 10746-2 and ITU-T Rec. X.903 | ISO/IEC 10746-3.

with

The purpose of this Recommendation | International Standard is to provide an architectural semantics for ODP. This essentially takes the form of an interpretation of the basic modelling and specification concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and the computational viewpoint language of ITU-T Rec. X.903 | ISO/IEC 10746-3, using the various features of different formal specification languages. An architectural semantics is developed in four different formal

specification languages: LOTOS, ESTELLE, SDL and Z. The result is a formalization of ODP's architecture. Through a process of iterative development and feedback, this has improved the consistency of ITU-T Rec. X.902 | ISO/IEC 10746-2 and ITU-T Rec. X.903 | ISO/IEC 10746-3.

Add the following paragraph at the end of Scope:

Annex A shows one way in which the computational viewpoint language of ITU-T Rec. X.903 | ISO/IEC 10746-3 can be represented in the formal languages LOTOS, SDL, Z and Estelle. This Recommendation | International Standard also makes use of the concepts defined in ITU-T Rec. X.902 | ISO/IEC 10746-2.

3) Clause 2 – Normative references

Change publication date for ITU-T Recommendation Z.100 *from* (1993) *to* (1999).

ISO/IEC 13568:

Add the following reference:

Z Notation, ISO/IEC JTC 1 SC 22 WG 19 Advanced Working Draft 2.C, July 13th 1999.

4) Subclause 3.2 – Definitions from ITU-T Recommendation Z.100

Replace the list with the following terms:

active, adding, all, alternative, and, any, as, atleast, axioms, block, call, channel, comment, connect, connection, constant, constants, create, dcl, decision, default, else, endalternative, endblock, endchannel, endconnection, enddecision, endgenerator, endnewtype, endoperator, endpackage, endprocedure, endprocess, endrefinement, endselect, endservice, endstate, endsubstructure, endsyntype, endsystem, env, error, export, exported, external, fi, finalized, for, fpar, from, gate, generator, if, import, imported, in, inherits, input, interface, join, literal, literals, map, mod, nameclass, newtype, nextstate, nodelay, noequality, none, not, now, offspring, operator, operators, or, ordering, out, output, package, parent, priority, procedure, process, provided, redefined, referenced, refinement, rem, remote, reset, return, returns, revealed, reverse, save, select, self, sender, service, set, signal, signallist, signalroute, signalset, spelling, start, state, stop, struct, substructure, synonym, syntype, system, task, then, this, timer, to, type, use, via, view, viewed, virtual, with, xor.

5) Subclause 3.3 – Definitions from the Z-Base Standard

Change subclause title to:

3.3 – Definitions from the Z Notation.

Replace the list with following terms:

axiomatic description, data refinement, hiding, operation refinement, overriding, schema (operation, state, framing), schema calculus, schema composition, sequence, type.

6) Annex A

Add a new Annex A as follows:

Annex A

Computational Formalization

A.1 Formalization of the Computational Viewpoint Language in LOTOS

A.1.1 Concepts

The formalization of the computational language in LOTOS uses the concepts defined in the formalization of the basic modelling and structuring rules given in ITU-T Rec. X.902 | ISO/IEC 10746-2 clauses 8 and 9.

Elementary Structures Associated with Operational and Signal Interfaces

To formalize the computational language in LOTOS it is necessary to introduce certain elementary structures. These include parameters that might be associated with certain computational interfaces and a basic model of information that might be used in a stream flow.

To formalize parameters it is necessary to introduce two concepts: names for things and types for things. Names are simply labels. As we shall see, the computational viewpoint requires that checks, e.g. for equality, are done on these labels when interfaces are constructed. We may represent names generally by:

```
type Name is Boolean
        sorts Name
        opns    newName: -> Name
                anotherName: Name -> Name
                _eq_,_ne_: Name, Name -> Bool
endtype (* Name *)
```

For brevity sake we omit the equations, which are expected to be obvious. It is possible to be more prescriptive here, e.g. using character strings from the LOTOS library. The only thing we are interested in regarding names is that we can determine their equality or inequality.

As discussed in this Recommendation | International Standard, a type in the ODP sense may not be interpreted directly in the process algebra part of LOTOS. It is however possible to model types through the Act One part of LOTOS. Unfortunately, whilst Act One was designed specifically for representing types, it is limited in the ways in which types and types relationships are checked. For example, it is not possible to check subtyping or equivalence up to isomorphism between types due to type equality in Act One being based on name equivalence of sorts. As a basis for reasoning here we introduce an elementary notion of types that allows us to test for equality, inequality and subtyping.

```
type AnyType is  Boolean
    sorts AnyType
    opns newType: -> AnyType
        anotherType: AnyType -> AnyType
        _eq_,_isSubtype_: AnyType, AnyType -> Bool
endtype (* AnyType *)
```

A parameter is a relation between a name and its underlying type representation. Thus a parameter may be represented by:

```
type Param is Name, AnyType
        sorts Param
        opns newParam: Name, AnyType -> Param
            _eq_,_ne_,_isSubtype_: Param, Param -> Bool
endtype (* Param *)
```

As previously, we require checks on the equality or inequality of parameters as well as when one parameter is a subtype of another. Two parameters are in a subtype relationship when their types are in a subtype relationship. It is also useful for us to introduce sequences of these parameters.

```
type PList is String actualizedby Param
        using sortnames PList for String Param for Element Bool for FBool
        opns _isSubtype_: PList, PList -> Bool
endtype (* PList *)
```

Here we use the type *String* from the LOTOS library actualised with the type *Param* defined previously. We also include an operation here *isSubtype* that can check whether one sequence of parameters is a subtype of another. One parameter list is a subtype of a second when all of the parameters it contains are subtypes of those found in the first. In addition the parameters should be in the same position in their respective lists. It should be noted that these parameters might contain references to interfaces used to restrict the interactions that can take place. Whilst it is quite possible to model an interface in the process algebra, it is not possible to model a reference to that interface in the process algebra that, loosely speaking, captures the functionality of that interface. To overcome this, we model interface references in Act One. Given that an interface reference captures, amongst other things, the signature of the interface, we provide an Act One model of signatures for operations. Operations consist of a name, a sequence of inputs and possibly a sequence of outputs. For simplicity's sake, we do not consider here whether the operation is of infix, prefix or suffix notation. This may be represented by:

```
type Op is Name, PList
        sorts Op
        opns makeOp: Name, PList -> Op
             makeOp: Name, PList, PList -> Op
             getName: Op -> Name
             getInps: Op -> PList
             getOuts: Op -> PList
             _eq_: Op, Op -> Bool
        eqns forall op1,op2: Op, n: Name; pl1, pl2: PList
ofsort Name     getName(makeOp(n,pl1,pl2)) = n;
ofsort PList    getInps(makeOp(n,pl1)) = pl1;
                        getInps(makeOp(n,pl1,pl2)) = pl1;
                        getOuts(makeOp(n,pl1)) = <>;
                        getOuts(makeOp(n,pl1,pl2)) = pl2;
ofsort Bool     op1 eq op2 = ((getName(op1) eq getName(op2)) and
                                        (getInps(op1) isSubtype getInps(op2)) and
                                        (getOuts(op2) isSubtype getOuts(op1)));
endtype (* Op *)
```

Having a method of determining whether two operations are the same reduces the problem of subtyping between abstract data types to a set comparison, where set elements are the created operations. Thus a server is a subtype of a second server if it supports all operations of the second server. We note here that we model two forms of operations: those that do not expect results and those that do expect results. We also introduce sets of these operations:

```
type OpSet is Set actualizedby Op
        using sortnames OpSet for Set Op for Element Bool for FBool
endtype (* OpSet *)
```

Now an interface reference may be represented by the following LOTOS fragment:

```
type IRef is OpSet
        sorts IRef
        opns makeIRef     :  OpSet -> IRef
             NULL         : -> IRef
             getOps       : IRef -> OpSet
             _eq_         : IRef, IRef -> Bool
        eqns forall o: OpSet; ir1, ir2: IRef
             ofsort OpSet   getOps(makeIRef(o)) = o;
             ofsort Bool    ir1 eq ir2 = getOps(ir1) eq getOps(ir2) ;
endtype (* IRef *)
```

Here we note that equality of interface references is based only on the operations contained in that reference. It might well be extended to cover other aspects, e.g. the location of the interface or constraints on its usage. We also introduce sets of these interface references.

```
type IRefSet is Set actualizedby IRef
        using sortnames IRefSet for Set IRef for Element Bool for FBool
endtype (* IRefSet *)
```

Elementary Structures Associated with Stream Interfaces

The computational viewpoint of ITU-T Rec. X.903 | ISO/IEC 10746-3 also considers interfaces concerned with the continuous flow of data, e.g. multimedia. These interfaces are termed stream interfaces. Stream interfaces contain finite sets of flows. These flows may be from the interface (produced) or to the interface (consumed). Each flow is modelled through an action template. Each action template contains the name of the flow, the type of the flow, and an indication of causality for the flow.

The computational viewpoint abstracts away from the contents of the flow of information itself. We consider here a generic idea of information flow where the flow of information is represented by a sequence of flow elements. A flow

element may be regarded as a particular item in the flow of information. We note here that flows are regarded in the computational viewpoint as continuous actions. In our model here we represent streams as sequences of discrete timed events. On the one hand this allows us to deal with the timing issues of information flows but we achieve this at the cost of losing the continuous nature of the flows.

Each flow element in an information flow can be considered as a unit consisting of data (this may be compressed) which we represent by *Data*. This model might include how the information was compressed, what information was compressed, etc. As such it is not considered further here. Flow elements also contain a time stamp used for modelling the time at which the particular flow element was sent or received. It is also often the case in multimedia flows that particular flow elements are required for synchronisation, e.g. synchronisation of audio with video for example. Therefore we associate a particular *Name* with each flow element. This can then be used for selecting a particular flow element from the flow as required. From this, we may model a flow element as:

```
type FlowElement is Name, NaturalNumber, Data, Param
sorts FlowElement
opns makeFlowElement: Data, Nat, Name -> FlowElement
     nullFlowElement  : -> FlowElement
     getData    : FlowElement -> Data
     getTime    : FlowElement -> Nat
     getName    : FlowElement -> Name
     toParam    : FlowElement -> Param
     setTime    : Nat, FlowElement -> FlowElement
eqns forall d: Data, s,t: Nat, n: Name
          ofsort Data       getData(makeFlowElement(d,t,n)) = d;
          ofsort Nat        getTime(makeFlowElement(d,t,n)) = t;
          ofsort Name       getName(makeFlowElement(d,t,n)) = n;
          ofsort FlowElement        setTime(s,makeFlowElement(d,t,n)) = makeFlowElement(d,s,n);
endtype (* FlowElement *)
```

It should be noted here that we model time as a natural number however it might well be the case that real (dense) time could be used, or time intervals. For simplicity here though, we restrict ourselves to discrete time represented as a natural number. We also introduce an operation that converts a flow element into a parameter. For simplicity we omit the associated equations. We also introduce sequences of these flow elements:

```
type FlowElementSeq is FlowElement
          sorts FlowElementSeq
          opns makeFlowElementSeq: -> FlowElementSeq
               addFlowElement: FlowElement, FlowElementSeq -> FlowElementSeq
               remFlowElement: FlowElement, FlowElementSeq -> FlowElementSeq
               getFlowElement: Name, FlowElementSeq -> FlowElement
               timeDiff: FlowElement, FlowElement -> Nat
          eqns forall f1, f2: FlowElement, fs: FlowElementSeq, n1,n2: Name
          ofsort FlowElementSeq
          getTime(f1) le getTime(f2) =>
                    addFlowElement(f1,addFlowElement(f2,makeFlowElementSeq)) =
                                                  addFlowElement(f2,makeFlowElementSeq);
          ofsort FlowElement
                    getFlowElement(n1,makeFlowElementSeq) = nullFlowElement;
                    n1 ne n2 =>
                              getFlowElement(n1,addFlowElement(makeFlowElement(d,t,n2),fs)) =
                                        getFlowElement(n1,fs);
                    n1 eq n2 =>
                              getFlowElement(n1,addFlowElement(makeFlowElement(d,t,n2),fs)) =
                                        makeFlowElement(d,t,n2);
endtype (* FlowElementSeq *)
```

For brevity we do not supply all of the equations. Flow elements are added to the sequence provided they have increasing timestamps. An operation is provided for traversing a sequence of flow elements to find a named flow element. We also introduce an operation to get the time difference between time stamps of two flow elements. It is possible using this operation to specify, for example, that all flow elements in a sequence are separated by equal time stamps. In this case we have an isochronous flow. We also introduce sets of these sequences of flow elements:

```
type FlowElementSeqSet is Set actualizedby FlowElementSeq
          using sortnames FlowElementSeqSet for Set FlowElementSeq for Element Bool for FBool
endtype (* FlowElementSeqSet *)
```

A.1.1.1 Signal

There is no inherent feature of LOTOS which can be used to distinguish between a signal, a stream flow and an operation. It may be the case, however, that a style of LOTOS can be used to distinguish between signals, streams and

operations. For example, all signals might have similar formats for their event offers. An example of one possible format for the server side of a signal is shown in the following LOTOS fragment.

<g> ?<sigName: Name> !<myRef> ?<inArgs: PList>;

Here and in the rest of A.1, we adopt the notation that <*X*> represents a placeholder for an *X*, i.e. *g*, *sigName*, *myRef* and *inArgs* represent placeholders for the gate, the name of the signal, the interface reference associated with the server offering this signal and the parameters associated with the signal respectively.

An example of one possible format for the client side of a signal is shown in the following LOTOS fragment:

<g> !<sigName> !<SomeIRef> !<inArgs>;

Here the client side of the signal contains a gate (*g*), a label for the signal name (*sigName*), a reference to the object the signal is to be sent to (*SomeIRef*) and the parameters associated with the signal (*inArgs*). We shall see in A.1.1.11 how these event offers may be used to construct signal interface signatures.

A.1.1.2 Operation

The occurrence of an interrogation or announcement.

A.1.1.3 Announcement

An interaction that consists of one invocation only. Due to the reasons given in A.1.1.1, only an informal modelling convention can be used to model announcements. One example of this for the client side of an announcement might be represented by:

<g> !<invName> !<SomeIRef> !<inArgs>;

The server side of an announcement might be represented by:

<g> ?<invName: Name> !<myRef> ?<inArgs: PList>;

The data structures here are similar to those in A.1.1.1. We shall see in A.1.1.12 how these event offers may be used to construct parts of operation interface signatures.

A.1.1.4 Interrogation

An invocation from a client to a server followed by one of the possible terminations from that server to that client. However, due to the reasons given in A.1.1.1, only an informal modelling convention can be used to model interrogations. One example of this for the client side of an interrogation might be represented by:

```
<g> !<invName> !<SomeIRef> !<inArgs> !<outArgs>;
( <g> ?<termName:Name> !<myRef> ?<outArgs: PList>;  (* ... other behaviour *)
  [ ]  (* ... other terminations *))
```

Here *termName* represents the termination names and *outArgs* represents the output parameters. The server side of an interrogation might be represented by:

```
<g> ?<invName: Name> !<myRef>  ?<inArgs: PList> ?<outArgs: PList>;
( <g> !<termName> !<SomeIRef>  !<outArgs>;  (* ... other behaviour *)
  [ ] (* ... other terminations *))
```

The other data structures here are similar to those in A.1.1.1. We shall see in A.1.1.12 how these event offers may be used to construct parts of operation interface signatures.

A.1.1.5 Flow

An abstraction of a sequence of interactions between a producer and a consumer object that result in the conveyance of information. Due to the reasons given in A.1.1.1, flows may only be represented in LOTOS through informal modelling conventions. It is often the case that flows have strict temporal requirements placed on them. One example of how this might be achieved for flow production is through a process that is parameterised by a sequence of data structures to be sent, e.g. flow elements that can be timestamped when they are sent. A simple example of how this might be modelled in LOTOS is:

```
process ProduceAction[ g, ...](... toSend: FlowElementSeq, tnow: Nat, rate: Nat ...):noexit:=
        g !<flowName> !<SomeIRef> !<SetTime(tnow+rate,head(toSend))>;
        (*... other behaviour and recurse with FlowElement removed from toSend *)
endproc (* ProduceAction *)
```

Here flow elements are sent together with the current (local) time plus the rate at which the flow elements should be produced.

Consumption of flow elements typically has different requirements placed upon it. The need to continually monitor the time stamps of the incoming flow of information is of particular importance. A simple representation of the consumption of an information flow may be represented by:

```
process ConsumeAction[ g,...](myRef: IRef, recFlowElements: FlowElementSeq, tnow, rate: Nat...) :noexit:=
        g ?<flowName: Name> !myRef ?<inFlowElement: FlowElement>;
        (* check temporal requirements of inFlowElement are satisfied then *)
        (* display FlowElement and recurse with time incremented *)
        (* or recurse with FlowElement added to received FlowElements and time incremented *)
endproc (* ConsumeAction *)
```

A.1.1.6 Signal Interface

As there is no direct means in LOTOS to distinguish formally between a signal and any other LOTOS event, establishing a given interface as being a signal interface is only possible informally by modelling the LOTOS events used to represent signals differently to any other event. An example of how a signal interface signature might be modelled in LOTOS is given in A.1.1.11.

A.1.1.7 Operational Interface

As there is no direct means in LOTOS to distinguish formally between an operation and any other LOTOS event, establishing a given interface as being an operational interface is only possible informally by modelling the LOTOS events used to represent operations differently to any other event. An example of how an operation interface signature might be modelled in LOTOS is given in A.1.1.12.

A.1.1.8 Stream Interface

As there is no direct means in LOTOS to distinguish formally between a flow and any other LOTOS event, establishing a given interface as being a stream interface is only possible informally by modelling the LOTOS events used to represent flows differently to any other event. An example of how a stream interface signature might be modelled in LOTOS is given in A.1.1.13.

A.1.1.9 Computational Object Template

In LOTOS a computational object template is represented by a process definition which has associated with it a set of computational interface templates which the object can instantiate; a behaviour specification, i.e. a behaviour expression that is not composed of events modelled as signal signatures, flow signatures or operation signatures. There should also be some form of environmental contract modelled as part of the process definition, however, LOTOS does not possess all of the necessary features to model environmental contracts fully. It may be possible to model some features in an environmental contract through an Act One data type. This should be given as a formal parameter in the value parameter list of the process definition.

A.1.1.10 Computational Interface Template

A signal interface template, a stream interface template or an operational interface template.

A.1.1.11 Signal Interface Signature

A signal interface signature is represented in LOTOS by a process definition, such that all event offers which require synchronisation with the environment in order to occur are modelled as signal signatures. The occurrence of these event offers result in a one-way communication from an initiating to a responding object. Structurally, a signal signature is similar to an invocation for an announcement (or a termination associated with an interrogation), i.e. it consists of a name (for the signal), a sequence of parameters associated with the signal and an indication of causality. Since all events in LOTOS are atomic, there is no inherent distinction between events modelled as announcements or signals.

Signal interface signatures differ from operational interface signatures though in that they do not require that the interface as a whole is given a causality. Instead, signal interface signatures may contain signals with either initiating or responding causalities. From this we model a signal interface signature in LOTOS by:

```
process SignalIntSig[ g... ](myRef: IRef, known: IRefs...):noexit:=
  g !<sigName> !<SomeIRef> !<pl>;                    ...(* other behaviour *)
    [ ]... (* other initiating actions *)
  [ ]
  g ?<sigName: Name> !myRef ?<inArgs: PList>;
          ([ not(makeOp(sigName,inArgs) IsIn getOps(myRef))] -> ...(* unsuccessful behaviour *)
          [ ]
          [ makeOp(sigName,inArgs) IsIn getOps(myRef) ] ->     ...(* successful behaviour *) )
  [ ]... (* other responding actions *)
endproc (* SignalIntSig *)
```

Here we state that a signal interface consists of a collection of event offers. These event offers may model either outgoing signals, i.e. those event offers with **!** prefixing the signal name and list of parameters, or incoming signals, i.e. those event offers with **?** prefixing the signal name and list of parameters. In the case of incoming signals, it is possible to check that the incoming signal is one expected, i.e. the signal is in the set of allowed signals associated with that interface reference.

NOTE – This specification fragment requires that the process is instantiated with at least one gate which corresponds to the interaction point at which the interface exists. The process should also be instantiated with a set of interface references and its own interface reference. We note here that it is not possible to write predicates on the signals sent. To do so would require a level of prescriptivity that we do not have, e.g. ensuring that SomeIRef is an interface reference that exists in the set of known interface references associated with the process. It is possible to perform checks on arriving signals though, i.e. the arriving signal should be one of the signals associated with that interface reference.We also note that we have used the choice operator here to model the composition of individual signals. It is quite possible to use several other composition operators here, e.g. interleaving. If interleaving composition is used then multiple arriving signals can be received before any responding signals are sent. Since interfaces usually have some form of existence, i.e. they offer operations that can be invoked more than one time, the components representing other behaviours are likely to contain recursive process instantiations. Through using the choice operator we have a form of blocking of signals, i.e. should a signal arrive then it has to be responded to before any other signals can be accepted. Similar arguments hold for all other processes representing computational interface signatures.

A.1.1.12 Operational Interface Signature

An operational interface signature is represented in LOTOS by a process definition, such that all event offers which require synchronisation with the environment in order to occur are modelled as part of operation signatures. That is, they all represent parts of either announcements or interrogations. We may model an operational interface signature for a client through the following process definition.

```
process OpIntSigClient[ g... ](myRef: IRef, known: IRefs, ...):noexit:=
  g !<invName> !<SomeIRef> !<inArgs>;                    ...(* other behaviour *)
   [ ]... (* other announcements *)
  [ ]
  g !<invName> !<SomeIRef> !<inArgs> !<outArgs>;          ...(* other behaviour *)
            (g ?<termName: Name> !myRef ?<outArgs: PList>;
              [ not(makeOp(termName,outArgs) IsIn getOps(myRef))] -> ...(* return error message *)
              [ ]
              [ makeOp(termName,outArgs) IsIn getOps(myRef)] ->      ...(* other behaviour *)
            [ ] ... (* other terminations *))
   [ ] ... (* other interrogations *)
endproc (* OpIntSigClient *)
```

Here we state that a client interface signature consists of a collection of event offers. These event offers may model either outgoing (announcement or interrogation) invocations, i.e. those event offers with **!** prefixing the invocation name and list of parameters, or incoming terminations, i.e. those event offers with **?** prefixing the termination name and list of parameters. In the case of incoming terminations, it is possible to check that the incoming termination is one expected, i.e. the termination is in the set of allowed termination associated with that interface reference.

The Note in A.1.1.11 also applies to operational interface signatures with the appropriate modifications, e.g. replace arriving signal by invocation.

Operational interfaces signatures for servers may be represented in LOTOS by:

```
process OpIntSigServer[ g... ](myRef: IRef, known: IRefs, ...):noexit:=
  g ?<invName: Name> !myRef ?<inArgs: PList>;
   ([ not(makeOp(invName,inArgs) IsIn getOps(myRef))] ->        ...(* ignore/other behaviour *)
   [ ]
   [ makeOp(invName,inArgs) IsIn getOps(myRef) ] ->            ...(* other behaviour *)
   [ ]... (* other announcements *))
 [ ]
  g ?<invName: Name> !myRef ?<inArgs:PList> ?<outArgs:PList>;    ...(* other behaviour *)
   ([ not(makeOp(invName,inArgs,outArgs) IsIn getOps(myRef))] -> ...(* return error message *)
   [ ]
   [ makeOp(invName,inArgs,outArgs) IsIn getOps(myRef) ] ->    ...(* other behaviour *)
    g !<termName> !<SomeIref> !resList ;                      ...(* other behaviour *)
    [ ] ... (* other terminations *))
 [ ] ... (* other interrogations *))
endproc (* OpIntSigServer *)
```

As with client interface signatures, a server interface signature has a set of known interface references and a reference for itself. This latter interface reference is used to ensure that the announcement or interrogation invocations the server receives are those that were expected, i.e. they were in the set of operations associated with that interface reference. If these invocations were not acceptable, e.g. the parameters were not correct or the operation requested was not available, then error handling behaviours are taken. In the case of announcements this might result in a recursive call with the

formal parameter list being unchanged. It is also possible to use a guard here to prevent the event from occurring in the first place. We do not do so since this might produce unwanted deadlocks in the specification. In the case of interrogations this would result in some form of error message being returned.

As with client operational interfaces it is possible to require that the messages received are those that were expected. It is not possible to have prescriptions on the messages sent though. It could be argued that this limitation is not necessarily a bad thing since, provided every process treats received messages the same way, sent messages should not cause deadlocks through their format not being understood for example.

A.1.1.13 Stream Interface Signature

A stream interface signature is represented in LOTOS by a process definition, such that all event offers which require synchronisation with the environment in order to occur are modelled as either producing or consuming flows. This might be represented in LOTOS as:

```
process StreamIntSig[ g... ](myRef: IRef, known: IRefSet, fss: FlowElementSeqSet...):noexit:=
        ConsumeAction[ g...](myRef, known, recFlowElements...) [ ]... (* other consume actions *)
        [ ]
        ProduceAction[ g...](myRef, known, FlowElementstoSend, ...) [ ]... (* other produce actions *)
endproc (* StreamIntSig *)
```

As with signal interfaces the notion of causality is applied to individual action templates in the stream interface signature. A stream interface signature contains sets of flows consuming or producing actions. Each flow signature is represented by a process. These processes contain the reference to the stream interface with which they are associated, a set of interface references representing the interface references known to that interface and a sequence of flow elements to send (in the case of producing flows) or receive (in the case of consuming flows). For brevity we do not specify how the set of sequences of flow elements that are passed to a stream interface signature are assigned to the producing flows in that interface. When instantiated all consume flows are of course empty.

The Note in A.1.1.11 also applies to stream interface signatures with the appropriate modifications, e.g. replace arriving signal by consumer flow.

A.1.1.14 Binding Object

An object which supports a binding between a set of other computational objects. An example of how this might be modelled is shown in the following LOTOS fragment:

```
process ServerInterface[ g ...](myRef: IRef, known: IRefSet, ...):noexit:=
    g ?bind: Name !myRef ?pl: PList;
        ([ getIRef(pl) IsIn known ] ->    ....(* already bound to server *)
          ServerInterface[ g...](myRef,known...)
        []
        [ not(getIRef(pl) IsIn known) and not(getOps(getIRef(pl)) IsSubsetOf getOps(myRef)) ]->
                ...(* operations requested by client not supported by server *)
        ServerInterface[ g ...](myRef,known...)
        [ not(getIRef(pl) IsIn known) and (getOps(getIRef(pl)) IsSubsetOf getOps(myRef)) ] ->
                ...(* successful behaviour *)
          ServerInterface[ g ...](myRef,Insert(getIRef(pl),known)...)
    []
    ...(* other behaviours restricted to clients in known *)
endproc (* ServerInterface *)
```

Here, if the client is already bound to the server, we then refuse the binding request and a recursive call is made. It should be noted that this need not necessarily be the case, i.e. the same client might be bound to the same server several times concurrently. Each of these bindings might have different properties associated with them, e.g. different sets of operations requested with different constraints. This would require that the server object returned different interface references for each successful binding. For example, instead of inserting the client interface reference into the set *known* for successful binding requests, the server might generate an interface reference which is sent to the client and added to the set *known*.

If the client is not already bound to the server, i.e. not in the set of *known* interface references, then the operations associated with the client's request are checked. If the operations asked for are not available at the server then some error behaviour is taken and a recursive call made. For simplicity we avoid dealing with the issues involved in type checking the parameters of client and server operations. For similar reasons we do not deal with environment contracts either. For brevity we do not provide the Act One operations for accessing the interface references contained in the parameter lists (*getIRef*). Rather, we simply state that the operations the client requests should be in the set of operations that the server provides.

Finally, if the client asks for legal operations, i.e. in the set of operations supported by the server interface reference then some successful behaviour occurs. This might be the sending of a positive response to the client. Following this the client interface reference is then added to the set of known interfaces. Membership of this set then allows access to the other behaviour (not specified here) available at this interface. This other behaviour should realise the set of operations and constraints given by the interface reference *myRef*.

A.1.2 Structuring Rules

A.1.2.1 Naming Rules

The naming rules contained in the computational language of ITU-T Rec. X.903 | ISO/IEC 10746-3 may only be supported in LOTOS provided strict modelling practices are followed. For example, when creating operational interface references, e.g. *myRef*, ensuring that the invocation, termination name are unique, as well as the parameter names associated with these operations. Enforcement of these rules can then be achieved through guards to detect the legality of the data structures received (via value passing) at that interface.

A.1.2.2 Interaction Rules

A.1.2.2.1 Signal Interaction Rules

It is always the case in LOTOS that an object offering a signal (or stream or operational) interface can only initiate (and respond to) signals, (or flows or operations) that are instances of the associated signal (or flow or operation) signature in its signal (or stream or operational) interface type. This is built in to the synchronisation rules of LOTOS, i.e. only event offers with matching (or overlapping) action denotations can synchronise. As discussed in this Recommendation | International Standard, there is no real notion of causality in LOTOS and events either happen instantaneously together or not at all. Thus it is not the case that an invocation is sent and then received. The invocation sending and receiving is represented by the occurrence of a single LOTOS event.

A.1.2.2.2 Stream Interaction Rules

See A.1.2.2.1.

A.1.2.2.3 Operation Interaction Rules

See A.1.2.2.1.

A.1.2.2.4 Parameter Rules

It is possible in LOTOS to use Act One sorts as computational interface identifiers. These identifiers may then be passed (via value passing) in the interactions of a given specification. Following these interactions, these identifiers (sorts) can be used in future event offers of the sender and the receiver objects, thereby allowing for interactions to occur between the sender and the receiver of the identifier. Thus the identifiers can be regarded as computational interface identifiers.

It is possible to have different representations of a given computational interface identifier modelled in Act One. This can be achieved by having some form of equality through Act One rewriting.

It is possible in LOTOS to ensure that computational interface identifiers identify the same computational interface.

A.1.2.2.5 Flows, Operations and Signals

If it is required that flows and operations are to be represented in terms of signals then this requires that appropriate modelling conventions in LOTOS are adopted. For example, through having checks (guards) on the names of the signals and associated operations or flows.

A.1.2.3 Binding Rules

A.1.2.3.1 Implicit Binding for Server Operation Interfaces

An example of the server side of implicit binding is given in A.1.1.14. An example of the client side of an implicit binding which wishes to invoke operation *SomeOp* offered by server referenced by *SomeIRef* and terminates upon completion might be represented in LOTOS as:

```
let myRef: IRef = makeIRef(insert(someOp,{})
in ClientInterface[g,...](myRef,Insert(SomeIRef,{},... )
        process ClientInterface[ g, ...](myRef: IRef, known: IRefSet, ...):noexit:=
            g !bind !SomeIRef !pl;                  (*  pl contains myRef, SomeIRef references server *)
            ( g ?reply: Name !myRef ?pl: PList;     (* receive successful bind response *)
              g !someOp !SomeIRef !pl2; stop)       (* invoke someOp (contained in myRef) and terminate *)
    endprocess (* ClientInterface *)
```

A.1.2.3.2 Primitive Binding Rules

Primitive binding can be achieved through adopting appropriate modelling conventions in LOTOS. An example of this is shown in A.1.1.14 and A.1.2.3.1. We note that the client side of primitive binding should not have to dynamically create the associated interface (and hence interface reference) as described in A.1.2.3.1. Rather, the interface should already exist.

A.1.2.3.3 Compound Binding Rules

Compound binding can be represented in LOTOS through adopting appropriate modelling conventions. These require that a process (representing a binding object) is specified that accepts (via value passing) collections of interface references representing the objects wishing to be bound together. Once all references have been received, the binding object issues a request to create instances of all of these referenced interfaces to the sender of the respective original bind requests. These created interfaces are then bound (using primitive binding as shown in subclauses A.1.1.14 through A.1.2.3.1) to one another. Once binding has taken place, processes are instantiated (created) within the binding object that can subsequently be used to control the interactions of the objects involved in the compound binding.

A.1.2.4 Type Rules

A.1.2.4.1 Subtyping Rules for Signal Interfaces

The subtyping rules for signal interfaces can be achieved in LOTOS through ensuring that certain modelling conventions are followed. Examples of these conventions are shown in A.1.1.11 and A.1.1.14.

A.1.2.4.2 Subtyping Rules for Stream Interfaces

The subtyping rules for stream interfaces can be achieved in LOTOS through ensuring that certain modelling conventions are followed. Examples of these conventions are shown in A.1.1.13 and A.1.1.14.

A.1.2.4.3 Subtyping Rules for Operational Interfaces

The subtyping rules for operation interfaces can be achieved in LOTOS through ensuring that certain modelling conventions are followed. Examples of these conventions are shown in A.1.1.12 and A.1.1.14.

A.1.2.5 Template Rules

A.1.2.5.1 Computational Object Template Rules

A computational object can:

- initiate or respond to signals by having event offers in its associated behaviour expression modelled as signal signatures with the appropriate causality;
- produce or consume flows by modelling flow signatures in its associated behaviour expression;
- invoke or terminate operations by having event offers modelling interrogation and announcement signatures in its behaviour expression;
- instantiate interface or object templates by having interface and object templates as part of its behaviour expression;
- bind interfaces by having a behaviour expression that will enable binding between interfaces to occur;
- access and modify its state through events occurring which make up part of its behaviour expression;
- stop (delete) itself by providing a suitable LOTOS termination as part of its behaviour expression, e.g. stop, exit or [>;
- spawn, fork and join activities by using combinations of different LOTOS composition operators in its associated behaviour expression, e.g. choice, interleaving and parallel composition;
- bind to a trading function by offering event offers that can synchronise with a trading function as part of its behaviour expression. This implies that the LOTOS specification containing the object contains a trader specification also. It should be pointed out that the LOTOS event that may represent a trading action may not be different to any other LOTOS event. That is, the distinction between a trading action and any other action is merely that the event is between the trader and the object. Therefore a trading action will be distinguishable from any other action only through its action denotation parameters in LOTOS. Hence Act One sorts used as identifiers need to be carefully dealt with to be able to distinguish between the different actions.

A.1.2.5.2 Computational Interface Template Instantiation

Whilst it is the case that computational interface template instantiation creates a new computational interface in LOTOS, it is not the case that computational identifiers are established for these created interfaces also. It is possible to achieve this however, provided certain modelling conventions are adopted. For example, through engineering interface references through Act One data structures which are created when the associated interfaces (LOTOS processes) are created.

A.1.2.5.3 Computational Object Template Instantiation

Instantiation of a computational object template in LOTOS is achieved through instantiating the associated process definition representing the object template. References to the interfaces created in the object instantiation process have to be engineered as discussed in A.1.2.5.2.

A.1.2.6 Failure Rules

The modelling of failures in LOTOS may be achieved to a certain extent by giving all possible behaviours for a given system. That is, successful behaviours and failed behaviours. This means, however, that all possible behaviours were known beforehand which depending on the failure type, is not always possible. As such this method of modelling failures is limited in that failures here are predictable, whilst in general this will not be the case.

The infrastructure failures which can occur during interaction (i.e. binding, security, communication and resource) may all be modelled to a certain extent in LOTOS by giving all possible system behaviours.

A.1.2.7 Portability Rules

LOTOS supports all of the portability rules of ITU-T Rec. X.903 | ISO/IEC 10746-3, however, signature subtype checking and binding to trading interfaces is only possible if modelling conventions are adopted that allow the signature of interfaces in Act One to be represented and used as a basis for type checking and subsequent binding operations.

It should be noted that there is no real notion of a given action in LOTOS being a fork or a join action. It is only when considering the specification behaviour that fork and join actions can be identified.

A.1.2.8 Conformance and Reference Points

A LOTOS specification consists of a system of possible behaviours. As such, it is not possible to directly identify reference points which may become either programmatic, perceptual, interchange or interworking conformance points. A specification contains all its reference points inbuilt and it is up to the implementor of the specification to identify, label and test reference points. Thus a testing process acting on a LOTOS specification may be restricted to a certain part of the specification, i.e. a given object or interface. The identification of this object or interface as a conformance point is part of the testing process, however, and not part of the specification process.

There are many different types of conformance possible within LOTOS. These all relate the behaviour of the specification to some form of expected behaviour. Thus a given specification is said to conform if it exhibits the correct behaviour, i.e. the behaviour expected in the testing process.

A.2 Formalization of the Computational Viewpoint Language in SDL

An ODP system (applications, ODP functions) is described in SDL from the computational viewpoint as a configuration of computational objects. These computational objects are instances of block types and process types. These (SDL-) types have to be derived from the process types and block types defined in this contribution. The computational objects interact using an infrastructure offering the ODP functions. The computational model of the infrastructure is modelled by an SDL-block. This provides all ODP-functions visible at the computational viewpoint as SDL remote procedures.

Using the formal description technique SDL this subclause shows how the concepts and rules of the computational language can be expressed. For concepts not fully covered, the use of informal text is proposed. *Italics* are used to distinguish between ODP and SDL terms.

An ODP system is described in SDL from the computational viewpoint as a configuration of computational objects. These computational objects are instances of block and process types.

A.2.1 Concepts

The concepts of the computational language are expressed in SDL-92 according to ITU-T Recommendation Z.100 and ITU-T Recommendation Z.105 and according to the generic definitions, rules and guidelines of this Recommendation | International Standard.

A.2.1.1 Signal

A signal may be represented by the occurrence of an *OUTPUT* <*sdl-signal*> action and the reception of that <*sdl-signal*> (or a related <*sdl-signal*> containing the information of the original <*sdl-signal*>) by the *inputport* of the receiving *PROCESS*.

NOTE – A signal is an atomic action. Although it is modelled in SDL by a series of (SDL-) actions, atomicity is guaranteed because the transmission by a channel and the reception through the receivers inputport are implicit. An output may be instantaneous in case that initiator and responder are connected by no-delay channels or by signalroutes.

A.2.1.2 Operation

An operation is the occurence of an interrogation or an announcement.

A.2.1.3 Announcement

An announcement is a sequence of actions, modelled by the occurence of the following actions:

- *OUTPUT* of an <*sdl-signal*> by an interface *PROCESS* of a client object
- INPUT of that <*sdl-signal*> or a related <*sdl-signal*> containing the information of the original <*sdl-signal*>, by an interface *PROCESS* of the server object (**Invocation**).

NOTE – The start of the function to be performed by the server is modelled by the transition triggered by INPUT.

The structure of an announcement is given in Table A.1.

Table A.1 – Announcement (Client and Server side)

PROCESS TYPE Client INHERITS OperationInterface; ... OUTPUT servicel(p1,p2,p3) TO Serverl ... ENPROCESS TYPE Client	PROCESS TYPE Server INHERITS OperationInterface; ADDING GATE InterfaceSignature ADDING IN WITH servicel; ... STATE bound; INPUT service1(a1,a2,a3); TASK 'perform required function'; NEXTSTATE-; ... ENDPROCESS TYPE Server

A.2.1.4 Interrogation

An interaction between a client object and a server object that consists of:

- *OUTPUT* of an <*sdl-signal*> by an interface *PROCESS* of a client object;
- *INPUT* of that <*sdl-signal*> (or a related <*sdl-signal*> containing the information of the original <*sdl-signal*>) by an interface *PROCESS* of the server object (**Invocation**) followed by;
- possible execution of the requested function;
- *OUTPUT* of an <*sdl-signal*> by an interface *PROCESS* of the server object; and
- *INPUT* of that <*sdl-signal*> (or a related <*sdl-signal*> containing the information of the original <*sdl-signal*>) by an interface *PROCESS* of the client object (**Termination**).

The structure of an interrogation is given in Table A.2.

Table A.2 – Interrogation using SDL signals

```
PROCESS TYPE Client                          PROCESS TYPE Server
INHERITS OperationInterface;                 INHERITS OperationInterface;
...                                                ADDING GATE InterfaceSignature
OUTPUT servicel(p1,p2,p3) TO Serverl;              ADDING IN WITH servicel;
NEXTSTATE waitService1;                      ...
STATE waitService1                           STATE bound;
  INPUT service1Term1;                             INPUT service1(a1,a2,a3);
    ...                                            TASK 'perform required function';
  INPUT service1Term2;                             OUTPUT service1term1 TO SENDER;
                                                     NEXTSTATE-;
    ...
  SAVE*;                                     ...
...                                          ENDPROCESS TYPE Server
ENPROCESS TYPE Client
```

Synchronous interrogations can be modelled by value returning *REMOTE PROCEDURES*. The client calls a *REMOTE PROCEDURE* of the interface *PROCESS* of the server object. The SDL replacement model for *REMOTE PROCEDURES* ensures the sequenciality requirements of the interrogation.

The structure of a synchronous interrogation is given in Table A.3.

Table A.3 – Interrogation using SDL remote procedures

```
PROCESS TYPE Client                          PROCESS TYPE Server
INHERITS OperationInterface;                 INHERITS OperationInterface;
IMPORTED PROCEDURE service1;                 ADDING
...                                          EXPORTED PROCEDURE service1 REFERENCED;
CALL servicel(par1,par2,par3) TO Serverl;    ...
...                                          STATE bound;
ENPROCESS TYPE Client                          INPUT PROCEDURE service1(p1,p2,p3);
                                                     NEXTSTATE-;

                                             ...
                                             ENDPROCESS TYPE Server
```

A.2.1.5 Flow

A flow is an abstraction of a set of interactions. It is modelled in SDL at the producer side by a *continuous signal* as shown in Table A.4.

Table A.4 – Flow (signal based)

```
/* Producer*/                                /* Consumer*/

...                                          ...
PROVIDED Available(Data);                    STATE Receive;
  OUTPUT Frame(GetFrame(Data)) TO BoundTo;           PRIORITY INPUT Frame(Data);
  NEXTSTATE-;                                              ...;
...                                                  NEXTSTATE-;
                                                     INPUT NONE;

                                                         ...;
                                                     NEXTSTATE-;
```

A further abstraction may be based on the concept of *IMPORTED* and *EXPORTED* values.

A.2.1.6 Signal Interface

A *PROCESS* instance, which communicates only with (SDL-) *SIGNALS* via non-delaying *CHANNELS*. The *PROCESS* is an instantiation of a subtype of a SignalInterfaceTemplate.

A.2.1.7 Operational Interface

An interface in which all interactions are operations. The *PROCESS* is an instantiation of a subtype of an OperationalInterfaceTemplate.

A.2.1.8 Stream Interface

A *PROCESS* instance which communicates only with (SDL-) *SIGNALS* via *CHANNELS* or through *EXPORTED* or *IMPORTED* variables. The *PROCESS* is an instantiation of a subtype of a StreamInterfaceTemplate.

A.2.1.9 Computational Object Template

An object template for a Computational Object. It is represented in SDL as a type based *BLOCK* definition, where the *BLOCK TYPE* is a specialization of the *BLOCK TYPE* ComputationalObjectTemplate. The concept of environment contract is not supported in SDL, informal text has to be used instead.

All ingoing/outgoing gates of a ComputationalObjectTemplate have to be connected to instantiations of subtypes of the InterfaceTemplate processes.

The structure of a *BLOCK TYPE* computational object template is given in Table A.5.

Table A.5 – BLOCK TYPE ComputationalObjectTemplate

```
BLOCK TYPE ComputationalObjectTemplate;
VIRTUAL PROCESS TYPE SignalInterfaceTemplate REFERENCED;
VIRTUAL PROCESS TYPE StreamInterfaceTemplate REFERENCED;
VIRTUAL PROCESS TYPE OperationInterfaceTemplate REFERENCED;
VIRTUAL PROCESS TYPE BehaviourTemplate REFERENCED;
PROCESS LocalBehaviour(1,) : BehaviourTemplate;
/*
        GATE Definitions
        SIGNALROUTE Definitions
        PROCESS Definitions
        have to be added in specializations of this TYPE
/*
ENDBLOCK TYPE;
```

A.2.1.10 Computational Interface Template

A signal interface template, stream interface template or operational interface template. The concept of environment contract is not supported in SDL, informal text has to be used instead.

A SignalInterfaceTemplate is represented by a type based *PROCESS* definition, where the *PROCESS TYPE* is at least a specialization of the *PROCESS TYPE* signal interface template, as shown in Table A.6. The *PROCESS TYPE* must have only one *GATE* connected to the outside of the surrounding *BLOCK*. This *GATE* represents the signal interface signature. All communication to the outside of the *BLOCK* must be based on *SIGNAL* exchange through this *GATE*. There must not exist any *OUTPUT* to the outside of the surrounding *BLOCK* in the *STATE* unbound. The behaviour is specified by the process graph in Table A.6.

Table A.6 – PROCESS TYPE SignalInterfaceTemplate

```
PROCESS TYPE SignalInterfaceTemplate;
GATE InterfaceSignature IN FROM ATLEAST SignalInterfaceTemplate
        OUT TO ATLEAST SignalInterfaceTemplate;
DCL BoundTo PId;
                START VIRTUAL; NEXTSTATE unbound;
                STATE unbound;  INPUT VIRTUAL Bind(BoundTo); NEXTSTATE bound;
                                INPUT VIRTUAL *;   NEXTSTATE -;
                STATE bound;    INPUT VIRTUAL UnBind;   NEXTSTATE unbound;
                                INPUT VIRTUAL *;   NEXTSTATE -;
                STATE*;         INPUT VIRTUAL Delete;  STOP;
ENDPROCESS TYPE;
```

A Stream Interface Template is represented by a type based *PROCESS* definition, where the *PROCESS TYPE* is at least a specialization of the *PROCESS TYPE* StreamInterfaceTemplate, as shown in Table A.7. The *PROCESS TYPE* must have only one *GATE* connected to the outside of the surrounding *BLOCK*. This *GATE* represents the Stream Interface Signature. All communication to the outside of the *BLOCK* must be based on *SIGNAL* exchange through this *GATE* by *continuous signals* (flows) in the *STATE* bound.

The behaviour is specified by the process graph in Table A.7.

Table A.7 – PROCESS TYPE StreamInterfaceTemplate

```
PROCESS TYPE StreamInterfaceTemplate;
        INHERITS SignalInterfaceTemplate
        GATE InterfaceSignature ADDING IN FROM ATLEAST BinderStreamInterfaceTemplate
                OUT TO ATLEAST BinderStreamInterfaceTemplate;
ENDPROCESS TYPE;
```

An Operation Interface Template is represented by a type based *PROCESS* definition, where the *PROCESS TYPE* is at least a specialization of the *PROCESS TYPE* OperationInterfaceTemplate, as shown in Table A.8. The *PROCESS TYPE* must have only one *GATE* connected to the outside of the surrounding *BLOCK*. This *GATE* represents the Operation Interface Signature. All communication to the outside of the *BLOCK* must be based on *SIGNAL* exchange through this *GATE* or by *REMOTE PROCEDURES*. The behaviour is specified by the *PROCESS* body in Table A.8.

Table A.8 – PROCESS TYPE OperationInterfaceTemplate

```
PROCESS TYPE OperationInterfaceTemplate;
        INHERITS SignalInterfaceTemplate
        GATE InterfaceSignature ADDING IN FROM ATLEAST OperationInterfaceTemplate
                OUT TO ATLEAST OperationInterfaceTemplate;
ENDPROCESS TYPE;
```

A.2.1.11 Signal Interface Signature

The signature of a Signal Interface is given by a *GATE* definition of the corresponding *PROCESS*. This contains the set of names of all *SIGNALS* sent/received by the Interface *PROCESS* and gives an indication of their causality (*IN WITH* or *OUT WITH* respectively). A Signal Signature is given by a *SIGNAL* definition, comprising:

- a name for the Signal;
- the number and types of the parameters for the Signal.

SDL does not allow for a naming of parameters, however parameters are identified by their position.

Additionally the *SIGNALSET* of the *PROCESS* contains the set of names of all *SIGNALS* the *PROCESS* can receive.

A.2.1.12 Operation Interface Signature

An Interface Signature for an operational Interface, comprises the signature of each operation in the interface. It is given by *GATE* definitions according to A.2.1.10 or by *REMOTE* and *IMPORTED* or *EXPORTED PROCEDURE* definitions (in case of *REMOTE PROCEDURES*).

The announcement signature is, in case of REMOTE PROCEDURE, reflected by the *PROCEDURE* signature, otherwise it is given by a *SIGNAL* definition.

The termination signature is, in case of *REMOTE PROCEDURES*, reflected by the signature of the data type of the *PROCEDURE* return value, otherwise it is given by a *SIGNAL* definition.

A.2.1.13 Stream Interface Signature

The signature of a Stream Interface is given by a *GATE* definition, containing the set of names of all *SIGNALS* sent/received by the Interface *PROCESS*, i.e. the flows, and an indication of causality as given according to A.2.1.10.

Additionally the *SIGNALSET* of the *PROCESS* contains the set of names of all *SIGNALS* the *PROCESS* can receive.

In case of remote procedures, these have to be specified in the interface *PROCESS* as *EXPORTED* or *IMPORTED*. The *REMOTE* clause restricts the visibility of *EXPORTED PROCEDURES*.

NOTE – Complementary interface signatures are indicated by *GATE* definitions with identical signal lists but complementary direction clauses (*IN WITH* and *OUT WITH* respectively).

A.2.1.14 Binding Object

An instance of a *CHANNEL*. The internal structure of a binding object is specified by a *CHANNEL SUBSTRUCTURE*.

More complex binding objects should be represented by a configuration of two or more *CHANNELS* and one or more *BLOCK* -instances.

NOTE – Some *CHANNELS* in SDL are given implicitly, e.g. *CHANNELS* supporting the communication via *EXPORTED PROCEDURES* and *EXPORTED* values. The use of a configuration consisting of a *BLOCK* and two or more *CHANNELS* allows for bindings between more than two interfaces and for bindings with control interface(s).

A.2.2 Structuring Rules

A computational specification in SDL describes the functional decomposition of an ODP system in distribution transparent terms, as:

- a configuration of *BLOCK*S and *CHANNELS*;
- internal actions are modelled by local *PROCESSES* which do not communicate to the outside of the *BLOCK*;
- interactions are modelled by Interface *PROCESSES*.

A computational specification in SDL is constrained by the rules of the computational language and by the semantics of SDL.

The initial set of computational objects is given by the set of *BLOCK*S and the *PROCESSES* contained in them. An initial number may be specified for each different kind of *PROCESS*. The changes in the configuration are expressed in the behaviour specification.

A.2.2.1 Naming Rules

The static semantics of SDL ensures the following rules:

- signal names are distinct in any signal interface signature because a *SIGNAL* may appear only once in a *GATE* definition;
- operation names are distinct in any operation interface signature, because a *PROCESS* must not export or import two different *PROCEDURES* with the same name;
- parameters are uniquely identified by their position. There is no naming for *SIGNAL* parameters;
- computational interface identifiers are mapped to *PIds* (*PROCESS* identifiers) and are therefore unique throughout the specification.

A.2.2.2 Interaction rules

To fulfil the constraint that interactions at an unbound interface cause an infrastructure failure, all *OUTPUT* actions must be qualified with *VIA* and/or *TO* clauses. All *SIGNALS* sent outside a Computational Object *BLOCK* have to be sent by an Interface *PROCESS* and routed through a *GATE* of that *PROCESS*.

All *Remote Procedure Calls* must be qualified with TO. All failures have to be explicitly specified, since SDL does not provide a mechanism for handling failures. Behaviour is undetermined after an error has occurred.

A.2.2.2.1 Signal Interaction rules

The Signal interaction rules are guaranteed by the SDL semantics. A Signal Interface *PROCESS* can only send/receive those *SIGNALS* that are specified for that *PROCESS*.

A.2.2.2.2 Stream interaction rules

Stream interfaces are based on SIGNAL exchange, therefore the stream interaction rules are guaranteed by the SDL semantics. A Stream Interface *PROCESS* can only produce/consume those flows that are specified for that *PROCESS*.

A.2.2.2.3 Operation interaction rules

In case of Remote PROCEDURES the Operation Interaction Rules are guaranteed by the SDL semantics. In case of SIGNALS the specifier has to apply the following rules:

- for all *SIGNALS* modelling the operation invocations there must be transitions in all states of the *PROCESS* to handle these *SIGNALS* (no implicit discard!);
- the transition or sequence of transitions triggered by the invocation must end with exactly one *OUTPUT* <*sdl-signal*>, where <*sdl-signal*> is a *SIGNAL* modelling one of the terminations.

The order of occurrence of invocation/termination deliver signals of concurrent invocations/terminations does not necessary follow the order of occurrence of the corresponding invocation/termination submit signals. Concurrent *SIGNALS* are ordered in SDL in arbitrary (non-deterministic) order. There is no means to describe the duration of an operation directly, the duration is arbitrary.

A.2.2.2.4 Parameter rules

Computational interface identifiers are represented as *PIds*. They can be both argument and result parameters in signal and operation interactions. The identifiers can be passed as a parameter in further interactions.

The recipient of a computational interface identifier can use the identifier to engage in interactions with the object supporting the interface, provided a binding can be established between the interfaces.

Computational identifiers are unambiguous within the *SYSTEM* specification. *SYNONYMS* may be used to give an interface more than one identifier.

It is always possible to determine whether two computational interface identifiers identify the same computational interface, however, there is no means to qualify a PId with an interface signature type or to detect the type of the interface the *PId* references.

A.2.2.2.5 Flows, Operations and Signals

The replacement model of operations and streams by signals is guaranteed in SDL. In case of *REMOTE PROCEDURES* and *EXPORTED/IMPORTED* a similar replacement model is provided by the SDL standard.

A.2.2.3 Binding rules

Interaction between computational objects is only possible when their interfaces are bound to the same binding object. This is ensured by the SDL semantics since each communication between BLOCKS in SDL requires the *BLOCK*S to be connected by a *CHANNEL*. Each *OUTPUT* shall use the *TO* <*PId*> or *VIA* <*channel*> or *VIA* <*gate*> clause.

A.2.2.3.1 Implicit Binding for Server Operation Interfaces

Implicit binding is possible in SDL for computational object *BLOCKS* which are directly connected by a *CHANNEL*. Implicit binding always takes place with operations modelled with *REMOTE PROCEDURES*. The SDL replacement model guarantees the implicit binding rules for server operation interactions. However, the client operation interface has to be created explicitly. The scope of an *EXPORTED PROCEDURE* is restricted by a *REMOTE* clause.

A.2.2.3.2 Primitive binding rules

Primitive binding requires that the interface *PROCESSES* of the two computational objects are directly connected by a *CHANNELS* as shown in Table A.9.

Table A.9 – Binding Action

```
/*Initiater*/

...
STATE unbound;
...
TASK
Destination:=CALL Bind(SELF,TDesc) TO Dest;
DECISON Destination=Dest;
FALSE: /*Binding Failure*/
        NEXTSTATE -;
TRUE: TASK BoundTo:= add(BoundTo,Dest)
        NEXTSTATE Bound;
ENDDECISION;
STATE Bound;
...
```

```
/*Destination*/
VIRTUAL EXPORTED PROCEDURE Bind
NEWTYPE PIDSet ATLEAST
        OPERATORS noequality;
                add: PIDSet,PId->PIDSet;
                remove: PIDSet,PId->PIDSet;
ENDNEWTYPE;
DCL BoundTo PIDSet,ThisType TypeDescrType, Failure Boolean>>
FPAR Source PId, Typ TypeDescrType;
RETURNS Dest PId;
START;
        /*type checking*

                NULL: TASK Failure:=true; RETURN NULL;
                ELSE: TASK BoundTo:=add(BoundTo,Source),
                        Failure:= false; RETURN SELF;
        ENDDECISION;
RETURN NULL
ENDPROCEDURE;
STATE unbound;
        INPUT Bind;
                DECISION failure;
                        false: NEXTSTATE Bound;
                ENDDECISION;
                NEXTSTATE -;
```

The binding action is modelled by an *EXPORTED PROCEDURE* according to Table A.9. A complementary *PROCEDURE* has to be provided to delete the binding.

A.2.2.3.3 Compound Binding

In order to use compound binding, a binding object has to be specified (*BLOCK TYPE*). This *BLOCK TYPE* must contain at least two interface *PROCESSES* and a local behaviour *PROCESS*. This binding object has to be connected to the objects involved in the binding by *CHANNELS*. *ATLEAST* clauses may be used to ensure the binding pre-conditions.

The compound binding action comprises:

- instantiation of the binding object;
- instantiation of the interface templates within the binding object, which are associated with a formal role in the binding object template (this may be part of the object instatiation);
- primitive binding of the interfaces involved in the binding with the corresponding interfaces of the binding object;
- instantiation of control interfaces as required.

Control interfaces may be specified and instantiated according to A.2.1.10 and A.2.2.5.2.

A.2.2.4 Type rules

SDL does not have means to formally describe the ODP type concept. The type rules of ITU-T Rec. X.903 | ISO/IEC 10746-3 must be used as a style guide for the specification process. The required subtyping relations for the binding of interfaces have to be expressed as behaviour of the binding action (*PROCESS* Binder).

Template type rules may be expressed using *ATLEAST* clauses.

NOTE – ASN.1 or Act One data types can be used to model the type concept, however, the relation between an object and the object type can not be verified formally.

A.2.2.5 Template rules

A.2.2.5.1 Computational object template rules

A computational object can:

- initiate or respond to signals (*INPUT*/*OUTPUT*);
- produce/consume flows;
- initiate operation invocations;
- respond to operation invocations;
- initiate operation terminations;
- respond to operation terminations;
- instantiate interface templates (*CREATE* <*process*>);
- instantiate object templates (*OUTPUT* CreateObject(<objectname>) *TO* LocalBehaviour);
- bind interfaces;
- access and modify its state (*TASK*-actions, *NEXTSTATE*);
- delete one or more of its interfaces (*OUTPUT* Delete *TO* Interface);
- delete itself (*STOP*);
- spawn, fork and join activities (*PROCESS* Creation, *SERVICE* decomposition);
- obtain a computational interface identifier for an instance of the trading function.

A.2.2.5.2 Computational interface instantiation

Computational interface template instantiation:

- creates a new interface *PROCESS*;
- produces a computational interface identifier for the interface (*PId*).

The instantiation is modelled by *CREATE* <*interface-process-type*>. The variables *SELF* of the creator and *OFFSPRING* of the createe contain the new interface identifier.

A.2.2.5.3 Computational object template instantiation

Computational object template instantiation:

- creates a new *PROCESS* LocalBehaviour for the object;
- produces a (non-empty) set of identifiers for the initial interface *PROCESSES* of the new object.

The instantiation is modelled by *CALL* CreateObject *TO* <*object-type*>. This creates a new *PROCESS* LocalBehaviour. The instantiation of the interface *PROCESSES* of the new object may be included in the CreateObject *PROCEDURE* or may be part of the start transistion of LocalBehaviour. The *PROCEDURE* CreateObject may be refined by inheritance.

Table A.10 – CreateObject PROCEDURE

```
EXPORTED PROCEDURE CreateObject ATLEAST CreateObject
RETURNS ObjectID PId;
        START VIRTUAL;  CREATE THIS;
        RETURN OFFSPRING;
ENDPROCEDURE CreateObject;
```

NOTE – The approach specified here is based on the assumption that there always will exist at least one object of that type. If this can not be guaranteed, a special (manager) process has to be added to the ComputationalObjectTemplate-BLOCK. Its purpose is the creation of new instances.

A.2.2.6 Failure rules

All possible computational failures have to be specified explicitly. SDL does not provide a means to handle failures in the execution of a specification. After the occurence of an (SDL-) error the further behaviour of a system is undefined.

A.2.2.7 Portability rules

SDL meets all requirements of the portability rules with the following exceptions:

- ordering and delivery guarantees for announcements (delivery of *SIGNALS* always succeeds or fails);
- interface signature subtype testing.

SDL provides an event based processing model, the permitted actions are represented directly as a part of the language (*INPUT,OUTPUT,CREATE,CALL*) and through syntactic structures.

A.2.2.8 Conformance and reference point

The *GATES* of interfaces *PROCESSES* act as the communication to the outside of the computational object represent the reference points.

Message Sequence Charts (MSC) can additionally be used to specify the required behaviour at a reference point. Test methodologies are available to check the conformance between an SDL specification and an ITU-MSC specification.

A language binding between the standardized interface specification language CORBA-IDL and SDL has been developed and can be used to test the conformance of an object at programmatic conformance points.

A.3 Formalization of the Computational Viewpoint Language in Z

Elementary Structures Associated with Operational and Signal Interfaces

To formalize the concepts associated with the computational viewpoint in Z, it is necessary to introduce labels [Name] for things, e.g. names of operations and their ODP types. The ODP types existing in the system are denoted by [Type].

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 context, e.g. as given in 7.2.1 of ITU-T Rec. X.903 | ISO/IEC 10746-3. Thus Param is introduced as a partial function from names to ODP types in such a context.

$$Param: Name \pfun Type$$

This function includes in its domain, all of the parameter names that exist in a given context. It is also useful to introduce sequences of these parameters to enable consideration of the sets of parameters associated with signals, invocations or terminations.

$$PList == \mathbf{seq}\ Param$$

Elementary Structures Associated with Stream Interfaces

As with the LOTOS formalization of the computational viewpoint, we consider a generic notion of flow consisting of flow elements. Each flow element in an information flow can be considered as a unit consisting of data (this may be compressed) which we represent by [Data]. This model might include how the information was compressed, what information was compressed, etc. As such it is not considered further here. Flow elements also contain a time stamp (ts) used for modelling the time at which the particular flow element was sent or received, hence we introduce the type [Time]. It is also often the case in multimedia flows that particular flow elements are required for synchronisation, e.g. synchronisation of audio with video for example. Therefore we associate a particular Name with each flow element. This can then be used for selecting a particular flow element from the flow as required. From this, we may model a flow element as:

```
┌─FlowElement──────────────────
│label : Name
│data : Data
│ts : Time
└──────────────────────────────
```

A.3.1 Concepts

A.3.1.1 Signal

A signal is an atomic interaction from an initiator to a responder. Since Z does not fully possess the object oriented feature of encapsulation, the modelling of interactions between objects requires restrictions on specification styles to be followed. For example, through ensuring that signals sent from initiators to responders have appropriate variable names (and compatible types) for the associated output and input labels respectively. Ensuring naming considerations are satisfied can be achieved through appropriate renaming of the schema text representing signal signatures as provided in A.3.1.11. An example of how this can be achieved is shown in the following Z fragment.

```
InitiatingSignalSignature ≙ SignalSignature[pl!/inArgs]
RespondingSignalSignature ≙ SignalSignature[pl?/inArgs]
```

Now a template for an initiating and responding signal may be given as:

```
┌─InitiatingSignalTemplate────        ┌─RespondingSignalTemplate────
│ InitiatingSignalSignature           │ RespondingSignalSignature
│ ...                                 │ ...
├─────                                ├─────
│ ...                                 │ ...
└─────────────────────────────────    └─────────────────────────────────
```

Here the dots are used to imply that there will likely be more information contained in the declaration part of the schemas, e.g. related to state information of the objects associated with the initiating and responding signals, and predicates to indicate the effects of the operation schemas occurring, i.e. the behaviour.

It should be noted that schema hiding and schema projection can be used to hide declarations that should not be visible during the interaction, i.e. they can be removed from the declarations and existentially quantified in the predicate part of the schema. The signal template for the interaction itself may be modelled through piping of the respective initiating and responding signal schema templates.

```
SignalTemplate ≙ InitiatingSignalTemplate >> RespondingSignalTemplate
```

Whether the signal itself can actually take place is dependent upon the satisfaction of the predicates associated with the composed schemas.

NOTE – The information that is being conveyed as required in 8.8 of ITU-T Rec. X.902/ISO/IEC 10746-2 is given by the output parameters from the initiating signal, i.e. pl!.

A.3.1.2 Operation

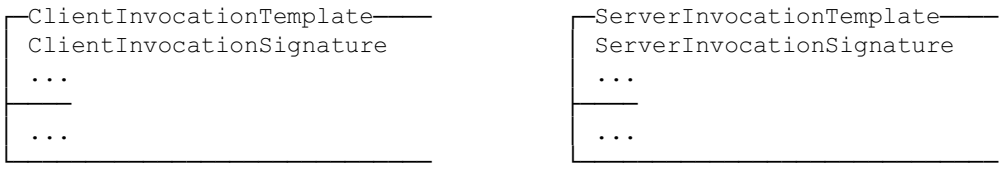
The occurrence of an operation schema modelling an interrogation or announcement. See also the Note in A.3.1.4.

A.3.1.3 Announcement

The occurrence of an operation schema modelling an invocation from a client to a server. As discussed in A.3.1.1, Z does not fully support object-orientation so modelling conventions have to be adopted to model systems of interacting objects. The conventions on naming can be enforced through appropriate renaming of the invocation template given in A.3.1.12. This can be represented as:

```
ClientInvocationSignature ≙ InvocationSignature[pl!/inArgs]
ServerInvocationSignature ≙ InvocationSignature[pl?/inArgs]
```

Now a template for a client and server invocation may be given as:



The announcement itself may then be modelled through piping of the respective client and server invocation schemas.

$$InvocationTemplate \triangleq ClientInvocationTemplate >> ServerInvocationTemplate$$

Whether the announcement itself can actually take place is dependent upon the satisfaction of the predicates associated with the composed schemas.

NOTE – The text in A.3.1.1 to explain the dots in the schemas to represent the unspecified behaviour and the usage of schema hiding and projection to provide a form of encapsulation is also applicable to announcements.

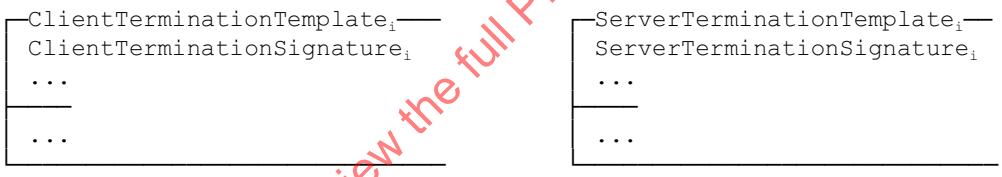
A.3.1.4 Interrogation

The occurrence of an operation schema modelling an invocation between a client and server followed by the occurrence of an associated operation schema modelling a termination between that server and client. As discussed in A.3.1.1, Z does not fully support object-orientation so modelling conventions have to be adopted to model systems of interacting objects. The conventions on naming can be enforced through appropriate renaming of the invocation template given in A.3.1.12. Invocations associated with interrogations are modelled similarly to invocations associated with announcements as given in A.3.1.3.

As discussed in A.3.1.1, schema hiding and schema projection can be used to model a form of encapsulation. Terminations and the naming conventions they are to adhere to, can be modelled through renaming of termination templates (see A.3.1.12). The client and server side of a termination may be represented as:

$$ServerTerminationSignature_i \triangleq TerminationSignature[pl_i!/outArgs]$$
$$ClientTerminationSignature_i \triangleq TerminationSignature[pl_i?/outArgs]$$

Here the underscore *i* indicates that there may be several of these (termination signatures) associated with a single invocation. Now a template for a client and server termination may be given as:



The subscript *i* is used to imply that there will likely be several termination templates, each of which has an associated signature. These signatures may be different.

Terminations themselves may then be modelled through piping of the respective server and client termination schemas.

$$TerminationTemplate_1 \triangleq ServerTerminationTemplate_1 >> ClientTerminationTemplate_1$$
$$TerminationTemplate_2 \triangleq ServerTerminationTemplate_2 >> ClientTerminationTemplate_2$$
$$...$$

A template for an interrogation as an invocation followed by one of the possible terminations may be represented as:

$$InterrogationTemplate \triangleq InvocationTemplate >> (TerminationTemplate_1 \vee TerminationTemplate_2 \vee ...)$$

Whether the interrogation itself can actually take place is dependent upon the satisfaction of the predicates associated with the composed schemas.

NOTE 1 – The text in A.3.1.1 to explain the dots in the schemas to represent the unspecified behaviour and the usage of schema hiding and projection to provide a form of encapsulation is also applicable to interrogations.

NOTE 2 – This model of an interrogation, represents a single operation schema, i.e. it is not the case that the invocation occurs first and is followed by one of the terminations. The whole interrogation represents a single atomic action which either occurs in its entirety or does not occur at all, depending upon the associated predicates. The informal commentary associated with the Z specification should be used to explain the intended effect.

A.3.1.5 Flow

The modelling of flows in Z is very much dependent upon the level of abstraction used when considering the sequence of interactions representing the flow. Typically, flows of information have stringent timing considerations associated with them. We consider here a model based upon a flow producer having a sequence of data items (flow elements) to send to

a flow consumer. These are time stamped when they are sent by the producer and this is used to determine their validity on arrival by the consumer. Examples of the state of a producer (*PState*) and consumer (*CState*) can be represented as:

```
┌─PState──────────────────────────────┐
│ps : seq FlowElement                 │
│ptnow, prate : Time                  │
│...                                  │
├─────────────────────                │
│∀f₁,f₂ : FlowElement |               │
│   ⟨f₁,f₂⟩ in ps • f₂.ts > f₁.ts     │
│...                                  │
└─────────────────────────────────────┘
```

```
┌─CState──────────────┐
│cs : seq FlowElement │
│ctnow, crate : Time  │
│...                  │
├─────────────────────│
│...                  │
└─────────────────────┘
```

Here we use the model of a flow element as given in A.3.1. We state that the producer (and consumer) state consists of at least a sequence of flow elements (*ps*/*cs*), the current local time (*ptnow/ctnow*) and the rate at which flow elements are to be sent (*prate*) or accepted (*crate*). We also state that all flow elements in the sequence of flow elements associated with a producer have increasing time stamps. With this model of the producer state we can model the sending of a flow element as:

```
┌─PSendFlowElement──────────────────────────────────────────────┐
│ΔPState                                                       │
│f!: FlowElement                                               │
│...                                                           │
├─────────────────────                                         │
│ps ≠ < > ∧ prate' = prate ∧                                   │
│ps' = tail ps ∧ ptnow' = ptnow + 1 / prate ∧                  │
│f! == (μ FlowElement | data = (head ps).data ∧                │
│             ts = ptnow' ∧ label = (head ps).label) ...       │
└──────────────────────────────────────────────────────────────┘
```

Several things should be pointed out here. Sending a flow element removes that flow element from the sequence of flow elements to be sent. The current rate associated with the flow is unchanged. The actual time at which the flow element was sent is dependent upon the current rate and time.

The actual flow element sent is the head flow element in the sequence of flow elements to be sent. This is time stamped with the value for the time calculated previously. We note here that the use of the definite description requires that a proof obligation is fulfilled to ensure that the flow element sent is unique. This obligation is satisfied through modelling all flow elements in the sequence with increasing (i.e. non-equal) time stamps. Since no flow element in the sequence has the same time stamp, the flow element sent with the current time is unique. Also we require as a precondition that the sequence of flow elements is non-empty.

A consumer may receive a flow element successfully provided the constraints for its acceptance are satisfied.

```
┌─CGetFlowElementOk──────────────────────────────────────────────┐
│ΔCState                                                         │
│f?: FlowElement                                                 │
│...                                                             │
├─────────────────────                                           │
│cs' = cs^<f?> ∧ crate' = crate ∧ ctnow' = ctnow + 1 / crate ∧ ...│
└────────────────────────────────────────────────────────────────┘
```

For brevity we do not consider the acceptance constraints in detail. These might entail allowing variations in the times at which the flow element is acceptable, e.g. jitter. The actual model of a flow may now be represented as:

```
Flow ≙ PSendFlowElement >> CGetFlowElementOk
```

We note here that this model of a flow requires that the flow element sent and received has the same base name and that the other local variables of the producer and consumer states have different labels. For brevity, we do not consider the erroneous cases associated with sending and receiving flow elements in a flow of information.

A.3.1.6 Signal Interface

An interface in which all *operation schemas* are modelled as signals. An example of the format of an *operation schema* representing a signal signature is given in A.3.1.1.

NOTE – The behaviour specification and environment contract associated with a given interface should be represented by additional Z data structures, e.g. schemas representing the state of the objects involved in the interactions at the interface. The instantiation of a given interface template should satisfy all predicates associated with the interface template.

A.3.1.7 Operation Interface

An interface in which all *operation schemas* are modelled as operations. An example of the format of the *operation schemas* representing parts of an operation signature are given in A.3.1.3 and A.3.1.4. See also the NOTE in A.3.1.6.

A.3.1.8 Stream Interface

An interface in which all *operation schemas* are modelled as flows. An example of the format of the *operation schemas* representing a flow signature is given in A.3.1.5. See also the Note in A.3.1.6.

A.3.1.9 Computational Object Template

An object template (see 4.4.2.11) which comprises a set of interface templates the object can instantiate, a behaviour specification and an environment contract. It should be noted that Z is essentially a flat notation and hence does not support the modelling of objects as a language feature directly. Instead, the natural language commentary that should be associated with every Z specification should be used to denote the Z text, e.g. the *operation schemas*, comprising the interface(s) of the objects and the relation between them.

A.3.1.10 Computational Interface Template

An interface template for either a signal interface, a stream interface or an operation interface. See also the Note in A.3.1.6.

A.3.1.11 Signal Interface Signature

An interface signature for a signal interface. A signal interface signature comprises a finite set of action templates, one for each signal type in the interface. Each action template comprises the name for the signal, the number, name and types of its parameters and an indication of causality (initiating or responding) with respect to the object that instantiates the template. A signal signature may be represented by:

```
┌─SignalSignature──────────────
│inArgs: PList
└──────────────────────────────
```

Here the schema name (`SignalSignature`) is used to represent the signal name and *inArgs* to represent the number, name and type of the parameters associated with this signal. The usage of this schema to create instances of initiating or responding signal signatures, i.e. with causality is given in A.3.1.1. See also the Note in A.3.1.6.

A.3.1.12 Operation Interface Signature

An interface signature for an operation interface. An operation interface signature comprises a finite set of announcements and interrogations as appropriate, one for each operation in the interface, together with an indication of causality (client or server) for the interface as a whole with respect to the object that instantiates the template. Announcements consist of invocations only. Interrogations consist of an invocation followed by one of the possible terminations. An invocation signature may be represented as:

```
┌─InvocationSignature──────────
│inArgs: PList
└──────────────────────────────
```

Here the schema name is used to represent the invocation name and *inArgs* to represent the number, name and type of the parameters associated with this invocation. The usage of this schema to create instances of client or server invocations signatures, i.e. with the associated causality, is given in A.3.1.3 and A.3.1.4. The predicate associated with this schema is used to satisfy the naming rules for parameters, i.e. that parameter names are unique in the context of an invocation template. See A.3.2.1.

A termination signature may be represented as:

```
┌─TerminationSignature─────────
│outArgs: PList
└──────────────────────────────
```

Here the schema name is used to represent the termination name and *outArgs* to represent the number, name and type of the parameters associated with this termination. The usage of this schema to create client or server termination signatures, i.e. with the associated causality, is given in A.3.1.3 and A.3.1.4. It is likely that there will be predicates associated with this schema, e.g. naming rules etc. as discussed in A.3.2.1. These predicates are similar to those given previously for invocation templates (with appropriate quantification changes, e.g. replace *inArgs* with *outArgs*). See also the Note in A.3.1.6.

A.3.1.13 Stream Interface Signature

An interface signature for a stream interface. A stream interface signature comprises a finite set of action templates, one for each flow type in the interface. Each action template for a flow contains the name of the flow, the information type of the flow and an indication of causality for the flow (producer or consumer) with respect to the object which instantiates the template. An example of a particular flow signature is given in A.3.1.5. The identification of flow signatures as being part of a given stream interface can be done through the informal text associated with every Z specification. See also the Note in A.3.1.6.

A.3.1.14 Binding Object

An object that supports a binding between a set of other computational objects. Since Z does not support the modelling of objects and their associated interfaces as a language feature, the modelling of binding objects in general is limited. Using the schema calculus and providing informal textual descriptions however, it is quite possible to model complex interaction scenarios where a form of binding can be considered as existing. For binding objects between client and server objects for example, this might be achieved through modelling additional operation schemas (representing parts of the interface to the binding object) that are composed with client invocations and their subsequent delivery at servers. This might be represented as:

```
InvocationViaBind ≙ (ClientInvocation >> BindInvocation) >> ServerInvocation
```

Schema *BindInvocation* should have compatible data types and labels for the variables that represent the information being passed between the client and server. An example of the schema *BindInvocation* for the client and server invocation given in A.3.1.3. is:

```
┌─BindInvocation──────────
│inArgs!,inArgs?: PList
│...
├─────────────────────────
│...
└─────────────────────────
```

Here the schema should have the same base name for the data structures being passed from the client (*inArgs?*) and being passed to the server (*inArgs!*). It should be pointed out that there is no notion of the schema *InvocationViaBind* partially failing. That is, it is not the case that the client invocation (*ClientInvocation*) and its acceptance by the binder object (*BindInvocation*) can pass and the delivery of the invocation from the binding object to the server (*ServerInvocation*) can fail. This can then be represented as:

```
InvocationViaBindFail ≙ ClientInvocation >> BindInvocationFail
```

Here the schema *BindInvocationFail* might be modelled as:

```
┌─BindInvocationFail──────
│inArgs?: PList
│...
├─────────────────────────
│...
└─────────────────────────
```

That is, the binding object accepts the data from the client (*inArgs?*) but does not deliver it to the server for some unspecified reason. The different possibilities of successful or unsuccessful operations that might take place through a binding object can be represented through the schema calculus. Typically, logical disjunction is used to represent the choices that are possible, i.e. failure cases.

NOTE – The behaviour associated with the schema *BindInvocation* might impose constraints on the data it receives and subsequently sends, i.e. it is possible to write predicates on the values of the variables it accepts as inputs and gives as outputs.

A.3.2 Structuring Rules of Computational Viewpoint

A.3.2.1 Naming Rules

The naming rules of the computational viewpoint language can be supported either through the writing of predicates, as shown in A.3.1.12 for the naming rules associated with parameters, or through the global scoping of schema names. Thus it is not possible to declare two operation schemas with the same names, i.e. all actions are uniquely identified in a semantically correct Z specification.

A.3.2.2 Interaction Rules

It is typically not the case in Z that *operation schemas* are grouped together to form a new Z construct, e.g. a schema, that represents the interface to an object. To do so would in the general case would result in a schema that does not have

the same modular structuring and with potentially conflicting predicates representing the behaviour of the individual schemas. From this it follows that the interaction rules of ITU-T Rec. X.903 | ISO/IEC 10746-3 are not generally supported in Z.

A.3.2.2.1 Signal Interaction Rules

There is no notion of causality in Z, hence it is not meaningful to state that interfaces initiate signals if they have initiating causality or respond to signals if they have responding causality. The causality label that can be applied to a given interface is done so informally. It might be the case, however, that notions of causality can be dealt with in the informal commentary associated with every Z specification in accompaniment with appropriate schema combinations, e.g. through $\gg$.

A.3.2.2.2 Stream Interaction Rules

See A.3.2.2 and A.3.2.2.1.

A.3.2.2.3 Operation Interaction Rules

See A.3.2.2 and A.3.2.2.1. It should also be noted that it is typically not the case that Z models sequencing or ordering of actions. This is typically done when refinement of a specification is made. Thus because a client sends an invocation which a server receives, there is no inherent Z language construct that requires that server to send an appropriate termination at some later stage. Instead, the sending and receiving of the invocation from the client to the server and the subsequent sending and receiving of the termination from the server to the client is usually modelled as a single schema as shown in the example of A.3.1.4. Alternatively, the actions of sending and receiving an invocation and sending and receiving a termination can be modelled as separate schemas where the accompanying informal text is used to explain their relationship.

A.3.2.2.4 Parameter Rules

It is typically not the case in Z that *operation schemas* are grouped together to form an interface of an object that can be labelled and subsequently used for interacting with the interface it references. As such Z does not directly support the modelling of computational interface references as parameters.

A.3.2.2.5 Flows, Operations and Signals

There is no inherent distinction between a flow, operation or signal in Z. They are all represented by operation schemas that can be composed with one another in numerous ways, e.g. through the schema calculus, depending upon the behaviour of the system being specified. As such, modelling flows or operations through signals can be achieved through ensuring that the schemas representing the signals have the appropriate labels and data types associated with the corresponding schema representing the flow or operation respectively.

A.3.2.3 Binding Rules

It is typically not the case in Z that *operation schemas* are grouped together to form an interface of an object that can be labelled and subsequently used for interacting with the interface it references. As such, the binding rules of ITU-T Rec. X.903 | ISO/IEC 10746-3 are not generally supported by a Z specification. Instead, it is more often the case that Z supports a form of binding based upon individual *operation schemas* (representing signals, flows, invocations or terminations) being composed with one another. An example of this is given in A.3.1.14. Through this, it is possible to ensure that certain binding rules are satisfied, e.g. through writing predicates to check on the types of parameters being passed. There is no feature of Z that restricts how *operation schemas* may be composed generally, however. For example, ensuring that only *operation schemas* with a certain name and having similar declarations are composed with one another. Composing schemas that are incompatible, e.g. combining schemas through $\fatsemi$ that have declarations of variables with similar basenames but different types, results in a semantically incorrect specification.

A.3.2.3.1 Implicit Binding Rules for Server Operation Interfaces

See A.3.2.2, A.3.2.3 and A.3.2.2.4.

A.3.2.3.2 Primitive Binding Rules

See A.3.2.2, A.3.2.3 and A.3.2.2.4.

A.3.2.3.3 Compound Binding Rules

See A.3.2.2, A.3.2.3 and A.3.2.2.4.