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 comments 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