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/IEC 2001

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2001

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

© ISO/IEC 2001 – All rights reserved

Not for Resale

CONTENTS

Page

1) Introduction 1
2) Clause 1 – Scope 1
3) Clause 2 – Normative references 2
4) Subclause 3.2 – Definitions from ITU-T Recommendation Z.100 2
5) Subclause 3.3 – Definitions from the Z-Base Standard 2
6) Annex A 3

Annex A – Computational Formalization 3
A.1 Formalization of the Computational Viewpoint Language in LOTOS 3
A.2 Formalization of the Computational Viewpoint Language in SDL 12
A.3 Formalization of the Computational Viewpoint Language in Z 20
A.4 Formalization of the Computational Viewpoint Language in ESTELLE 28

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

© ISO/IEC 2001 – All rights reserved
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 *)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 *)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 *)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 I 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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table A.2 – Interrogation using SDL signals

```
PROCESS TYPE Client
INHERITS OperationInterface;
...
OUTPUT service1(p1,p2,p3) TO Server1;
NEXTSTATE waitService1;
STATE waitService1
  INPUT service1Term1;
    ...
  INPUT service1Term2;
    ...
  SAVE*;
...
ENPROCESS TYPE Client
```

```
PROCESS TYPE Server
INHERITS OperationInterface;
        ADDING GATE InterfaceSignature
        ADDING IN WITH service1;
...
STATE bound;
        INPUT service1(a1,a2,a3);
        TASK 'perform required function';
        OUTPUT service1term1 TO SENDER;
          NEXTSTATE-;
...
ENDPROCESS TYPE Server
```

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
INHERITS OperationInterface;
IMPORTED PROCEDURE service1;
...
CALL service1(par1,par2,par3) TO Server1;
...
ENPROCESS TYPE Client
```

```
PROCESS TYPE Server
INHERITS OperationInterface;
ADDING
EXPORTED PROCEDURE service1 REFERENCED;
...
STATE bound;
  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*/

...
PROVIDED Available(Data);
  OUTPUT Frame(GetFrame(Data)) TO BoundTo;
  NEXTSTATE-;
...
```

```
/* Consumer*/

...
STATE Receive;
        PRIORITY INPUT Frame(Data);
                ...;
        NEXTSTATE-;
        INPUT NONE;

                ...;
        NEXTSTATE-;
```

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 *BLOCKS* 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*/                                  /*Destination*/
                                               VIRTUAL EXPORTED PROCEDURE Bind
...                                            NEWTYPE PIDSet ATLEAST
STATE unbound;                                       OPERATORS noequality;
...                                                          add: PIDSet,PId->PIDSet;
TASK                                                         remove: PIDSet,PId->PIDSet;
Destination:=CALL Bind(SELF,TDesc) TO Dest;    ENDNEWTYPE;
DECISON Destination=Dest;                      DCL BoundTo PIDSet,ThisType TypeDescrType, Failure Boolean>>
FALSE: /*Binding Failure*/                     FPAR Source PId, Typ TypeDescrType;
        NEXTSTATE -;                           RETURNS Dest PId;
TRUE: TASK BoundTo:= add(BoundTo,Dest)         START;
        NEXTSTATE Bound;                             /*type checking*
ENDDECISION;
STATE Bound;                                                 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 -;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 on 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.

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

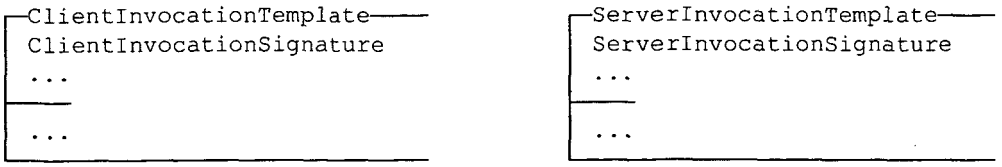
The occurrence of an operation schema modelling an invocation from a client to a server. As discussed in A.3.1.1, Z does not fully support object-orientation so modelling conventions have to be adopted to model systems of interacting objects. The conventions on naming can be enforced through appropriate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

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

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.

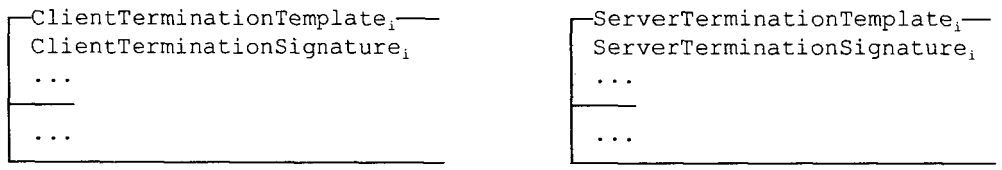
A.3.1.4 Interrogation

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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 >>.

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.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A.3.2.4 Type Rules

The type rules given for computational interfaces in ITU-T Rec. X.903 | ISO/IEC 10746-3 are not generally supported in Z due to the reasons given in A.3.2.2, A.3.2.3 and A.3.2.2.4.

A.3.2.4.1 Signature Subtyping Rules for Signal Interfaces

See A.3.2.4.

A.3.2.4.2 Signature Subtyping Rules for Stream Interfaces

See A.3.2.4.

A.3.2.4.3 Signature Subtyping Rules for Operation Interfaces

See A.3.2.4.

A.3.2.5 Template Rules

A.3.2.5.1 Computational Object Template Rules

Z supports all actions associated with the computational object template rules, provided appropriate Z text is given, e.g. operation schemas are given for modelling initiating or responding signals, with the following exceptions:

- the binding of interfaces is not supported fully due to the reasons given in A.3.2.3 and A.3.2.2.4;
- the spawning, forking and joining of activities are not a feature of the Z language, however, it is possible to model an abstract representation of these;
- obtaining a reference to a trading function is not supported fully due to the reasons given in A.3.2.3 and A.3.2.2.4;
- Z does not support computational interface signature subtype checking for the reasons given in A.3.2.3 and A.3.2.2.4.

A.3.2.5.2 Computational Interface Instantiation

Instantiation of a computational interface template can be achieved through providing a valid binding for the variables in the Z text identified as representing the interface template. It is not generally the case in Z that an identifier is produced that can be used for interacting with the created interface due to the reasons given in A.3.2.3.

A.3.2.5.3 Computational Object Template Instantiation

Instantiation of a computational object template can be achieved through providing a valid binding for the variables in the Z text identified as representing the object template.

A.3.2.6 Failure Rules

The failure modes visible to an object are determined by its behaviour specification and its environment contract. Any of the actions given in A.3.2.5 can fail.

In Z failure can be modelled in several ways. Perhaps the simplest of these is through logical disjunction of schemas. That is, a successful schema is specified along with possible error schemas. These are then combined with the successful schema through logical disjunction. The visiblity of these error schemas is dependent upon the possession of inputs (?) and outputs (!). Thus, since signal failures are visible and identical to all parties involved in the interaction, a signal failure can be specified as a schema with outputs that are sent to all parties involved in the interaction.

Depending upon the requirements, flow, operation schema failures and failures associated with interface and object instantiation, can be specified so that they are: visible to all parties at the same time; visible to only one party at a given time; and return different parameters to the different parties. All of these can be achieved through schema calculus in Z.

A.3.2.7 Portability Rules

The actions required to support a basic or extended portability standard are supported by Z, provided Z text is given to represent the appropriate action, with the exceptions identified in A.3.2.5.1.

A.3.2.8 Conformance and Reference Points

A reference point exists at any interface of any computational object. Implementors claiming conformance must list the engineering reference points corresponding to the computational reference point and state which transparencies and engineering structures apply to them. Through this act, the computational reference points become conformance points.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Thus, given a Z specification from the computational viewpoint, conformance can be checked through ensuring that the operation schemas associated with those interfaces in the specification that have been identified as conformance points by the implementor, are satisfied by the implementation. Here, satisfaction must ensure that the invariants of the specification are not violated and that all preconditions and postconditions satisfied by the specification are also satisfied by the implementation.

A.4 Formalization of the Computational Viewpoint Language in ESTELLE

This subclause shows how the formal description technique Estelle may be used to interpret the concepts and rules of the computational language of ITU-T Rec. X.903 | ISO/IEC 10746-3. In the following text, *italics* are used to denote Estelle concepts as defined in the standard document. The interpretation is explained in examples. Due to the reason that not all ODP concepts can be formally interpreted using Estelle, a style is sometimes used which implies the appropriate concept.

A.4.1 Concepts

A.4.1.1 Signal

A signal is interpreted as the submission of an Estelle *interaction* through an *interaction point* of a *module instance* and its subsequent consumption by the recipient.

A.4.1.2 Operation

The occurrence of an announcement or interrogation.

A.4.1.3 Announcement

An announcement is interpreted as the submission of an Estelle *interaction* through an *interaction point* of a *module instance* which represents the client and its subsequent use by a *module instance* which represents the server.

A.4.1.4 Interrogation

An interrogation is interpreted as the submission of Estelle *interactions*. One *interaction*, the **invocation**, is submitted from a *module instance*, representing the client. The other *interaction*, the **termination**, is submitted from a *module instance* representing the server. The management of the relations between invocations and terminations should be done in the *module instances*.

A.4.1.5 Flow

There are two ways to interpret this concept in Estelle. A flow is modelled through a sequence of *interactions* sent from a producer object to a consumer object. This can be realised by attaching a *child module* to the *interaction point*, which generates the sequence of *interactions*. Alternatively, a continuous *interaction* between both modules and the data conveyed can be used to represent an information flow.

A.4.1.6 Signal interface

A signal interface is interpreted as an *interaction point* in Estelle. The *interaction point* has a defined set of *interactions* which can only be signals. The definition of the *interaction point* is based on an appropriate *channel definition*.

A.4.1.7 Operation interface

An operation interface is interpreted as an *interaction point* in Estelle. The *interaction point* has a defined set of *interactions* which can be only invocations and their respective terminations and announcements. The definition of the *interaction point* is based on the respective *channel definition*.

A.4.1.8 Stream interface

A stream interface is interpreted as an *interaction point* in Estelle. The *interaction point* has a defined set of *interactions* which can be only flows. The definition of the *interaction points* is based on the appropriate *channel definition*. If the flow is modelled through a sequence of *interactions*, a *child module instance* is attached to the *interaction point* which realises the sequence of *interactions*.

A.4.1.9 Computational object template

A computational object template can be modelled using a *module header definition*, which contains the *interaction points*, a *module body definition* which contains the internal behaviour and the behaviour of the interface templates. There is no way to explicitly specify environment constraints in Estelle.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A.4.1.10 Computational interface template

A computational interface template is modelled using a *channel definition* which contains the interface signatures, using a *module header definition* which contains the appropriate *interaction points* and using a *module body definition* which contains the behaviour specification of the interface. Environment constraints can not be explicitly modelled using Estelle. A computational interface template in Estelle consists of an *interaction point* of a *parent module instance* which is attached to a *child module instance* with the same *interaction point* of the same *channel*. This approach provides for transparent reception of various *interactions*, because the *child module instance* is always ready for reception.

A.4.1.11 Signal interface signature

A signal interface signature is interpreted as part of the *channel definition*. This definition contains the roles associated with the *channel*, the name of the *interactions* and the information carried by them, specified using one or more parameter names and associated types.

A.4.1.12 Operation interface signature

An operation interface signature is interpreted as part of the *channel definition*. This definition contains the roles associated with the *channel*, the name of the *interaction* and the information carried by them, specified using parameter names and associated types. An announcement signature contains only an *interaction* representing the invocation. An interrogation signature contains both an *interaction* representing the invocation and a set of *interactions* representing the possible terminations.

A.4.1.13 Stream interface signature

A stream interface signature is interpreted as part of the *channel definition*. This definition contains the roles associated with the *channel*, the name of the *interactions* and the information carried by them, specified using one parameter name and associated type.

A.4.1.14 Binding object

A binding object is modelled by a *module instance*, together with the *CHANNELS* through which it is connected to computational objects.

A.4.2 Structuring rules

In Estelle, a computational specification describes the functional decomposition of an ODP system as a configuration of computational and binding objects, both represented as *module instances*. The internal actions of a *module instance* are modelled as an extended finite state machine. This machine is realised in Estelle using transitions. *Interactions* between objects are exchanged via *CHANNELS*. *Interactions* between *modules* are asynchronous. The endpoints of a *channel* are computational interfaces represented in Estelle as *interaction points*. The *module instances* are all direct *child module instances* of a *module instance* attributed *systemactivity*. This means, that the execution of all *child modules* is non-deterministic. The *parent module* can be substructered only into *modules* attributed with *activity*. The *parent module instance* provides for: instantiation; destruction; connection and disconnection of *module instances* representing computational objects. The environment contracts of the objects are not modelled explicitly in Estelle.

A.4.2.1 Naming rules

The semantics of Estelle ensure all naming rules. Names in Estelle have the following associated context: *interaction, role* and *channel* are part of a *channel definition*. There is no syntactic difference between the various *interactions*. Semantic differences are caused by the different use of the *interactions*. *Channel* and *role identifier* are used for the definition of *interaction points* in the *module header definition*. An *interaction* name is defined in the context of the *module instance*, combining it with the respective *interaction point* name. The parameter name of an *interaction* is an identifier in the context of the *interaction definition*. The name of an *interaction point* is defined in the context of all *module instances* whose *module header definition* have defined this interaction point.

The name of a *module instance* is valid in the *module body definition* where it has been defined.

A.4.2.2 Interaction rules

In Estelle *interactions* can be sent and received only at *interaction points*. The sending of an *interaction* through an *interaction point* that is not connected causes the loss of the *interaction*. The sending *module instance* is not informed about its error. If we have to report an infrastructure failure in the case where *interaction* occurs at an interface that has not been bound, all interfaces of an object that are not bound, may be connected to internal *interaction points* of the environment. Thus, *interactions* at any of these interfaces can be detected and reported as an infrastructure failure.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.4.2.2.1 Signal Interaction Rules

When a *module instance* represents an object that offers a signal interface of a given signal interface type, this interface is represented through an *interaction point* of appropriate *channel* and *role definition*, with a *child module instance* attached to it that models the behaviour at the interface. This *child module instance* may only output *interactions* with the causality parameter set to initiating. It must contain a transition with the appropriate *WHEN clauses* for the reception of any of the *interactions* defined in the signal interface template.

A.4.2.2.2 Stream Interaction Rules

A stream interface is modelled through an *interaction point* with a *child module instance* attached to it which models the behaviour at the interface. A flow can be represented by one *interaction* or by a sequence of *interactions*, according to the various kinds of a flow. If the object has a consumer role for this interface, the *child module instance* must contain a transition with the appropriate *WHEN clauses* for the reception of the flow *interaction*. If the object has a producer role, the *child module instance* must contain output statements for the flow *interaction*.

A.4.2.2.3 Operation Interaction Rules

When a *module instance* represents an object that offers an operation interface of a given operation interface type, this interface is represented through an *interaction point* of appropriate *channel* and *role* definition, with a child module instance attached to it that models the behaviour at the interface. If the object has the client role for this interface the *child module instance* must contain a transition with the appropriated *WHEN clause* for the reception of terminations and the appropriate output statements to deliver invocations. If the object has the server role for this interface the *child module instance* must contain a transition with the appropriated *WHEN clause* for the reception of invocations and the appropriate output statements to deliver terminations. In Estelle, there is no way to assign a duration to a transition or a sequence of transitions.

A.4.2.2.4 Parameter Rules

For a computational interface identifier, a specific (Estelle) data type has to be defined. Each computational interface can be identified by a variable of this type. These variables can be passed as arguments in interactions representing operations. The recipient of such a computational interface identifier can subsequently use it to bind to the referenced interface.

A.4.2.2.5 Flows, Operations and Signals

Information between Estelle modules is always exchanged using *interactions*. Using this approach, the kind of information is not visible. A signal is interpreted using only the interaction model. As the interpretation of flow and operations are based on the notion of an interaction, signals can be used to explain their special features. In Estelle, only complementary *interaction points* of the same channel can be connected. Based on the assumption that all *interactions* at an *interaction point* have the same type, a compound binding between different kinds of interfaces is not possible in Estelle.

A.4.2.3 Binding rules

In order to allow *interaction* between interfaces of computational objects, the *interaction points* modelling these interfaces are both connected to *interaction points* of the same binding object. The binding object is instantiated by a binding action. A binding action is invoked by the environment of the *module instances*. Each computational object possesses a designated external *interaction point* through which it is connected with (an internal *interaction point* of) its environment, i.e. the surrounding *module instance*. At this *interaction point*, the binding action is invoked with the appropriate parameters.

It should be noted that this *interaction point* may be considered as a special kind of interface in the ODP sense, but the binding action itself is not an ODP interaction since it does not occur between computational objects. A control interface of the binding object is connected to a corresponding control interface of the client object. The environment, i.e. the surrounding *module instance*, can check whether the interfaces to be bound are in an appropriate subtype relationship. Since environment contracts are not modelled explicitly (so far), the satisfaction of environment contracts cannot be checked. Binding without explicit participation of the environment can not be interpreted using Estelle concepts.

A.4.2.3.1 Implicit Binding Rules for Server Operation Interfaces

Implicit binding is required if the client object and the server object are still not bounded. There is no way in Estelle to detect the non-existence of this binding. The client has to handle an array with *interaction points* and bounded servers.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A.4.2.3.2 Primitive Binding Rules

In the case of primitive binding there is no binding object involved. *Interaction points* of *module instances* modelling computational objects are connected immediately. This connection can be established during instantiation or can be done by the environment after the invocation of the respective function. In the second case there will be a notification if a failure occurs.

A.4.2.3.3 Compound Binding Rules

Compound binding actions enable a set of interfaces to be bound, using a binding object to support the binding. A binding object template can be interpreted using Estelle in the same way as a computational object template. The pre-conditions for compound binding can be expressed by the following Estelle expressions:

- corresponding interface parameters must be of the same kind; this can be realised in Estelle using a valid *channel definition*;
- corresponding interface parameters must be of complementary causality. This can be realised in Estelle using a valid *channel definition* and the respective *role*;
- corresponding interface parameter must be a subtype of the signature type. Although subtyping is not supported in Estelle this can be realised using a valid *channel definition.*

A compound binding action can be interpreted using the following Estelle constructs:

- A module instance for a binding object is initialised by the *parent module*. The *module header*, the *module body* and a *module variable* have been defined before.
- During instantiation the *interaction points* of the *module instance* are instantiated also.
- Using the *connect statement* a primitive binding between the binding object and the other computational objects will be done.
- Control interfaces have been instantiated. Their identifier can be returned.
- The necessary functions at the control interface can be provided by the *module instance.*

A.4.2.4 Type rules

In Estelle, *interaction points* can only be connected if they are defined with the same *channel definition*. There is no subtyping relationship between *interaction points*. In order to bind interfaces standing in a subtyping relationship using a binding object, this binding object needs to have different interface types for its client (consumer) and server (producer) role interfaces. Since recursive definition of interfaces is not supported in Estelle, only the simplified subtyping rules can be applied. Estelle data types are based on the data types of ISO PASCAL. Thus, the only subtyping supported are subrange types. Subrange types can be defined for ordinal data types, namely the data type integer and user-defined enumerated types.

A.4.2.5 Template rules

A.4.2.5.1 Computational object template rules

A computational object can:

- initiate a signal by executing the corresponding *output statements* for the *interaction* representing the signal or respond to signals by executing the corresponding *WHEN clauses* at an *interaction point* representing a signal interface. *Output statements* and *WHEN clauses* are realised in *child modules* attached to the *interaction point* of the *parent module*;
- produce flows by executing the corresponding *output statements* for the *interaction* representing the flow or consuming flows by executing the corresponding *WHEN clauses* at an *interaction point* representing a stream interface. *Output statements* and *WHEN clauses* are realised in *son modules* attached to the *interaction point* of the *father module*. A loop can exist inside the *son module* to initiate the sequence of *interactions* modelling a flow;
- invoke operations by executing the corresponding *output statement* for the *interaction* representing the operation invocation or responding to operation invocations by executing the corresponding *WHEN clause* at a operation interface. *Output statements* and *WHEN clauses* are realised in *son modules* attached to the *interaction point* of the *father module*;
- terminate operations by executing the corresponding *output statement* for an *interaction* representing an operation termination;
- instantiate interface templates by assigning an unused *interaction point* to the new interface, instantiating the *child module instance* defining the behaviour at the interface and attaching it to the *interaction point*;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- instantiate object templates directly by instantiating a new instance of the corresponding *module definition* (this way only objects internal to the instantiating object can be created), or indirectly via an exchange of *interactions* with the environment;
- bind interfaces by invoking a binding action with the environment;
- access and modify its state by either executing a state transition or modifying variables;
- delete itself indirectly by requesting itself to be stopped by the environment;
- spawn an activity by instantiating a *child module instance*;
- bind to a trading function object by invoking a binding action with the environment.

It should be noted that forking and joining of activities are not supported in Estelle. It should also be noted that interfaces cannot be deleted, only connections between interaction points can be disconnected.

A.4.2.5.2 Computational Interface Template Instantiation

An unused *interaction point* is assigned to the new interface, a *child module instance* is created by the *parent module* and attached to the *interaction point* and an internal interface identifier is produced for the interface. Dynamic creation of *interaction points* is not possible in Estelle. This implies, that each *module instance* is created with (for each interface type) an array of *interaction points*, representing the maximum number of simultaneously existing interfaces of this type. At a given point in time, an *interaction point* is either unused or representing an interface (assigned to an interface). The *parent module* has to maintain information about the state of each *interaction point* in a suitable data structure. The initial interfaces of the object are instantiated in the *initialise-transition* of the *module*. Each *module* produces an internal interface identifier for each of its interfaces. The full external computational identifier is created by the environment of the object using the internal interface identifier and a *module identifier*.

A.4.2.5.3 Computational Object Template Instantiation

The instantiation of a computational object can only be undertaken by the environment which is modelled as a module with the attribute *systemactivity*. Two phases have to be differentiated. First, the *module instance* is created. This can be done using two generic forms of the *init-statement*. In one form, parameters can be passed to the *module instance* being created. Since *module-variables* used in the *init-statement* have to be declared before being used, dynamic creation of *module-instances* is not possible in Estelle. In a second step, appropriate *interaction points* have to be connected to their respective partners.

A.4.2.6 Failure rules

Since signals, like operation invocation submit, are modelled in Estelle by messages sent through an *interaction point*, an infrastructure failure does not occur automatically if the interface is not bound. A message sent through an *interaction point* which is not connected is simply lost. In an implementation of the Estelle interpretation, however, the loss of a message due to unbound interfaces can be detected and reported to the sending *module instance* as an infrastructure failure. Dynamic creation of *interaction points* or *module instances* is not possible in Estelle. It is always necessary to define an array with variables of the respective type. The instantiation of templates will fail if all possible instances are already existing.

A.4.2.7 Portability rules

Action templates for all actions which can be interpreted, using Estelle, can be given. Estelle meets all requirements of the portability rules with the following exceptions: forking and joining actions are not supported in Estelle nor are ordering and delivery guarantees for announcements

A.4.2.8 Conformance and Reference Points

Reference points are external *interaction points* of Estelle *modules*. Methods to derive test cases from Estelle specifications are available. Nevertheless, derivation of test cases for multi-module specifications is still an open issue. There is no language binding between a standardized interface specification language and Estelle. A comprehensive methodology for testing, based on interactions visible in communications protocols, does, however, exist.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

ICS 35.080

Price based on 32 pages

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

STD.ISO 10746-4-ENGL 1998 ■ 4851903 0773616 629 ■

INTERNATIONAL STANDARD

ISO/IEC 10746-4

First edition
1998-12-15

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

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

This material is reproduced from ISO documents under International Organization for Standardization (ISO) Copyright License Number IHS/ICC/1996. Not for resale. No part of these ISO documents may be reproduced in any form, electronic retrieval system or otherwise, except as allowed in the copyright law of the country of use, or with the prior written consent of ISO (Case postale 56, 1211 Geneva 20, Switzerland, Fax +41 22 734 10 79), IHS or the ISO Licensor's members.

Reference number
ISO/IEC 10746-4:1998(E)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Contents

		Page
1	Scope	1
2	Normative references	2
3	Definitions	2
3.1	Definitions from ISO/IEC 8807	2
3.2	Definitions from ITU-T Recommendation Z.100	2
3.3	Definitions from the Z-Base Standard	3
3.4	Definitions from ISO/IEC 9074	3
4	Interpretation of modelling concepts	3
4.1	Architectural semantics in LOTOS	3
4.2	Architectural semantics in ACT ONE	9
4.3	Architectural semantics in SDL-92	15
4.4	Architectural semantics in Z	20
4.5	Architectural semantics in ESTELLE	25

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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. 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.

International Standard ISO/IEC 10746-4 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology,* Subcommittee SC 33, *Distributed application services,* in collaboration with ITU-T. The identical text is published as ITU-T Recommendation X.904.

ISO/IEC 10746 consists of the following parts, under the general title *Information technology — Open Distributed Processing — Reference Model:*

— *Part 1: Overview*

— *Part 2: Foundations*

— *Part 3: Architecture*

— *Part 4: Architectural semantics*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

This Recommendation | International Standard is an integral part of the ODP Reference Model. It contains a formalisation of the ODP modeling 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.

This Recommendation | International Standard is accompanied by an amendment and a technical report. The associated amendment focuses on the formalisation of the computational viewpoint language contained in ITU-T Rec. X.903 | ISO/IEC 10746-3. The associated technical report contains examples on how the formalisation of the ODP Reference Model can be applied to develop specifications.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

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

1 Scope

The rapid growth of distributed processing has lead to a need for a coordinating framework for the standardization of Open Distributed Processing (ODP). This Reference Model of ODP provides such a framework. It creates an architecture within which support of distribution, interworking, interoperability and portability can be integrated.

The Basic Reference Model of Open Distributed Processing (RM-ODP), (see ITU-T Recs. X.901 to X.904 | ISO/IEC 10746), is based on precise concepts derived from current distributed processing developments and, as far as possible, on the use of formal description techniques for specification of the architecture.

The RM-ODP consists of:

- ITU-T Rec. X.901 | ISO/IEC 10746-1: **Overview**: Contains a motivational overview of ODP giving scooping, justification and explanation of key concepts, and an outline of ODP architecture. This part is not normative.
- ITU-T Rec. X.902 | ISO/IEC 10746-2: **Foundations**: Contains the definition of the concepts and analytical framework and notation for normalized description of (arbitrary) distributed processing systems. This is only to a level of detail sufficient to support ITU-T Rec. X.903 | ISO/IEC 10746-3 and to establish requirements for new specification techniques. This part is normative.
- ITU-T Rec. X.903 | ISO/IEC 10746-3: **Architecture**: Contains the specification of the required characteristics that qualify distributed processing as open. These are the constraints to which ODP standards must conform. It uses the descriptive techniques from ITU-T Rec. X.902 | ISO/IEC 10746-2. This part is normative.
- ITU-T Rec. X.904 | ISO/IEC 10746-4: **Architectural Semantics**: Contains a formalisation of the ODP modeling 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 standardized formal description techniques. This part is normative.

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 modeling and specification concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2 and 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 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.

An architectural semantics provides the additional benefits of:

- assisting the sound and uniform development of formal descriptions of ODP systems; and
- of permitting uniform and consistent comparison of formal descriptions of the same standard in different formal specification languages.

Rather than provide a mapping from all the concepts of ITU-T Rec. X.902 | ISO/IEC 10746-2, this Recommendation | International Standard focuses on the most basic. A semantics for the higher level architectural concepts is provided indirectly through their definition in terms of the basic ODP concepts.

Examples of the use of some of the formal specification languages in this report can be found in TR 10167 (Guidelines for the Application of ESTELLE, LOTOS and SDL).

In the following clauses, the concepts are numbered in accordance with the scheme used in ITU-T Rec. X.902 | ISO/IEC 10746-2.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This Recommendation | International Standard specifies an architectural semantics for ODP. This is required to:

- provide formalisation of the ODP modelling concepts;
- assist sound and uniform development of formal descriptions of standards for distributed systems;
- act as a bridge between the ODP modelling concepts and the semantic models of the specification languages: LOTOS, SDL, ESTELLE and Z;
- provide a basis for uniform and consistent comparison between formal descriptions of the same standard in specification languages that are used to develop an architectural semantics.

This part is normative.

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

- ISO/IEC 8807:1989, *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour.*
- ITU-T Recommendation Z.100 (1993), *CCITT Specification and Description Language (SDL).*
- ISO/IEC TR 10167:1991, *Information technology – Open Systems Interconnection – Guidelines for the application of Estelle, LOTOS and SDL.*
- ISO/IEC 13568[1)], *Information technology – Programming Languages their Environments and System Software Interfaces, Z Specification language.*
- The Z Notation, *A Reference Manual, J.M. Spivey, International Series in Computer Science, Second Edition, Prentice-Hall International,* 1992.
- ISO/IEC 9074:1997, *Information technology – Open Systems Interconnection – Estelle: A formal description technique based on an extended state transition model.*

3 Definitions

3.1 Definitions from ISO/IEC 8807

This Recommendation | International Standard makes use of the following terms defined in ISO/IEC 8807:

action denotation, actualisation of parameters, behaviour expression, choice, conformance, disabling, enabling, enrichment, equation, event, extension, formal gate list, formal parameter list, gate, gate hiding, guard, instantiation, interleaving, internal observable event, operation, parallel composition, parameterised type definition, process definition, reduction, selection predicate, sort, synchronisation, type definition, value parameter list.

3.2 Definitions from ITU-T Recommendation Z.100

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. Z.100:

action statement, active, atleast, block (type), call, channel, content parameter, continous signal, create, enabling condition, export, exported procedure, exported variable, finalized, gate, import, imported variable, input, nextstate, nodelay, now, output, procedure, process (type), provided, redefined, remote procedure, reset, return, revealed variable, service (type), set, signal, signalroute, stop, system (type), task, time, timer, transition, view, viewed variable, virtual.

---

1) Currently at the stage of draft.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.3 Definitions from the Z-Base Standard

This Recommendation | International Standard makes use of the following terms defined in the Z-Base Standard:

axiomatic description, conjunction, data refinement, invariant, operation refinement, overriding, postcondition, precondition, schema (operation, state, framing), schema calculus, schema composition.

3.4 Definitions from ISO/IEC 9074

This Recommendation | International Standard makes use of the following terms defined in ISO/IEC 9074:

activity, assignment statement, attach, channel, channel definition, connect, control state, DELAY-Clause, detach, disconnect, exported variable; external interaction point, FROM-Clause, function, init, instantiation, interaction, interaction point, module body definition, module header definition, module instance, output, parent instance, primitive procedure, procedure, PROVIDED-Clause, release, role, systemactivity, systemprocess, TO-Clause, transition, transition block, transition clause, WHEN-clause.

4 Interpretation of modelling concepts

4.1 Architectural semantics in LOTOS

LOTOS is a standardized (ISO/IEC 8807) Formal Specification Language (FSL). Tutorial material is available in the standard.

This clause explains how the fundamental modeling concepts can be expressed in LOTOS (see ISO/IEC 8807). It should be pointed out that there exist two main ways in LOTOS to model the concepts contained in ITU-T Rec. X.902 | ISO/IEC 10746-2. These are based upon the process algebra part of the language and the ACT ONE data typing part of the language. Since the ACT ONE formalisation of the concepts is applicable to SDL-92 also, the ACT ONE formalization is given in an independent clause. See 4.2.

To avoid confusion in the ODP and LOTOS terminology, the following clause uses *italics* to denote LOTOS specific terms.

4.1.1 Basic modeling concepts

4.1.1.1 Object

An *instantiation* of a LOTOS *process definition* which can be uniquely referenced.

4.1.1.2 Environment (of an object)

The part of a model which is not part of the object. In LOTOS, the environment of an object within a specification at a given time is given by the environment of the specification and by the other *behaviour expressions* that are composed with that object in the specification at that time.

NOTE – The environment of a specification is empty if the specification is not parameterised.

4.1.1.3 Action

Actions in LOTOS are modeled as either *internal events* or *observable events*. All events in LOTOS are atomic. An internal action may be given explicitly by the *internal event* symbol, **i**, or by an event occurrence whose associated *gate is hidden* from the environment.

An interaction is represented in LOTOS by a *synchronisation* between two or more *behaviour expressions* associated with objects at a common interaction point *(gate)*. Interactions may be of the kind:

- pure *synchronisation* on a common *gate* with no offer: No passing of values between objects occurs;
- ! and ! for pure *synchronisation*: No values are exchanged between the objects;
- ! and ? for value passing provided the ? event contains the ! event: Another way of considering this is that the ! event selects a value from a choice of values for the ? event;
- ? and ? for value establishment: Here the effect is an agreement on a value from the intersection of the set of values. If the intersection of the values is the empty set then no *synchronisation* and hence no interaction occurs.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If a non-atomic granularity of actions is required event refinement may be used. This will then enable non-instantaneous and overlapping actions to be modelled. It should be noted that event refinement is a non-trivial problem, especially when behavioural compatibility is to be maintained.

There exists no construct in LOTOS to express cause and effect relationships, although this might sometimes be possible to represent informally.

4.1.1.4 Interface

An abstraction of the behaviour of an object that consists of a subset of the observable actions of that object. As all observable actions of an object in LOTOS require *gates* with which to synchronise with the environment, the subset of observable actions is usually achieved by partitioning the *gates* given in the *process definition* associated with the object. In order to obtain an interface, *hiding* the *gates* not required for the interface under consideration can be achieved. Alternatively, *synchronising* on only a subset of the *gates* associated with an object can be used. In this case, actions occurring at those *gates* in the *process definition* not in the set synchronised with, may be regarded as actions internal to the object as far as the environment synchronising on those *gates* making up the interface is concerned.

It should be noted that this definition requires that the interfaces of an object use different *gate* names, i.e. it is not possible to distinguish between interfaces that use the same *gate*.

4.1.1.5 Activity

An activity is a single-headed directed acyclic graph of actions, where each node in the graph represents a system state and each arc represents an action. For an action to occur it must satisfy the preconditions of the system state.

4.1.1.6 Behaviour (of an object)

The behaviour of an object is defined by the LOTOS *behaviour expression* associated with the *process definition* that constitutes the object template. A *behaviour expression* may consist of a sequence of both externally visible event offers and *internal events*. The actual behaviour of an object as might be recorded in a trace, is dependent upon the *behaviour expression* associated with the object and how this is configured with the environment. The actual behaviour the object exhibits depends upon the *behaviour expression* of the object and how this synchronises with its environment. An object may exhibit non-deterministic behaviour.

4.1.1.7 State (of an object)

The condition of an object that determines the set of all sequences of actions in which the object can take part. This condition is governed by the *behaviour expression* defined in the object template from which the object was created and possibly by the current bindings of any existing local variables.

4.1.1.8 Communication

The conveyance of information (via value passing) between two or more interacting objects. It is not possible to write directly, cause and effect relationships. It should also be pointed out that the *synchronisation* itself may be construed as communication.

4.1.1.9 Location in space

LOTOS abstracts away from the notion of location in space. It is only possible to equate space with the structure of the specification model. The location of an event – the structural location with respect to the specification model – is given by a *gate* for interactions in LOTOS. The notion of location in space at which an *internal event* can occur is abstracted away from in LOTOS. This abstraction is achieved implicitly using the LOTOS *hide ... in* construct which makes *gates* used internally within a process invisible to the environment of the process, or explicitly using the *internal event* symbol, **i**.

It is possible for the same location in space to be used for more than one interaction point. This is made possible in LOTOS by having a single *gate* with different *action denotations*.

The location of an object is given by the union of the locations of the *gates* associated with that object, i.e. the union of all of the locations of the actions in which that object may take part.

4.1.1.10 Location in time

LOTOS abstracts away from the concept of time, only considering temporal order so there is no *absolute location in relative metric time*. Location in time would be possible, however, if an extended form of LOTOS were used with time aspects incorporated.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.1.1.11 Interaction point

A *gate* with a possibly empty list of associated values.

NOTE – In a specification, changes in location may be reflected by changes in the associated values.

4.1.2 Specification concepts

4.1.2.1 Composition

- **of objects:** A composite object is an object described through the application of one or more LOTOS combination operators. These include:
  - *interleaving operator* (|||);
  - *parallel composition operators* (|| and |[*gate-list*]|);
  - *enabling operator* (>>);
  - *disabling operator* ([>);
  - *choice operator* ([])
- **of behaviours:** The composition of the *behaviour expressions* associated with the component objects in the creation of a composite object through composition. The operators for the composition of behaviours are the same as those for the composition of objects.

4.1.2.2 Composite object

An object described using one or more of the *interleaving, parallel composition, disabling, enabling* and *choice operators* of LOTOS.

4.1.2.3 Decomposition

- **of objects:** The expression of a given object as a *composite object.* There may be more than one way to decompose a composite object, however.
- **of behaviours:** The expression of a given behaviour as a composite behaviour. There may be more than one way to decompose a composite behaviour.

NOTE – It might also be considered that the notion of decomposition of behaviours is inherently supplied by the ACT ONE *operations* and equations associated with a *sort.* That is, these *operations* and *equations* provide all possible combinations of behaviours. Thus for example, sequential composition might be generated through *operations* applied sequentially. Each *operation* application in the sequence must satisfy the necessary equations for occurrence. Whether this is behavioural composition is debatable though, since the *operations* and equations already existed and defined all possible behaviours.

4.1.2.4 Behavioural compatibility

In LOTOS, specific theories have been developed to check for behaviour compatibility. There are no specific LOTOS language syntactical features to construct and ensure behaviour compatibility generally. The LOTOS standard, however, develops the notion of *conformance* which provides a basis for consideration of behaviour compatibility.

In order to determine whether or not two object behaviours are compatible, the notion of *conformance* needs to be introduced. *Conformance is* concerned with assessing the functionality of an implementation against its specification, where here the term implementation may be taken to be a less abstract description of a specification.

If **P** and **Q** are two LOTOS processes, then the statement **Q** conforms to **P** (written as **Q conf P**) signifies that **Q** is a valid implementation of **P**. This means that if **P** can perform some trace σ and then behave like some process **P'**, and if **Q** can also perform trace σ and then behave like **Q'** then the following conditions on **P'** and **Q'** must be met: whenever **Q'** can refuse to perform every event from a given set A of observable actions, then **P'** must also be able to refuse to perform every event of A.

Thus **Q conf P** if and only if, placed in any environment whose traces are limited to those of **P**, **Q** cannot deadlock when **P** cannot deadlock. Another way of defining this is **Q** has the deadlocks of **P** in an environment whose traces are limited to those of **P**.

An object can be made behaviourally compatible with a second object after some modification to its behaviour, which might include *extending* the object's behaviour (adding additional behaviour) or a *reduction* of the object's behaviour (restricting the object's behaviour). This process of modification of an object is known as refinement (see 4.1.2.5).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.1.2.5 Refinement

Refinement is the process by which an object may be modified, either by extending or reducing its behaviour or a combination of both, so that it conforms to another object. Letting **P** and **Q** be LOTOS processes, an *extension* of **P** by **Q** (written as **Q extends P**) means that **Q** has no less traces than **P**, but in an environment whose traces are limited to those of **P**, then **Q** has the same deadlocks. A *reduction* of **P** by **Q** (written as **Q reduces P**) means that **Q** has no more traces than **P**, but in an environment whose traces are limited to those of **Q**, then **P** has the same deadlocks.

4.1.2.6 Trace

A trace of the behaviour of an object from its initial instantiated state to some other state is a recording of the finite sequence of interactions *(observable events)* between the object and its environment.

4.1.2.7 Type of an <X>

Types that can be written down explicitly in LOTOS for objects and interfaces are template types. There is no explicit construct in LOTOS that will permit the modeling of action types as such. A LOTOS specification consists of a *behaviour expression* which is itself composed of *action denotations* (action templates). These action templates either occur as part of the behaviour of the system, in which case their occurrence may loosely be regarded as the action template instantiation, or they do not occur, in which case the action template remains uninstantiated. The action templates themselves may be given by the *internal event* symbol, **i**, or event offers at *gates* which may or may not have finite sequence of value and/or variable declarations.

LOTOS does not offer facilities to characterise actions directly, however, a limited form of action characterisation is built into the *synchronisation* feature of LOTOS. That is, it might be considered that synchronised *action denotations* (action templates) must satisfy the same action type in order for the action to occur. However, LOTOS does not classify the characterising features of these arbitrary *action denotations* and thus it is not possible to put a formal type to any given action. It might be the case that informally the event offers involved in an interaction are given a cause and effect role, but this is generally not the case. See 4.1.1.8.

The *internal event* symbol may be used to represent an action type, where the common characteristics of this collection of actions are that they have no characteristics.

It should be noted that by stating that the only predicate possible in LOTOS for objects (and interfaces) are that they satisfy their template type, the concepts of type and template type as given in ITU-T Rec. X.902 | ISO/IEC 10746-2 reduce to the same modeling technique in LOTOS. Thus there is no distinction in LOTOS between a type in its broad characterisation sense, and a template type in its more restrictive sense of template instantiation.

4.1.2.8 Class of an <X>

The notion of class is dependent upon the characterising type predicate which the members of the class satisfy. Objects, interfaces and actions can satisfy many arbitrary characterising type predicates. A type that can be written down is a template type. When this is the case, the class of objects, interfaces and actions associated with that type is the template class.

NOTE – It should be noted that by stating that the only classification possible in LOTOS for objects, interfaces and actions is that they satisfy their template type, the concepts of class and template class as given in ITU-T Rec. X.902 | ISO/IEC 10746-2 reduce to the same modeling technique in LOTOS. Thus there is no distinction in LOTOS between a class in its general classification sense, and a template class in its more restrictive sense as the set of instances of a given template type.

4.1.2.9 Subtype/Supertype

As the types that can be written down in LOTOS for objects, interfaces and, to a lesser extent, actions, are template types, a subtype relation in LOTOS is a relation that may exist between template types. In LOTOS, however, there exists no direct feature to write down subtyping relations directly. If subtyping is required then *extension* can be used to give a subtype relation based on substitutability, however, this is not a feature explicitly provided for in LOTOS.

4.1.2.10 Subclass/Superclass

As the types that can be written down in LOTOS for objects, interfaces and, to a lesser extent, actions, are template types, a subclass relation exists between two classes when a subtyping relation exists between their corresponding template types.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.1.2.11 <X> Template

- **Object Template:** A *process definition* with some means by which it can be uniquely identified once instantiated. If no value parameter list is given, then object identification will not be possible for more than one object instantiated from the object template.

  With regard to combination of object templates in LOTOS there are no existing combination operators except for a limited form of scoping using the LOTOS "*where*" term.

- **Interface Template:** Any behaviour obtained from a *process definition* by considering only the interactions at a subset of the *gates* associated with the *process definition.* This subsetting of the *gates is* achieved by *hiding* the *gates* not required for the interactions under consideration.

  With regard to combination of interface templates in LOTOS there are no existing combination operators except for a limited form of scoping using the LOTOS "*where*" term.

- **Action Template:** An *action denotation* where an *action denotation* may be either an *internal-event* symbol, a gate-identifier or a gate-identifier followed by a finite sequence of value and/or variable declarations.

NOTE – The definition here of *action denotation* is contrived as LOTOS does not really support the concept of an action template. In LOTOS, possible behaviours are specified by giving *action denotations* combined in some form. To relate a template to an *action denotation* is the closest that can be achieved in LOTOS. However, the text of ITU-T Rec. X.902 | ISO/IEC 10746-2 requires an action template to group the characteristics of actions. This is not part of LOTOS as event offers *(action denotations)* exist in isolation and it is not possible to collect them and apply a template to characterise them.

Composition of action templates may loosely be likened to *synchronisation* with value passing or value generation. In this case, two (or more) action templates agree on a common action template for the *synchronisation* to occur, i.e. an action template with the common characteristics of all of the action templates involved in the *synchronisation* (composition).

4.1.2.12 Interface signature

An interface signature as a set of action templates associated with the interactions of an interface is represented in LOTOS by a set of *action denotations.* The members of this set are those *action denotations* that require *synchronisation* with the environment in order to occur.

4.1.2.13 Instantiation (of an < *X* > Template)

- **of an Object Template:** The result of a process which uses an object template to create a new object in its initial state. This process involves the *actualisation* of the *formal gate list* and *formal parameters* of a *process definition* by a one-one relabelling from a specified gate list and list of actual parameters. The features of the object created will be governed by the object template and any parameters used to instantiate it.

- **of an Interface Template:** The result of a process by which an interface is created from an interface template. The interface created can thereafter be used by the object it is associated with to interact with the environment. The features of the interface created will be determined by the interface template and any parameters used to instantiate it.

- **of an Action Template:** This is given as action occurrence in LOTOS. This may involve the rewriting of ACT ONE expressions.

4.1.2.14 Role

A name associated with a *process definition* in the template for a composite object (i.e. LOTOS composition of *behaviour expressions).* As such, roles cannot be used as parameters. However, it is possible to assign data values to each role in a composition in order to distinguish or address them specifically.

4.1.2.15 Creation (of an < *X* >)

- **of an object:** The instantiation of an object template as part of the behaviour of an existing object.

- **of an interface:** As objects and interfaces are modelled the same way in LOTOS (via *process definitions),* creation of objects corresponds to creation of interfaces. Thus the definition for interface creation is given by the creation of objects.

4.1.2.16 Introduction (of an object)

The *instantiation* of the behaviour associated with a LOTOS specification.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.1.2.17 Deletion (of an < *X* >)

- **of an object:** The termination of a process *instantiation.* This may be achieved through the use of the LOTOS *disabling operator,* the LOTOS inaction *(stop) behaviour expression* which does not allow for the passing of control, or the successful termination *(exit) behaviour expression* where passing of control is possible via the *enabling operator.*
- **of an interface:** The process by which the future behaviour of an object is limited to that behaviour which did not require the participation of the given interface to be deleted.

4.1.2.18 Instance of a type

- **of an Object Template:** An instance of a given object template is represented in LOTOS by an instantiation of that object template or an acceptable substitution for an instantiation of that object template. Here the acceptable substitute should capture the characteristics that identify this type. Thus an acceptable substitute might be another template that is behaviourally compatible with the first. This might be achieved through *extension* as defined in section 4.1.2.4. Using this relation guarantees that all characteristics of the type under consideration are included. It might be the case, however, that a weaker form of type satisfaction relation can be found which does not require all characteristics associated with a given template to be included, but some subset of the total characteristics.
- **of an Interface Template:** As an interface template is represented the same way as an object template (via a *process definition* in LOTOS), the above text applies equally well (i.e. replace all occurrences of object with interface) for instance of an interface template.
- **of an Action Template:** An instance of an action template *(action denotation)* is represented in LOTOS by another *action denotation* offering an equivalent event offer.

4.1.2.19 Template type (of an < *X* >)

A predicate expressing that an < *X* > is an instance of a given template, where an < *X* > may be an object, an interface or an action.

4.1.2.20 Template class (of an < *X* >)

The template class of an < *X* > is the set of all < *X* >'s that are instances of that < *X* > template, where an < *X* > may be an object, an interface or an action.

NOTE – The notion of the template class of an action is limited in its application to LOTOS, as LOTOS does not provide explicitly for action templates, action template instantiations or action template types.

4.1.2.21 Derived class/Base class

If the template of a class is an incremental modification of the template of a second class, then the first class is a derived class of the second class, and the second class is a base class of the first.

LOTOS templates can be incrementally modified by *extending, enriching* and modifying the *data types* or by modifying the behaviour. Problems arise with the behaviour modifications however, specifically:

- subtyping: Non-determinism may be introduced into the system when the initials of the inherited template and its modification are the same, thus subtyping cannot be guaranteed;
- the need for a redirection of self-reference: Any reference to a derived template from a parent template should be redirected to the derived template, which is not always possible.

There is no satisfactory solution to these problems in standard LOTOS.

4.1.2.22 Invariant

In LOTOS, the only invariants which can be written down are *process definitions.* There is no way to attach an invariant to a *process definition* which is not the *process definition* itself.

4.1.2.23 Precondition

A precondition is a predicate that a specification requires to be true in order for an action to occur and may be expressed directly in LOTOS using one or more of:

- sequencing of actions;
- *guards* and *selection predicates.*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.1.2.24 Postcondition

In LOTOS, the occurrence of an action is independent of the state of the system after the occurrence of the action. As such, LOTOS does not provide the means to directly express postconditions.

4.2 Architectural semantics in ACT ONE

This subclause provides a formalisation of the basic modelling and specification concepts contained in ITU-T Rec. X.902 | ISO/IEC 10746-2. Whilst ACT ONE is not by itself a standardised FSL, it is used in the standardised FSLs LOTOS and SDL-92 and provides an alternative way in which to formalise the aforementioned concepts. Therefore the ACT ONE formalisation of the concepts contained in ITU-T Rec. X.902 | ISO/IEC 10746-2 is presented in its own separate clause.

4.2.1 Basic modelling concepts

4.2.1.1 Object

An instance of a *sort* which can be uniquely referenced. It should be pointed out that objects modelled this way must be specified so that they have some form of existence. This can be achieved through a process algebra specification style. Examples of this style include recursion in *process definitions,* where the object is an element of the *value parameter list* associated with that *process definition.* Alternatively, **let...in** clauses can be used to model objects with a form of existence. In both these styles, *guards* and/or *selection predicates* are required to ensure that instantiations of *sort* definitions are unique.

4.2.1.2 Environment of an object

The environment of an object is not provided for in ACT ONE. This notion can only be considered through the process algebra and the ACT ONE expressions that exist there. In effect, the environment of an object may be regarded as all of the process algebra other than the ACT ONE expressions representing the object in question and the *operations* on that object. That is, the environment is used to cause *operations* on an object to occur. This notion of environment does not require that the *operations* on an object are invoked by other objects. This has consequences on notions such as interaction, i.e. here interaction does not take place between objects but between an object and some outside agency – here the process algebra.

4.2.1.3 Action

An *operation* occurrence. It should be noted that there is in general, no inherent distinction between an interaction and an internal action purely from an ACT ONE perspective. That is, possible actions are modelled through *operations* in the signature of an ACT ONE *sort,* and these may or may not occur, depending upon the occurrence of the ACT ONE expressions existing in the process algebra. Thus internal actions are not explicitly catered for in ACT ONE. It might be the case, however, that a form of internal action can be modelled through *sorts* defined locally in the process algebra. Alternatively, all *operations* declared in the process algebra may be regarded as interactions. Operations used to satisfy these *operations,* i.e. in the *equations* associated with the *operations* under consideration, may be regarded as internal actions. For example, if processes call an *operation pop2* which removes two elements from a queue and this uses the *operation pop* twice in its associated *equations,* then *pop2* may be regarded as an interaction, whilst *pop* can be regarded as an internal action. The problem with this treatment of internal action, however, is that there is no notion of spontaneous transitions as such, e.g. as for instance with the *internal event* symbol **i** in the process algebra.

It should be pointed out that this form of interaction does not require that two or more objects interact in the process algebra sense, i.e. through *synchronisation* on a common *gate.* Rather, here interaction may be interpreted as something which is caused indirectly by the environment but not necessarily caused by an object, i.e. not by another instance of a *sort* modelling an object. Thus it might be the case that an event offer occurrence which does not involve ACT ONE expressions causes an interaction to take place, e.g. through an event offer occurrence which results in the *instantiation* of a *process definition* whose *value parameter list* contains an *operation* (interaction) on an object (or objects).

4.2.1.4 Interface

The *operations* and *equations* associated with an object.

4.2.1.5 Activity

A sequence of *operation* applications on a given *sort.* These *operations* must satisfy the *equations* associated with the *sort.* Each *operation* in the sequence of *operations* that occurs, i.e. each *operation* in the activity, must have preconditions that satisfy the postconditions of the previous *operation* occurrence. Preconditions and postconditions on *operations* are defined in 4.2.2.23 and 4.2.2.24 respectively.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.1.6 Behaviour (of an object)

The behaviour of an object modelled in ACT ONE is dependent upon the *operations* associated with the object template and the current value the state of the object is bound to. That is, the value the state of an object is bound to can be used to limit the possible *operations* that can take place, e.g. through excluding certain *operations* from occurring whose *equations* are not valid for that value of the state.

ACT ONE does not explicitly provide for constraints on *operation* occurrences such as sequencing, non-determinism, concurrency and real-time constraints as such. Rather, ACT ONE provides *operations* and *equations* which in themselves denote all possible constraints, i.e. all possible behaviours with their associated constraints.

There does not exist the feature in ACT ONE to model internal actions specifically. That is, there is no notion of spontaneous transitions as might occur in the process algebra with the *internal event* symbol **i** for example.

It should be pointed out that there is no real notion of behaviour actually occurring solely in ACT ONE. The notion of ACT ONE behaviour occurring is normally associated with the ACT ONE expressions that are evaluated in the process algebra.

4.2.1.7 State (of an object)

The current value that an instance of a *sort* modelling an object is bound to. It should be pointed out that a *sort* modelling an object should contain an identifier used to distinguish between different instances of the *sort.* The value of this identifier does not represent part of the state though, i.e. this value should remain immutable in the *operations* and *equations* associated with the *sort.*

4.2.1.8 Communication

This notion is not supported in ACT ONE. It might be the case that an abstract form of communication might be modelled through a process algebra specification style. This does not reflect the text of ITU-T Rec. X.902 | ISO/IEC 10746-2, however, i.e. it is not the case that one object conveys information to another object. Rather, it is the environment (the process algebra) being used to communicate with and not other objects.

4.2.1.9 Location in space

The notion of a location in space is not explicitly supported in ACT ONE. If required, this notion can be engineered into the specification model, e.g. through a *sort* modelling a location in space that is used in *operations* whose location in space is to be ascertained.

4.2.1.10 Location in time

The notion of a location in time is not explicitly supported. However, if the notion of time is related to the current state of a given object, i.e. to the state changes that have occurred and those that can occur, then the location in time at which a given action can occur may be determined, to some extent, by the current state of a given object.

The location in time at which an action can occur may also be engineered into the specification, e.g. through a *sort* modelling a location in time that is used in *operations* whose location in time is to be ascertained.

4.2.1.11 Interaction point

This notion is not directly supported in ACT ONE. It might be the case, however, that this concept can be engineered into a specification. For example, through a *sort* used in the *operations* that are in the set of interfaces at the same location. That is, all *operations* in the set of interfaces at the same location require an input parameter *(sort)* indicating the location at which this *operation* exists. If required, several interaction points can be modelled so that they exist at the same location. This could be achieved through an *operation* to create a location *sort* that required several interaction point *sorts* as inputs. The *operations* and *equations* associated with these *sorts* should enable identification of distinct interaction points and locations.

4.2.2 Specification concepts

4.2.2.1 Composition

- **of objects:** It is not generally the case that two arbitrary objects can be combined in ACT ONE and have a meaningful result, i.e. a composite object with its own behaviour, etc. The most likely form of composition in ACT ONE is through a constructor *operation* which has two or more objects as input parameters and a means whereby it can be uniquely identified. Consider the following ACT ONE constructor *operation:*

makeCO: Id, Ob1, Ob2 -> CO

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Here a composite object is being created from two other objects which have their own associated *operations* and *equations,* i.e. their own behaviours. Whilst it is the case that *co: CO = makeCO (id1,ob1,ob2) is* composed of objects *ob1* and *ob2,* object *co* has no behaviour as such. That is the behaviours associated with *ob1* and *ob2* are not applicable to instances of *CO.*

To solve this problem, the composite object can have additional *operations* and *equations* specified. The form of these *operations* and *equations* and their relationship to the component objects, then determines the form of the composition. For example, if the *operations* and *equations* of the composite object simply enable access to isolated component objects, then a form of delegation is achieved, with the component object composition representing an aggregation. If the *operations* and *equations* of the composite object are specified in such a way that they modify the behaviour of the component objects, then a form of composition is achieved that more closely resembles the text of ITU-T Rec. X.902 | ISO/IEC 10746-2. It should be pointed out, however, that the notion of composition given in ITU-T Rec. X.902 | ISO/IEC 10746-2 does not require further specification to be made, i.e. it involves composing existing objects and behaviours to generate new objects and behaviours, and not specifying additional behaviour to enable the component object composition meaningful.

- **of behaviours:** Since ACT ONE does not provide specific composition operators, the notion of composition of behaviours is not explicitly provided for. It should be pointed out that a form of composition does exist in ACT ONE: that of *enrichment.* This may not be classified as composition generally, however, as it does not provide explicit composition of behaviours (or objects) as such. Rather, *enrichment* by itself, i.e. without further specification, offers no composition features. All of the *data types* exist independently when *enrichment is* applied. It is only when *operations* and *equations* are specified which use the *sorts* made available through *enrichment,* that the notion of composition can be applied. This may not adequately reflect the text of ITU-T Rec. X.902 | ISO/IEC 10746-2 however, i.e. it is not the case that two behaviours are simply combined. Further specification is required to combine the behaviours. It should be added that *enrichment* does make available all of the existing *operations* and *equations* of the *sorts* being combined though. Thus the respecification necessary to combine behaviours does not include the *operations* and *equations* of the *sorts* introduced through *enrichment.*

A form of composition of behaviours might also be achieved through the *actualisation* of *parameterised data types.* These require the *data type actualisations* satisfy any *formal sorts, operations* and *equations* of the *data type* being *actualised.* This is the closest that can be achieved in ACT ONE to capture the notion of composition of behaviour as put forward in ITU-T Rec. X.902 | ISO/IEC 10746-2. It is debatable whether this represents composition of behaviours though, since there can be no behaviour of a *parameterised data type* until it has been *actualised,* i.e. it is not the case that an instance of this *sort* can occur in the process algebra and *operations* applied.

4.2.2.2 Composite object

The result of a composition of objects.

4.2.2.3 Decomposition

- **of objects:** Objects may be decomposed in ACT ONE provided that *operations* exist in the signature associated with the object to permit the decomposition. For example the following *data type* permits a composite object to be decomposed into its component objects.

```
type Z is X, Y, IdType
    sorts Z
    opns makeZ: Id, X, Y -> Z
         getX: Z -> X
         getY: Z -> Y
    eqns forall x: X, y: Y, id: ID
         ofsort X
              getX (makeZ(id,x,y)) = x;
         ofsort Y
              getY (makeZ(id,x,y)) = y;
endtype (* Z *)
```

Thus given *z: Z = makeZ(id1,x,y),* where *x* and *y* are instances of *sorts* modelling objects and *id1 is* a unique identifier, this can be decomposed into *x* and *y*, i.e. it's component objects, through *getX(Z)* and *getY(Z)* respectively.

NOTE – This interpretation is based upon the idea of being able to separate out a composite object into its component parts (objects). The text given in ITU-T Rec. X.902 | ISO/IEC 10746-2, however, requires only that decomposition specify a given object as a combination of two or more objects, i.e. a composition. In ACT ONE, it is always the case that composite objects are specified from combinations of component objects. Hence the distinction between composition and decomposition as given in ITU-T Rec. X.902 | ISO/IEC 10746-2 is somewhat blurred when represented in ACT ONE.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

– **of behaviours:** The notion of decomposition of behaviours is dependent upon the specification of behavioural composition. This concept is not explicitly provided for in ACT ONE (see 4.2.2.1). That is, behaviours are represented by *operations* and *equations* acting on a *sort*. It is not the case that two arbitrary *sort* behaviours can be combined and a new behaviour yielded.

NOTE – It might also be considered that the notion of decomposition of behaviours is inherently supplied by the ACT ONE *operations* and *equations* associated with a *sort*. That is, these *operations* and *equations* provide all possible combinations of behaviours. Thus for example, sequential composition might be generated through *operations* applied sequentially. Each *operation* application in the sequence must satisfy the necessary *equations* for occurrence. Whether this is behavioural composition is debatable though, since the *operations* and *equations* already existed and defined all possible behaviours.

4.2.2.4 Behavioural compatibility

In LOTOS, behavioural equivalence of *data types is* based on name equivalence of *sorts* and possibly also by the value that these *sorts* are bound to. As a result of this, it is not generally the case that an object can replace another object in some environment if the objects are obtained from different object templates, i.e. they are instances of different *sorts*. It may sometimes be possible to replace one object with another object derived from a different object template, however. This requires that the environment offer *operations* that are applicable to both *sorts* and the results of these *operations* are the same. For example, *sorts* representing a stack of integers and a queue of integers might be behaviourally compatible in some environment if the environment offers only a *top operation*, and the queue and stack have the same number of integers pushed onto them. That is, the result in both cases will be an integer. If an environment offers a *pop* or a *push operation*, then behavioural compatibility will not exist between the objects, as the *operations* return stack *sorts* and queue *sorts*. Since in general, the environment of an object may invoke any *operation* in the signature, this form of behaviour compatibility is limited.

4.2.2.5 Refinement

Whilst the notion of refinement has been explicitly provided for in the process algebra of LOTOS, e.g. through *conformance* testing and equivalence relations, there has been little work done on refinement in ACT ONE. Intuitively, however, refinement in ACT ONE might take many forms. For example, through extending the signature of a given *sort*, i.e. providing more *operations*. This form of refinement should generate natural behaviour compatibility, i.e. the existing *operations* and *equations* remain unchanged. Other forms of refinement might also be possible, e.g. modification of the *equations* associated with the *operations* on a *sort*. Ensuring behaviour compatibility is unlikely to be trivial in this case.

4.2.2.6 Trace

Since interactions are not explicitly provided for in ACT ONE, the notion of a trace is limited, i.e. it cannot be guaranteed not to contain internal actions. If interactions are considered as the *operations* that occur in the process algebra and internal actions as the *operations* used to evaluate the *equations* associated with these *operations*, then a trace may be modelled to a limited extent. In this case it corresponds to the sequence of *operation* applications associated with an instance of a *sort* modelling an object. It should be pointed out that if the *equations* associated with *operations* modelling interactions are rewritten, then the record of an object's interactions, i.e. the trace, is likely to be incorrect. For example, the *operation* applications of a *push* followed by a *pop* on a *queue* are likely to be rewritten as *queue* as opposed to the expression *pop(push(x,q))*. Hence the notion of a trace is limited in ACT ONE.

4.2.2.7 Type of an <X>

Objects, interfaces and actions specified in ACT ONE can satisfy many different arbitrary characterising predicates. Types that can be written down explicitly are template types.

4.2.2.8 Class of an <X>

The notion of class is dependent upon the characterising type predicate which the members of the class satisfy. Objects, interfaces and actions can satisfy many arbitrary characterising type predicates. A type that can be written down is a template type. When this is the case, the class of objects, interfaces and actions associated with that type is the template class.

NOTE – It should be noted that by stating that the only classification possible in LOTOS for objects, interfaces and actions is that they satisfy their template type, the concepts of class and template class as given in ITU-T Rec. X.902 | ISO/IEC 10746-2 reduce to the same modelling technique in LOTOS. Thus there is no distinction in LOTOS between a class in its general classification sense, and a template class in its more restrictive sense as the set of instances of a given template type.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.2.9 Subtype/Supertype

The notion of subtype and supertype is not generally supported in ACT ONE since LOTOS uses name equivalence when type checking. For example, two object types are only the same when they are represented by the same *sort*. Thus it is not generally the case that one *sort* can be substituted for another *sort*. It might be the case, however, that a limited form of subtyping exists between two different *sorts* if the characterising type predicate is based on some aspect of the *sorts* other than their name, e.g. this *operation is* valid on this *sort* and returns this result. This is a limited form of typing though and is unlikely to exist in most cases.

4.2.2.10 Subclass/Superclass

The notion of subtypes and supertypes are only supported to a very limited extent in ACT ONE, i.e. where the characterising type predicate is based upon some aspect of the *sort*. As a result the notions of subclass and superclass are not fully supported in ACT ONE. If a subtype/supertype relationship does exist between two *sorts*, then a subclass/superclass relationship also exists between the instances of the *sorts* in the process algebra.

4.2.2.11 <X> Template

- **Object Template:** A *sort* definition with associated *operations* and *equations* modelling an object.
- **Interface Template:** The set of *operations* and *equations* associated with a *sort* definition modelling an object. It should be noted, that the notions of interface and object templates may be regarded as identical since *operations* and *equations* must always act on a *sort* definition. It is also the case that the declaration of an instance of a *sort* in the process algebra has implicitly associated with it, the *operations* and *equations* as specified in the ACT ONE part of the specification.
- **Action Template:** An *operation* with associated *equations*.

4.2.2.12 Interface signature

The *operations* that apply to a variable declared as an instance of a *sort* modelling an object.

4.2.2.13 Instantiation (of an < *X* > Template)

- **of an object template:** Instantiation of an object template requires the initialisation of a *sort* modelling an object to a valid initial state. This *sort* initialisation should ensure that the *sort* instance can be uniquely referenced.
- **of an interface template:** Instantiation of an interface template occurs when an object template is instantiated. As such, an object has a single interface given by the *operations* and *equations* acting on the *sort* from which the object is instantiated.
- **of an action template:** The occurrence of an ACT ONE *operation* in the process algebra. This *operation* must satisfy the *equations* associated with that *operation*.

4.2.2.14 Role

The notion of a role may best be modelled through a *sort* in ACT ONE. This is because a role represents an identifier for a behaviour. That is, through the declaration of a *sort*, the *operations* and *equations* that apply to that *sort* are made accessible.

4.2.2.15 Creation (of an < *X* >)

- **of an object/interface:** Since objects and interfaces only have a form of existence when ACT ONE is used in conjunction with the process algebra, ACT ONE by itself may not be used to model creation. ACT ONE used in conjunction with the process algebra may be used to model creation of objects and interfaces to a limited extent provided a certain specification style is followed. For example, an *operation* associated with a *sort* modelling an object, i.e. an *operation* on an already existing object, which results in the generation of a new object. It should be possible to uniquely reference the newly generated object. This new object should also be used in the process algebra so that it has some form of existence (see 4.2.1.1 for further details on how this can be achieved.)

4.2.2.16 Introduction (of an object)

Introduction of an object may be achieved in several ways in ACT ONE when used in conjunction with the process algebra. For example, through event offers occurring whose *action denotations* result in a new instance of a *sort* modelling an object being generated. These new instances should be in a valid initial state, be uniquely referenceable, and have some form of existence in the process algebra. Alternatively, objects may be introduced through **let ... in** clauses. Here too they should have a valid initial state, be used in the process algebra so that they have a form of existence, and it should be possible to uniquely identify them.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.2.2.17 Deletion (of an <*X*>)

- **of an object/interface:** Deletion of an object or interface may be achieved in ACT ONE when used in conjunction with the process algebra through the rewriting that occurs with the *equations* associated with the *operations* on *sorts* modelling objects. For example, an *operation* which removes an element from a set can be used to model deletion, e.g. an object with a certain identifier is removed (deleted) from the set of instantiated objects existing as part of the *value parameter list* of a recursive *process definition.*

4.2.2.18 Instance of a type

- **of an Object/Interface/Action Template:** An instance of a type depends upon the characterising predicate defining the type. If the type predicate is the template type for objects and interfaces, an instance of the object or interface type corresponds to an occurrence of the *sort* modelling the objects and interfaces under consideration in the process algebra. Similarly, if the type predicate is the template type for actions, an instance of an action type is given by the occurrence of an *operation* modelling the action type under consideration in the process algebra.

4.2.2.19 Template type (of an <*X*>)

- **of an Object/Interface:** A predicate on instantiations of the *sort* used to model an object. All instantiations (occurrences) of the template (*sort*) in the process algebra have the *operations* and *equations* that are associated with that *sort.* Since a template for an object and an interface are modelled the same way, i.e. by a *sort* definition with associated *operations* and *equations,* the template type of an object and the template type of the interface to that object are synonymous in ACT ONE. That is, they both correspond to an occurrence of a *sort* in the process algebra.
- **of an Action:** A predicate on *operation* occurrences in the process algebra. That is, all instantiations (occurrences) of the template (*operation*) in the process algebra must fulfill the requirements of the template, i.e. they must have the same inputs and produce the same results as given by the *operation* definition, and the *operation* evaluation is governed by the *equations* associated with that *operation.*

4.2.2.20 Template class (of an <*X*>)

- **of an Object:** The set of instances of a given *sort* modelling an object in the process algebra.
- **of an Interface:** The set of instances of a given *sort* modelling an interface to an object in the process algebra.
- **of an Action:** The set of instances of a given *operation* in the process algebra.

4.2.2.21 Derived class/Base class

Derived classes and base classes are not supported in ACT ONE. This is because classes in ACT ONE are normally only given through template classes, i.e. for objects, the set of instances of a given *sort* modelling an object in the process algebra. It is not the case that *sorts* can be incrementally modified. That is, *sorts* and the labels that are attached to them, i.e. the name of the *sort,* do not allow reference to another *sort,* i.e. self-reference always exists. Thus the *operations* and *equations* associated with a given *sort* are only applicable to that *sort* and not another *sort.*

It should also be pointed out that the notion of *actualisation* of parameterised classes, whilst intuitively possessing the features of derived/base class relationships, do not in effect represent such a relationship. This is because it is not the case that instances of a parameterised class can occur in the process algebra, i.e. they must be *actualised so* that a class can exist.

4.2.2.22 Invariant

This notion is implicit within ACT ONE, i.e. objects must always satisfy the *operations* and *equations* that apply to them.

4.2.2.23 Precondition

In ACT ONE all *operations* must satisfy all *equations* (and any associated *guards*) that apply to them before they can occur.

4.2.2.24 Postcondition

This notion is implicit within ACT ONE, i.e. the occurrence of a given *operation* (action) requires the associated *equations* to be defined (true).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3 Architectural semantics in SDL-92

SDL-92 is a standardised FSL. Tutorial material is annexed to the ITU-T Rec. Z.100. A number of SDL textbooks exist as well as commercial tools, which support different aspects of SDL from graphics handling to analysis and generation of programming code based on SDL.

SDL models a *system* as a set of extended finite state machines communicating by messages, called *signals*. The data concept of SDL is based on ACT ONE. The state machines are extended in that they may define local variables to hold part of their history. The signals are communicated asynchronously, thereby offering loose coupling between components in a distributed system.

This subclause expresses one way in which modelling concepts can be expressed in SDL. The representation is not regarded as unique. However it establishes that almost all of the fundamental concepts can be expressed in SDL. It should be pointed out here that an alternative approach to modelling many of the concepts given here also exists. This focuses on the use of ACT ONE. Details of this approach may be found in 4.2.

The version used is SDL-92, as defined in the ITU-T Rec. Z.100. SDL-92 contains a number of extensions compared with SDL-88. The most important ones are:

- object-oriented constructs;
- possible nondelaying *channels*;
- non-determinism;
- possible inclusion of alternate data concepts *and remote procedure* calls.

The most significant feature of the alternate data typing is that they enable combined usage of ACT ONE and ASN.1 within SDL. The semantics of the combination of SDL-92 with ASN.1 is defined in the ITU-T Rec. Z.105.

To avoid confusion, *italics* have been used in the text to indicate SDL concepts, whenever it has been felt necessary.

4.3.1 Basic modelling concepts

4.3.1.1 Object

Objects in SDL are instances of *system type, block type, process type, service type, timer, channel,* and *signalroute.* These instances are characterised by a state and a behaviour.

Each instance is encapsulated, i.e. any change in its state can only occur as a result of an internal action or as a result of an interaction with its environment.

References for some kind of objects have to be provided explicitly.

4.3.1.2 Environment (of an object)

The environment of an object depends on the object kind. See Table 1.

Table 1 – Object environment

Object kind	Environment constraints
system	– incoming *signals* of channels
block	– global datatypes – incoming *signals* of channels – calls of *exported procedures* – imported variables
process	– global datatypes – incoming *signals* of implicit or explicit *signalroutes* – calls of *exported procedures* – viewed/imported variables – time constraints for *input* actions
service	– global variables/*timers*/datatypes, shared signal buffer (owned by enclosing *process* instance) – incoming *signals* of implicit or explicit *signalroutes* – calls of *exported procedures* – viewed/imported variables – time constraints for *input* actions
timer	– calls of *set* and *reset, stop* of the owner *process*
channel	– incoming *signals* from the connected *blocks* (resp. *system* environment env)
signalroute	– incoming *signals* from the connected *process/service* instances

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.3.1.3 Action

An action in SDL is a single *input* or *save,* an *action statement,* a whole *transition* or the complete execution of a *procedure.* Possible single *action statements* are:

- *task, import, export, view;*
- *output;*
- *create;*
- *set, reset, active;*
- *procedure call;*
- *stop/return;*
- *nextstate.*

The transmission of a *signal* by a *channel* or a *signalroute* is an action also, as is the generation of a *timer signal.* Interactions are the *input/output* of a *signal,* the *call* and the *return* action of a *remote procedure,* and the use of shared variables (global *process* variables of *services, revealed/viewed* and *import/export* of *process* variables). The action sequence of sending, conveying and eventually receiving of a *signal (output-input)* can be considered as one interaction.

4.3.1.4 Interface

Depending on the kind of an object there are different means in SDL to describe interfaces, as shown in Table 2.

In case of a *block* object the set of *remote procedures exported/imported* to/from the outside of the *block* as well as the set of *signals* sent/received by *processes* of that *block* should be encapsulated in one or more processes. These *processes* then act as the interfaces of the *block* object. With those interface descriptions only the potential interactions of an object are defined, where each interface describes a subset of potential interactions of an object.

Table 2 – Object Interfaces

Object kind	Interface is characterised by
system	– *system gates* with their *signal* lists – ingoing/outgoing *channels* with their *signal* lists
block	– *block gates* with their *signal* lists – ingoing/outgoing *channels* with their *signal* lists – ingoing/outgoing *signalroutes with* their *signal* lists – set of *remote procedures (exported/imported* to/from outside of *block)* – set of remote variables *(exported/imported* to/from outside of *block)*
process	– *process gates* with their *signal* lists – set of all valid *input/output signals* – set of all *exported/imported procedures* – set of shared variables *(revealed/viewed* to/from outside of *process)*
service	– *service gates* with their *signal* lists – set of all valid *input/output signals* – set of all *exported/imported procedures*
timer	– *timer* identification
channel	– sets of all *signals* carried in each direction
signalroute	– sets of all *signals* carried in each direction

4.3.1.5 Activity

In general an activity cannot be denoted explicitly, since it may span several objects.

One special case of an activity is the execution of a local or *remote procedure* with the *call* action being the head of the activity and the potential return actions being the tail actions.

4.3.1.6 Behaviour (of an object)

The behaviour of a *process/service* is the set of all transitions of that *process/service.* The *input* actions provide constraints on the circumstances in which the transitions may occur. Additional constraints can be introduced using the *provide* construct or the *continuous signal* construct. An object may exhibit non-deterministic behaviour.

The behaviour of a *block* is aggregated from the behaviour of the *processes* contained in that *block.* The behaviour of a system is aggregated from the behaviour of the *blocks* contained in that *system.*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The behaviour of a *channel* or a *signalroute* is the conveyance of *signals* (instantaneously or delayed when specified).

The behaviour of a *timer is* predefined in SDL in the sense of an alarm clock.

4.3.1.7 State (of an object)

The set of all sequences of actions in which an object can take part at a given instant in time is determined in case of a *process/service* by the current SDL state at that time, the values of the local variables, and the content of the input port.

The state of a *block* or a *system* is the total of the states of all contained *processes, blocks* plus all contained *channels* or *signalroutes.*

The state of a *channel* is given implicitly or explicitly by a *block.* The state depends whether the *channel* has a *delay* property or not.

The state of *signalroute* is always given implicitly.

The state of a *timer* is active or not active. The state of an active *timer* is determined by the remaining time delay for sending a timeout *signal.*

4.3.1.8 Communication

The conveyance of information between two or more objects is done by explicit or implicit (in case of remote *procedures* or *imported/exported* variables) *channels* or *signalroutes.* Information is carried by *signals.*

4.3.1.9 Location in space

Actions occur within *process* instances and *service* instances. Transmission actions occur within *channels* or *signalroutes.*

4.3.1.10 Location in time

Each action is characterized by a date when the action begins and a date when the action ends. Actions can be instantaneous. The duration of an action cannot be denoted explicitly. Actions can be scheduled for a concrete date or after a specified delay.

NOTE –

a) There is a global time in SDL accessible by *now.* Nothing is said about the time units.

b) Two consecutive actions *a1* and *a2*, hold the relation *now(a1) <= now(a2).*

c) The only ways to address explicitly time are the set action for a *timer* and the application of *now* in *enabling conditions* and *continuous signals.* Scheduling of actions for a fixed point in time should be avoided.

4.3.1.11 Interaction point

Interaction points are the gates of *block/process/service* instances and the endpoints of (possibly implicit) *channels* and *signal-routes.* Shared variables are interaction points too. They have a location. An object can have several interaction points.

4.3.2 Specification concepts

4.3.2.1 Composition

– **of objects:**

  a) a *system* object may be a concurrent composition of *block* objects which may be interconnected by *channels*;

  b) a *block* object may be a concurrent composition of *block* objects (which may be interconnected by *channels*) or a concurrent composition of *process* objects, which may be connected by *signalroutes*;

  c) a *process* object may be an interleaving composition of *service* objects;

  d) a *channel* may be a composition of *blocks* interconnected by *channels*;

– **of behaviour**

  a) the behaviour of a *system* is a concurrent composition of the behaviour of its *blocks*;

  db) the behaviour of a *block* is a concurrent composition of the behaviour of its *sub-blocks* or of its *processes*;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

c) the behaviour of a *process* is an interleaving composition of the behaviour of its *services* or a sequential composition of the actions of its *process* graph;

d) the behaviour of a *service* is a sequential composition of the actions of its *service* graph;

e) the behaviour of a *channel* may be a composition of the behaviour of the *blocks* it consists of and of the *channels* interconnecting them.

4.3.2.2 Composite object

According to 4.3.2.1 the following objects can be expressed as a composition:

- *system;*
- *block;*
- *process;*
- *channel.*

4.3.2.3 Decomposition

- **of objects:** The specification of a given object as a composition.
- **of behaviours:** The specification of a given behaviour as a composition.

4.3.2.4 Behavioural compatibility

There is no general means to describe behavioural compatibility in SDL explicitly, however the semantical basis of the language in terms of transition systems allows for the definition of behavioural compatibility and its verification.

An instance of a derived class can be considered restricted behavioural compatible with an instance of the corresponding base class and an instance of a *redefined* type can be considered restricted behavioural compatible with an instance of the corresponding *virtual* type. *Atleast*-clauses can be used to require a restricted behavioural compatibility.

4.3.2.5 Refinement

There are two ways to refine an SDL specification of an object:

- substructuring (on *block* or *system* level);
- use of the object-oriented features (inheritance, virtuality and generic parameters).

4.3.2.6 Trace

There is no explicit means to specify traces in SDL. Traces can be obtained as the result of the interpretation of an SDL specification according to the dynamic semantics of SDL.

NOTE – Message Sequence Charts (MSC) provide an appropriate syntax and semantics for the representation of traces of SDL specifications. There is a tight relation between the syntax and semantics of MSC and the syntax and semantics of SDL. MSC are defined and standardised in ITU-T Rec. Z.120.

4.3.2.7 Type (of an <X>)

There is no general explicit predicate in SDL.

4.3.2.8 Class (of <X>)

This concept is not supported in general, only for template types.

4.3.2.9 Subtype/Supertype

This concept is not supported in general.

4.3.2.10 Subclass/Superclass

This concept is not supported in general, only for template types.

4.3.2.11 <X> Template

- **Object Template:** Object templates are type definitions for the appropriate object kind *(system, block, process, service).* For *timers, channels* and *signalroutes,* the object templates are the corresponding declarations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- **Interface Template:** Depending on the interface kind, an interface template can be given implicitly by a declaration *(channel, signalroutes)* or explicitly by a type definition for a *process* (see 4.3.1.4).
- **Action Template:** An action template is specified by the definition of a *process/ procedure/ service graph*. Atomic action templates are *input, output, save, set, reset, create, task, stop, return, nextstate, call, import, export, view*.

Templates may be specified using parameters (formal parameters or formal context parameters). Parameters may have constraints. Templates may be combined (i.e. a type definition may contain other type definitions).

4.3.2.12 Interface signature

The set of *signal types* and *remote procedure types* which are valid for the interface.

4.3.2.13 Instantiation (of an <X> template)

There are two ways in SDL to instantiate templates:

- implicit instantiation (*system*, *block*, *channels*, *signalroutes*, *processes*, *services*) is done by object declaration;
- explicit instantiation using *create* (only for *processes)*.

Instantiations are always the result of an action to instantiate a template. Formal context parameters have to be actualized before the instantiation can be obtained (by a *process* type specialization or a *process* declaration).

4.3.2.14 Role

There is no general means to specify roles.

Roles may be described as formal context parameters.

NOTE – *Atleast*-clauses can be used for a further qualification of a role.

4.3.2.15 Creation (of an <X>)

There are two kinds of creation (see 4.3.2.13):

- implicit instantiation;
- explicit instantiation – interpretation of a *create* action.

4.3.2.16 Deletion (of an <X>)

Only *process* objects can be deleted. A *process* can delete only itself. This is done through the interpretation of the *stop* action. If a *service* interprets a *stop* action it results in the deletion of this *service,* the deletion of all other *services* belonging to the same *process* and the deletion of the *process.*

NOTE –

a) The deletion of one *process* by another *process* can be modelled using the *output* of a special *signal* which eventually consumption by the receiver causes the receiver to interpret a *stop.*

b) The deletion of all *processes* of a *block* can be considered as a deletion of that *block.*

4.3.2.17 Introduction (of an object)

The implicit instantiation (see 4.3.2.13) can be considered as introduction.

4.3.2.18 Instance of a type

An object is an instance of a *system type, block type, process type* or *service type* X, if there is an explicit or implicit instantiation for that X or a substitute of X. A substitute is an instance of a type template which is a type specialization in SDL.

4.3.2.19 Template type (of an <X>)

The fact that an <X> is an instantiation of an <X> template can be expressed for *processes, services, blocks* and *systems* by the denotation that the object is an SDL instance of the type definition.

4.3.2.20 Template class (of an <X>)

There is no general explicit notation to characterise the template class of an <X>, however the template class is the set of all *processes, blocks, services* or *systems* instances from a *process type* definition, *block type* definition, *service type* definition or system type definition respectively.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.3.2.21 Derived class/Base class

A type definition may be derived from another type definition *by specialisation*, which may comprise:

- binding of context parameters and addition of new context parameters;
- inheriting definitions;
- redefinition of virtual components;
- addition of further definitions.

Constraints may be applied using the *atleast* and *finalised* constructs.

NOTE – Multiple inheritance is not supported.

4.3.2.22 Invariant

There is no notation for invariants in SDL.

4.3.2.23 Precondition

Enabling conditions, *continuous signals* and *signals* can be used to specify the preconditions of a transition.

4.3.2.24 Postcondition

There is no notation for postconditions in SDL.

4.4 Architectural semantics in Z

The Z notation is a specification notation based on strongly typed set theory and first order predicate calculus. Z is not yet an ISO FSL but a standard is being prepared by ISO SC22 WG19. The latest version of the Standard is contained in the Z-Base Standard. See clause 2.

A de-facto standard for Z exists (Spivey's Z Notation). See clause 4. This version of the notation is stable and very close to the Z-Base Standard, and tool support exists for syntax and static semantics checking. This version of the notation will be used for the architectural semantics of RM-ODP until an ISO version of the documentation is widely available.

To avoid confusion in ODP and Z terminology, italics are used in the following subclauses to denote Z-specific terms.

4.4.1 Basic modelling concepts

4.4.1.1 Object

An object may be described in Z by a collection of specification fragments. These fragments should contain a collection of *operation schemas* (representing the interface to the object) including appropriate *state schema(s)*. These state schemas may include predicates which are used to represent (fragments of) the invariants of the object. The specification fragments should also have some means whereby they can be uniquely referenced (representing the identity of the object). This can be achieved through having an identifier in the *state schema(s)* of the object that remains constant in all operations defined for that object. Finally, there must exist a valid initial state for that object. This can be achieved through an initialisation schema that gives legal bindings to the variables declared in the *state schema* with a predicate that ensures the object is unique within the specification.

NOTE – Care has to be taken when specifying objects in Z, since the language does not possess features of encapsulation (essential for describing objects) as discussed in the Note of 4.4.1.3.

4.4.1.2 Environment (of an object)

The environment of an object in a Z specification is described in terms of its input and output. Input to an object comes from the environment. Output of an object goes to the environment. The environment of an object can either be specified directly or left unspecified. If it is unspecified, then the occurrence of *operation schemas* producing outputs or requiring inputs may occur with the environment either providing the inputs or receiving the outputs respectively. If the environment of an object is specified, however, then this implies that for each *operation schema* associated with the object, there exists another *operation schema* (possibly associated with another object) that requires inputs or outputs of the same type as the object under consideration. These two *operation schemas* are then conjoined with one another with the inputs/outputs of the operation under consideration being renamed as the outputs/inputs of the operation representing the environment respectively.

The environment of an object may also be given by variables referenced by an object that have a global scope, e.g. those found in *axiomatic descriptions*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.4.1.3 Action

An action is modelled in Z by the performance of an operation specified in an *operation schema.* The effect is the instantaneous change in state (or the null change) of the objects with which that action is associated. An action may produce a non-deterministic result.

Since there is no explicit notion of encapsulation in Z, it is not usual to determine whether an action is observable or internal in Z, hence the distinction between interactions and internal actions is not clearly defined. This Recommendation | International Standard will use the convention that an *operation schema* representing an action which has either inputs, outputs or variables global to the specification interacts with its environment. The environment may or may not be specified (see 4.4.1.2). Actions requiring inputs from an unspecified environment that produce no outputs may be regarded as externally invoked non-observable actions. Actions producing outputs going to an unspecified environment may be regarded as internally invoked (spontaneous) observable actions. Actions that require inputs from an unspecified environment and produce outputs going to that environment may be regarded as externally invoked observable actions.

If the environment of an object is specified, however, then this implies that for each *operation schema* requiring inputs or outputs that is associated with an interface to an object, i.e. an observable action, there exists another *operation schema* (possibly associated with another object) that requires inputs or outputs of the same type as the object under consideration. These two *operation schemas* are then conjoined with one another with the inputs/outputs of the operation under consideration being renamed as the outputs/inputs of the operation representing the environment respectively.

Alternatively the occurrence of operations that reference variables that are global throughout the specification can be regarded as interactions.

All operations in Z are atomic. That is, *operation schemas* in Z either happen in their entirety or do not happen at all. Thus, it is not possible in Z to have actions that are not atomic.

An object interacting with itself can be modelled informally by composition of Z *operation schemas.* For example, operation OpA with output $a!: A$ can be composed with operation OpB, with input $b?: A$, and a predicate *conjunction* added to state that $a! = b?$.

The notion of cause and effect relationships are not strictly within the scope of Z. However, if an operation requires an input to occur, then this might be considered as the environment causing this operation to occur, i.e. the environment acts as the producer and the *operation schema* as the consumer. Similarly, if an *operation schema* produces an output, then this might be considered as the environment acting as the consumer and the operation as the producer. If a given *operation schema* requires both inputs and gives outputs, or has no inputs or outputs, then it is not possible to give a cause and effect relationship to that particular action.

NOTE – It should be noted that this syntactic convention for distinguishing internal and observable actions is limited since there is no semantic distinction between operations which are to be interpreted as spontaneous or internal, and those which require environmental participation; this can only be achieved using the natural language commentary which should accompany all Z specifications. As a consequence of this, the above definition treats a lossy queue as a subtype of a queue. Clearly though the intention of the extra lose operation in a lossy queue is that it should occur non-deterministically.

4.4.1.4 Interface

An abstraction of the behaviour of an object obtained by identifying the operations associated with that object that are to form the substance of the interface. In all remaining *operation schemas* all inputs and outputs are hidden and the occurrence of the operations defined in these *operation schemas* are regarded as internal actions, i.e. they do not require or involve the participation of the environment of the object. The resulting Z text representing that object is an interface template. Any instance of the interface template is an interface.

4.4.1.5 Activity

The notion of an activity as a single headed directed acyclic graph of actions does not exist directly in the Z language. However, the concept of an activity may be modelled to some extent by noting that if action x precedes action y in some activity, then the *postcondition* of action x must imply the *precondition* for action y.

4.4.1.6 Behaviour (of an object)

The behaviour of an object in a given state is the set of all possible activities that may occur from that state. The actual sequence of actions that may occur may be affected by the environment of the object and the constraints expressed in the *preconditions.*

4.4.1.7 State (of an object)

A binding of the state variables declared in the *state schema(s)* associated with the object template used for calculating *preconditions.*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4.4.1.8 Communication

Communication may be modelled in Z through inputs and outputs to operations. Inputs to and outputs from *operation schemas* are normally considered as communications with the environment of an object. Since communication occurs between objects, the environment of an object (see 4.4.1.2) must be specified to model communication. Communication is then achieved by firstly normalising the *operation schemas* associated with the interacting objects and then conjoining them, with the outputs of one operation being renamed as the inputs to the other *operation schema.* This modelling of communication requires that the inputs and outputs of the associated *operation schemas* are of the same type.

Alternatively, the occurrence of *operation schemas* that reference variables global to the specification represent communications, with the value of the global variable following the operation occurrence being communicated with all other *operation schemas* referencing that variable.

4.4.1.9 Location in space

The concept of space is not considered primitive in Z. The location in space at which an action occurs can only be given in Z in terms of the specification model rather than the real world system being modelled. Thus a location in space might be introduced as a Z type. Through this, relations can be specified associating *operation schemas* with given locations in space. This then makes it possible to reason about locations in space at which actions can occur.

4.4.1.10 Location in time

The concept of time is not considered primitive in Z. The location in time at which an action occurs can only be given in Z in terms of the specification model rather than the real world system being modelled. Thus a location in time might be introduced as a Z type that can be associated with given actions, e.g. through some relation. Through this, quantification over the time at which actions can occur, can be achieved.

4.4.1.11 Interaction point

The concept of interaction point depends upon the definitions of interaction and locations in space and time. See 4.4.1.3, 4.4.1.9 and 4.4.1.10.

4.4.2 Specification concepts

4.4.2.1 Composition

- **of Objects:** Composition of objects is not a feature explicitly offered by the Z language, due amongst other things to the lack of encapsulation. However, it is possible to model some characteristics of composition through schema inclusion and redefinition of operations through promotion.
- **of Behaviours:** As a behaviour in its most degenerate case may be considered an action, and an action in Z is the performance of an operation defined by an *operation schema,* composition of actions equates to the combination of *operation schemas* in Z. Operation schemes may be combined in several ways in Z. such as:
  - *schema calculus;*
  - *schema composition* (;);
  - *overriding* (⊕).

Generally, characteristics of the resulting behaviour may be derived from the composition which may not be derived from the individual behaviours being combined. In addition, irrelevant details of the behaviours being combined may be suppressed.

4.4.2.2 Composite object

See interpretation of composition above.

4.4.2.3 Decomposition

- **of objects:** See interpretation of composition of objects above.
- **of behaviours:** See interpretation of composition of behaviours above.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.4.2.4 Behavioural compatibility

Behaviour compatibility is based upon the notion of substitutability in a given environment. Extension is one possible way of achieving this. An extension of a base template may have extra components in the associated *state schema,* a stronger state invariant, stronger initial conditions and more *operation schemas.* The *operation schemas* associated with the extension of the template type may have weaker *preconditions* and stronger *postconditions* than the corresponding *operation schemas* in the base template type.

4.4.2.5 Refinement

Refinement is the process of transforming one specification into a more detailed specification. Since Z deals with abstractions of systems where data and operations on that data are used to represent the given system under consideration, two main forms of refinement have been identified:

- *operation refinement*; and
- *data refinement.*

In order to refine a specification, the refinement must ensure behaviour compatibility between the specification and the refinement. To account for this, certain conditions exist to ensure that a Z specification refinement produces a valid more detailed specification. These are the *safety* and *liveness* conditions. The safety condition on the refinement of a specification is that any circumstance acceptable to the specification must be acceptable to the refinement. The liveness condition on the refinement of a specification is that for any circumstance acceptable to the specification, the behaviour of the refinement must be allowed by the specification.

The safety and liveness conditions need to apply to both the operation and data refinements.

4.4.2.6 Trace

The modelling of a trace in Z is limited for two reasons. Firstly, there is no direct way to record an objects actions, and secondly, there is no semantic distinction between internal and observable actions as discussed in the Note in 4.4.1.3.

4.4.2.7 Type (of an < *X* >)

An object, interface or action can have many different ODP types. ODP types correspond to sets in Z, where the characterising predicate is given by set membership.

4.4.2.8 Class (of < *X* >'s)

The set of all < *X* >'s such that the set membership predicate, i.e. the ODP type, is true.

4.4.2.9 Subtype/Supertype

Subtypes and supertypes in ODP correspond to subsets and supersets respectively in Z.

4.4.2.10 Subclass/Superclass

Subclasses and superclasses in ODP correspond to the subset and superset relationships respectively in Z.

4.4.2.11 < *X* > Template

- **Object Template:** Fragments of a specification that represent the common features of the objects possible states, have a unique (immutable) identity that can be referenced, and associated set of *operation schemas* that act on that state. If the object template is a generic one, the precise form of template will only be given when the type of the parameterising parameters is given.
- **Interface Template:** A set of *operation schemas* derived from the Z text representing an object template in the way described under the interpretation of interface (see 4.4.1.4). If the object template is a generic one, the precise form of interface template will only be given when the type of the parameterising parameters are given. Interface templates may be combined using the Z operations for schema combination.
- **Action Template:** An *operation schema.* Action templates may be combined using the Z operations for schema combination. If the action template is a generic one, the precise form of action template will only be given when the type of the parameterising parameters are given.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.4.2.12 Interface signature

The set of action templates associated with the interactions of an interface.

NOTE – It should be noted that the text in ITU-T Rec. X.902 | ISO/IEC 10746-2 treats an interface signature as a set of action templates associated with the interactions of an interface. Given that an action template is the common features of a collection of actions, it is likely that this definition is incorrect. That is, an action template is likely to include semantic information as well as syntactic. Common interpretations of interface signature deal primarily at the syntactic level, however. If a syntactic notion of interface signature is considered, then this corresponds to the set of normalised action templates associated with the interactions of an interface with all non-input/output variables declared in the *operation schema* signature being existentially quantified in the predicate part of the *operation schema.*

4.4.2.13 Instantiation (of an <*X*> Template)

- **of an Object Template:** Specification of the initial values of the variables referred to in the object template. This is often provided for explicitly in Z by an initialisation schema or set of schemas. The values of these variables after the initialisation must correspond to a valid state, i.e. a state which satisfies any *invariants* that may be present.

  NOTE – These *invariants* may refer to other objects.

- **of an Interface Template:** Constraining the interface template fragments of the Z specification by specifying values of the generic parameters and providing appropriate predicates on variables referred to in the interface template.

- **of an Action Template:** The performance of an operation specified in an *operation schema.*

4.4.2.14 Role

A role may be represented by a name which identifies an individual object, e.g. through an identifier found in the *state schema* associated with the object. This name may then be used with a *framing schema* to promote the operations of the individual object to effect the system (specification) as a whole.

NOTE – This description may not adequately capture the intention of the associated text in ITU-T Rec. X.902 | ISO/IEC 10746-2.

4.4.2.15 Creation (of an <*X*>)

- **of an Object:** The creation of an object in Z is given by providing a valid initial state for the Z text associated with the object template. That is, by providing a binding of the variables given in the *state schema* of the object to the initial values that they hold. Often this is provided for explicitly in Z through an initialisation schema. In this case, the action of creation is represented by the performance of the operation given in the initialisation schema.

- **of an Interface:** The creation of an interface in Z is inherently linked with the creation of objects. That is, when the text associated with an object template is initialised, any interface templates that might be present, are initialised also.

In creation, it is necessary to ensure that the identity of the object being created is unique within the specification. This can be achieved through a *framing schema* with an appropriate predicate ensuring the identities of all of created objects are unique. This *framing schema* can then be used to promote the initialisation of an object to effect the specification as a whole.

NOTE –

a) The text given here implies that because an initialisation schema is given as part of a specification, an object is created. However, in Z there is no notion of the specification actually applying this initialisation schema since Z itself is not executable. Thus is this in fact the introduction of an object, i.e. an object is instantiated by a mechanism not covered by the model? It would appear that the notion of initialisation of a Z specification is partly creation (in that an initialisation schema is given) and partly introduction (in that the application of the initialisation schema is not covered by the model).

b) It is normally the case that a proof obligation follows the application of an initialisation schema to ensure that the object is in a valid initial state.

4.4.2.16 Introduction (of an object)

See Creation (of an object) (see 4.4.2.15).

4.4.2.17 Deletion (of an <*X*>)

It may be possible to have an abstract representation of deletion where a *framing schema is* used. Deletion can then be modelled as the promotion of an operation to remove an object state and identity from the system as a whole.

It might also be the case that a form of deletion based upon inactivity may be modelled. For example, an object whose associated future behaviours may no longer occur due to *invariants* being violated may in some sense be considered as deleted. This form of deletion may not accurately capture the definition as given in ITU-T Rec. X.902 | ISO/IEC 10746-2 though, i.e. there is no destruction as such.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.4.2.18 Instance of a type

An instance of an ODP type in Z is represented by an element of the set whose members satisfy the predicate, i.e. the ODP type. Given a more specific notion of type, such as template type, an instance of an object or interface type corresponds to the initialisation of the Z text (or an extension of it) representing the object/interface under consideration, such that there exists a valid initial state for that object or interface. Here the characterising predicate is given by the specification text and associated predicates on possible legal bindings *(invariants),* which must be satisfied by the initialisation schema in order to be classified as an instance of the object or interface type.

As an *operation schema* gives the characteristics of an action type, an instance of an action type is given by the occurrence of this *operation schema* or by the occurrence of another *operation schema* which is an extension of this *operation schema,* i.e. the extension includes the characterising features of the action type.

4.4.2.19 Template type (of an $<X>$)

In Z, an object/interface template type is a predicate that an initialisation schema leaves the object and associated interfaces in a valid initial state. Thus all state variables should be bound and all necessary predicates *(invariants)* satisfied. Often checking an object or interface's template type requires a proof to be made.

An action template type corresponds to a predicate that an action template, as given by an *operation schema,* can occur, i.e. is an instantiation of a given *operation schema.* Thus all instantiations are expected to satisfy the predicates on legal bindings of variables, as given in the *operation schema's* predicates.

4.4.2.20 Template class (of an $<X>$)

A template class of an $<X>$ is the set of all $<X>$'s that are instances of that $<X>$ template, where an $<X>$ may be an object, interface or action.

4.4.2.21 Derived class/Base class

Given two templates A and B in Z where A is an incremental modification of B and the instances of A and B are in a derived class/base class relationship respectively, the incremental modifications to B to produce A may include:

- adding or deleting state parameters;
- adding, deleting or modifying operations; or
- strengthening or weakening *invariants.*

4.4.2.22 Invariant

A predicate that a specification always requires to be true. Z allows *invariants* to be written down directly in schemas and *axiomatic descriptions.* Often *invariants* impose restrictions on the possible bindings that the variables in schemas or *axiomatic descriptions* can take. As such, an *invariant is* often used to restrict the possible behaviours given in a specification.

4.4.2.23 Precondition

The condition on the state of the system before the occurrence of an operation defined by an *operation schema* and on its inputs such that there exists a possible state after the performance of the operation and outputs which satisfy the *postconditions.* Z allows *preconditions* to be written down directly.

4.4.2.24 Postcondition

A predicate which describes the set of states that a given system can be in after the performance of an operation defined by an *operation schema. Z* allows *postconditions* to be written down directly.

4.5 Architectural semantics in ESTELLE

ESTELLE is a standardised FSL (see ISO/IEC 9074). A tutorial is included in that ISO Standard. Various tools supporting simulation as well as distributed implementation of ESTELLE specifications exist.

ESTELLE is based on extended finite state automata. A system is modelled through a hierarchically structured set of module instances, communicating in an asynchronous fashion via the exchange of messages on channels. The syntax of the language and the definition of data types and variables are based on ISO Pascal.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

In an ESTELLE specification, the externally visible interface of a module is defined in the *module header*, whilst the *module body* describes the internal structure and behaviour of the module. *Module instances* define *interaction points* through which they can send and receive messages. *Two interaction points* can be connected if they have been defined for the opposing roles of the same *channel definition*. A *channel definition* contains two *roles* for the respective ends of the *channel*. For each *role* it defines the messages (in ESTELLE called *interactions*) that can be sent. Such an *interaction definition* consists of a name together with a set of parameters. To each *interaction point*, a queue is assigned in which incoming messages are stored. A *module instance* can also have a common queue that is shared by several or all *interaction points*.

The structure of an ESTELLE specification is dynamic, i.e. *module instances* can be instantiated and released and interaction points can be connected and disconnected dynamically. The module instances of an ESTELLE specification underlie a strong hierarchical order. Each module instance can instantiate or release child *module instances*, or connect or disconnect their *interaction points*. The only way for a *module instance* to access sibling instances (or other *module instances* which are not its own children) is via exchange of *interactions* on *channels*.

ESTELLE was developed initially for the purpose of specifying communication services and protocols. ESTELLE does support encapsulation, but it does not contain the object-oriented features of inheritance or subtyping. Despite this, ESTELLE permits the expression of most of the ODP concepts. ESTELLE specifications are easy to read and since ESTELLE is a constructive technique, it is well suited for simulation and implementation.

The following subclause shows how the basic ODP concepts can be expressed using ESTELLE.

In this text, *italics* are used to denote ESTELLE concepts as defined in that ISO Standard. It should be noted that ESTELLE uses the notion of *interaction* to denote the messages exchanged between communicating *module instances*.

4.5.1 Basic modelling concepts

4.5.1.1 Object

An object is modelled in ESTELLE by a *module instance.*

4.5.1.2 Environment (of an object)

The part of the specification that is not part of the *module instance;* particularly the *parent instance* and the other instances that are connected to the module instances *interaction points* via *channels.*

4.5.1.3 Action

An action is represented in ESTELLE by the execution of a *WHEN-clause,* the execution of an action statement, a whole *transition,* or the execution of a *procedure.* Possible action statements are:

- *output;*
- *init;*
- *connect;*
- *attach;*
- *release;*
- *disconnect;*
- *detach;*
- *assignment statement.*

There are several types of **interactions**. The execution of an output-statement is an interaction as well as the execution of a *WHEN-clause.* Also, the action sequence consisting of the *output* of an *interaction* through an *interaction point* and its subsequent consumption through the execution of a *WHEN-clause* can be considered as one interaction. Another type of interaction is the update of an *exported variable.* All other actions are **internal**.

NOTE – Since ESTELLE *channels* possess infinite queues, the environment is always ready to participate in *interactions*.

4.5.1.4 Interface

There are two kinds of interfaces in ESTELLE:

- the first kind of interface is formed by the set of all *interactions* defined for the assigned *role* (in the corresponding *channel definition)* at an *external interaction point* of an object;
- the second kind is formed by the set of all *exported variables* of an object.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

In the first case, the set of interactions (constituting the interface) contains the set of *output* statements and/or *WHEN-clauses* that the object executes at the *interaction point.* In the second case, the set of interactions consists of all statements that read or write the *exported variables.* At this kind of interface, interactions are only possible between a *module instance* and its *parent instance.*

4.5.1.5 Activity

In general, an activity cannot be denoted explicitly, since it may span several objects. An activity is some related sequence of actions, e.g. a *transition,* or a *procedure* or *function,* if single statements are considered as actions.

4.5.1.6 Behaviour (of an object)

The behaviour of an object is determined by the set of all *transitions* of that object. Constraints on the circumstances in which the specified actions may occur are defined in the *transition clauses* (e.g. *FROM-Clause, PROVIDED-Clause*). An object may exhibit non-deterministic behaviour.

4.5.1.7 State (of an object)

The state of an object is composed of the following aspects:

- the *module instances control state*;
- the contents of *interaction point* queues;
- the values of internal and *exported variables,* and formal parameters of the *module instance*;
- the state of existing children *instances,* their connection structure and *exported variables.*

These aspects together determine the set of all sequences of actions *(transitions)* in which the *module instance* can take part.

4.5.1.8 Communication

Information is conveyed between objects in two different ways:

- through the *output* and subsequent reception of an *interaction* (an action sequence forming an interaction);
- through the reading and updating of an *exported variable.*

4.5.1.9 Location in space

Actions occur within *module instances,* or at the *interaction points* of *module instances.* Thus, the location in space of an action corresponds to the location in space of the associated *module instance.*

4.5.1.10 Location in time

In ESTELLE, time is only represented in the *DELAY-clauses* possibly associated with *transitions.* The only assumption made about the character of time is that it is increasing uniformly as the execution proceeds.

4.5.1.11 Interaction point

An interaction point is represented either by an *interaction point* or by the set of all *exported variables* of an object.

4.5.2 Specification concepts

The expression of the given specification concepts in ESTELLE is subject to difficulties, since it's support for object oriented concepts is limited.

4.5.2.1 Composition

- **of objects:** A *module instance* may be composed of a set of children *module instances.* On the top level, a specification may be composed of a set of *system instances.*
- **of behaviours:** When a *module instance is* a composition of children *instances,* their behaviour is either:
  - composed in parallel, interleaved, if the parent *module* is attributed *activity* or *systemactivity;* or
  - composed in parallel, synchronous, if the parent *module* is attributed *process* or *systemprocess.*

On the top level, the behaviour of the children *instances is* composed in parallel, if the specification is composed of a set of system instances.

4.5.2.2 Composite object

A *module instance* which is described through a set of children *module instances.*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.5.2.3 Decomposition

- **of objects:** The specification of a given object as a composition.
- **of behaviours:** The specification of a given behaviour as a composition.

4.5.2.4 Behavioural compatibility

There is no direct means to express behavioural compatibility in ESTELLE. However, the semantical basis of the language in terms of transition systems allows for the definition of behavioural compatibility and its verification.

4.5.2.5 Refinement

An object can be refined by substructuring it into cooperating children *instances*.

4.5.2.6 Trace

Traces can by obtained from the dynamic interpretation (execution/simulation) of an ESTELLE specification.

4.5.2.7 Type (of an < *X* >)

There is no way to explicitly formulate predicates in ESTELLE.

4.5.2.8 Class (of < *X* >)

Not supported.

4.5.2.9 Subtype/Supertype

Not supported.

4.5.2.10 Subclass/Superclass

Not supported.

4.5.2.11 < *X* > Template

An object template is represented by a *module body definition* together with the corresponding *module header definition*. Action templates are:

- *output;*
- *init;*
- *release;*
- *connect;*
- *disconnect;*
- *attach;*
- *detach;*
- *assignment statement;* and
- *WHEN-clause.*

Since there are two kinds of interfaces, an interface template is given:

- Through the corresponding *channel definition* (for sets of *interactions* at an *interaction point*). In this case, the associated interface behaviour can be specified through a child *module instance* that is *instantiated* and attached to the *interaction point* when the interface is created. If this is the case, the interface template contains the *module header* and *body definition* of this module.
- In the *module header definition* of the object (for *exported variables*).

4.5.2.12 Interface signature

An interface signature is represented in two ways as follows:

- the definitions of *interactions* that are contained in the interface (for a set of *interactions* at an *interaction point*);
- the assignment actions for the *exported variables*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4.5.2.13 Instantiation (of an <*X*> template)

An instantiation of an object is a *module instance* of the corresponding *module definition.* An instantiation of an interface is a particular *interaction point* (possibly with a specific child *module instance* attached to it, representing the interface behaviour) or the set of *exported variables* of a particular *module instance.*

4.5.2.14 Role

A *role is* associated with each *interaction point* declaration. *Interaction points* to be connected must possess opposite *roles* on the same *channel definition.* ESTELLE also supports the specification of different *module bodies* for the same *module header.* This allows for the possible choice among different behaviours when an object is instantiated. The interfaces of the object are the same regardless of the selected *module body.*

4.5.2.15 Creation (of an <*X*>)

Objects are created through the *init* action. Interfaces are created implicitly, when the object is created. There is no dynamic creation of interfaces. However, the creation of an interface as an *interaction point* may be modelled by the selection of an *interaction point* (out of an array of *interaction points*) with an appropriate *channel* and *role* associated. If a child *module* is specified to represent the interface behaviour, an *instance* of the *module* is *instantiated* and attached to the *interaction point* provided.

4.5.2.16 Deletion (of an <*X*>)

An object is deleted explicitly through the *release* action. Interfaces are deleted implicitly together with the object. If dynamic creation of interfaces is modelled as mentioned above (see 4.5.2.15), the deletion of an interface corresponds to *releasing* the attached *child module instance* (if any) and marking the *interaction point* as not representing an interface.

4.5.2.17 Introduction (of an object)

There is no way to introduce objects. In general, mechanisms not covered by the model may be captured using *primitive, external procedures* or *functions.*

4.5.2.18 Instance of a type

Since there is no subtyping/subclassing in ESTELLE, instances of a template are given by the instantiations of that template.

4.5.2.19 Template type (of a <*X*>)

For an object, the predicate which can be stated is that an object is an instance of the corresponding *module definition.* An interface (a set of *interactions* at an *interaction point*) is an instance of the corresponding *channel definition.* An interface represented by the set of all *exported variables* is an instance of the corresponding *module header definition.* An action is an instance of the corresponding action template.

NOTE – What can be stated in ESTELLE is the fact that an object, interface or action is an instantiation of a given template. Since the concept of subclass is not truly supported, only instantiations of a template can be identified as such, but not instances.

4.5.2.20 Template class (of a <*X*>)

The template class of an object is the set of all *instances* of the same module. The template class of an interface is the set of all *interaction points* defined using the same *channel definition* and *role.*

4.5.2.21 Derived class/Base class

Not supported.

4.5.2.22 Invariant

There is no way to explicitly formulate invariants in ESTELLE.

4.5.2.23 Precondition

The preconditions of a *transition* execution are stated in the *transition clauses* of the transition. The preconditions of an action inside a *transition block* are given by the preconditions of the transition together with the actions contained in the *transition block* that precede the action of concern.

4.5.2.24 Postcondition

The postcondition of a transition is defined through the *TO-clause* of the transition together with the actions in the *transition block.*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC

ICS 35.100.01

Price based on 29 pages

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale