INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.904
(12/97)

SERIES X: DATA NETWORKS AND OPEN SYSTEM COMMUNICATION

Open distributed processing

Information technology – Open distributed processing – Reference Model: Architectural semantics

ITU-T Recommendation X.904

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS AND OPEN SYSTEM COMMUNICATION

PUBLIC DATA NETWORKS	
Services and facilities	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalling and switching	X.50–X.89
Network aspects	X.90–X.149
Maintenance	X.150–X.179
Administrative arrangements	X.180–X.199
OPEN SYSTEM INTERCONNECTION	
Model and notation	X.200–X.209
Service definitions	X.210–X.219
Connection-mode protocol specifications	X.220–X.229
Connectionless-mode protocol specifications	X.230–X.239
PICS proformas	X.240–X.259
Protocol Identification	X.260–X.269
Security Protocols	X.270–X.279
Layer Managed Objects	X.280–X.289
Conformance testing	X.290–X.299
INTERWORKING BETWEEN NETWORKS	
General	X.300–X.349
Satellite data transmission systems	X.350–X.399
MESSAGE HANDLING SYSTEMS	X.400–X.499
DIRECTORY	X.500–X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600–X.629
Efficiency	X.630–X.639
Quality of service	X.640–X.649
Naming, Addressing and Registration	X.650–X.679
Abstract Syntax Notation One (ASN.1)	X.680–X.699
OSI MANAGEMENT	
Systems Management framework and architecture	X.700–X.709
Management Communication Service and Protocol	X.710–X.719
Structure of Management Information	X.720–X.729
Management functions and ODMA functions	X.730–X.799
SECURITY	X.800–X.849
OSI APPLICATIONS	
Commitment, Concurrency and Recovery	X.850–X.859
Transaction processing	X.860–X.879
Remote operations	X.880–X.899
OPEN DISTRIBUTED PROCESSING	**X.900–X.999**

For further details, please refer to ITU-T List of Recommendations.

INTERNATIONAL STANDARD 10746-4

ITU-T RECOMMENDATION X.904

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

Summary

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.

Source

The ITU-T Recommendation X.904 was approved on the 12th of December 1997. The identical text is also published as ISO/IEC International Standard 10746-4.

FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, the ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

Foreword

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.

.INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

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

0 Introduction

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.

1 Scope

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.

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 IS0/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.

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.

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).

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.

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.

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.*

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.

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, Obl, Ob2 -> CO

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.

– **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.

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.

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).

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

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.*

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 *al* and *a2*, hold the relation *now(al) <= 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*;

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.

–   **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.

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*.

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.*

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* ($\oplus$).

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.

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.

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.

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.

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.

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.*

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.*

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*.

ITU-T RECOMMENDATIONS SERIES

Series A   Organization of the work of the ITU-T

Series B   Means of expression: definitions, symbols, classification

Series C   General telecommunication statistics

Series D   General tariff principles

Series E   Overall network operation, telephone service, service operation and human factors

Series F   Non-telephone telecommunication services

Series G   Transmission systems and media, digital systems and networks

Series H   Audiovisual and multimedia systems

Series I   Integrated services digital network

Series J   Transmission of television, sound programme and other multimedia signals

Series K   Protection against interference

Series L   Construction, installation and protection of cables and other elements of outside plant

Series M   TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits

Series N   Maintenance: international sound programme and television transmission circuits

Series O   Specifications of measuring equipment

Series P   Telephone transmission quality, telephone installations, local line networks

Series Q   Switching and signalling

Series R   Telegraph transmission

Series S   Telegraph services terminal equipment

Series T   Terminals for telematic services

Series U   Telegraph switching

Series V   Data communication over the telephone network

Series X   Data networks and open system communication

Series Y   Global information infrastructure

Series Z   Programming languages