TECHNICAL REPORT

ISO/IEC TR 14369

Second edition
2018-01

Information technology — Programming languages, their environments and system software interfaces — Guidelines for the preparation of language-independent service specifications (LISS)

Technologies de l'information — Langages de programmation, leurs environnements et interfaces du logiciel d'exploitation — Lignes directrices pour l'élaboration de spécifications de service indépendantes du langage (LISS)

Reference number
ISO/IEC TR 14369:2018(E)

COPYRIGHT PROTECTED DOCUMENT

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Published in Switzerland

Contents

Page

Foreword

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

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This second edition cancels and replaces ISO/IEC TR 14369:1999 and ISO/IEC TR 14369:2014, of which it constitutes a minor revision.

The main changes compared to the previous edition are as follows:

— the reference section has been corrected/updated;

— editorial changes have been made to fully align with ISO/IEC Directives.

Introduction

This document is dedicated to Brian L. Meek in grateful recognition of his leadership and vision in the development of the concepts on programming language independent specifications, and his efforts in producing a set of documents in this area. Without his commitment this document never would have been published.

0.1 Background

This document provides guidance to those writing specifications of services, and of interfaces to services, in a language-independent way, in particular as standards. It can be regarded as complementary to ISO/IEC TR 10182, which provides guidance to those performing language bindings for such services and their interfaces.

NOTE 1 Here and throughout, "language", on its own or in compounds like "language-independent", means "programming language", not "specification language" nor "natural (human) language", unless explicitly stated.

NOTE 2 A "language-independent" service or interface specification can be expressed using either or both of a natural language like English or a formal specification language like VDM-SL or Z. In a sense, a specification can be regarded as "dependent" on VDM-SL, for example. The term "language-independent" does not imply otherwise, since it refers only to the situation where programming language(s) can otherwise be used in defining the service or interface.

The development of this document was prompted by the existence of an earlier draft IEEE Technical Report (IEEE TCOS-SCC Technical Report on Programming Language Independent Specification Methods, draft 4, May 1991). The TCOS draft was concerned with specifications of services in a POSIX systems environment, and as such contained much detailed POSIX-specific guidance; nevertheless it was clear that many of the principles, if not the detail, were applicable much more generally. This document was conceived as a means of providing such more general guidance. Because of the very different formats, and the POSIX-related detail in the TCOS draft, there is almost no direct correspondence between the two documents, except in the discussion of the benefits of a language-independent principles below. However, the spirit and principles of the TCOS draft were of great value in developing this document, and reappear herein, albeit in much altered and more general form.

NOTE 3 The TCOS draft has not in fact been published, as the result of an IEEE decision to concentrate activities in other POSIX areas.

0.2 Principles

Service or interface specifications that are independent of any particular language, particularly when embodied in recognized standards, are increasingly seen as an important factor in promoting interoperation and substitution of system components, and reducing dependence on and consequent limitations due to particular language platforms.

NOTE It is possible for a specification to be "independent" of a particular language in a formal sense, but still be dependent on it through inbuilt assumptions derived from that language which do not necessarily hold for other languages. The term "language-independent" here is meant in a much stronger sense than that, though complete independence from all inbuilt assumptions can be difficult if not impossible to achieve.

Potential benefits from language-independent service or interface specifications include:

— A language-independent interface specification specifies those requirements that are common to all language bindings to that interface, and hence provides a specification to which language bindings can conform.

— A language-independent interface specification is a re-usable component for constructing language bindings.

— A language-independent interface specification aids the construction of language bindings by providing a common reference to which all bindings can relate. Through this common reference it is possible to make use of pre-existing language bindings to language-independent standards

for common features such as datatypes and procedure calls, and to other language-independent specifications with related concepts.

— A language-independent service or interface specification provides an abstract specification of a service in isolation from language-dependent extensions or restrictions, and hence facilitates more rigorous modelling of services and interfaces.

— Language-independent service specifications facilitate the specification of relationships between one service and another, by making it easier to relate common concepts than is generally possible when the specifications are dependent on different languages.

— A language-independent interface specification facilitates the definition of relationships between different language bindings to a common service (such as requirements for interoperability between applications based on different languages that are sharing a common service implementation), by providing a common reference specification to which all the languages can relate.

— A language-independent interface specification facilitates the definition of relations between bindings to multiple services, including the requirements on management of multiple name spaces.

— A language-independent service or interface specification brings economic benefits by reducing the effort and resources needed to ensure compatibility and consistency of behaviour between implementations of the same service in different languages or between applications based on different languages using the same interface.

Information technology — Programming languages, their environments and system software interfaces — Guidelines for the preparation of language-independent service specifications (LISS)

1 Scope

This document provides guidelines to those concerned with developing specifications of information technology services and their interfaces intended for use by clients of the services, in particular by external applications that do not necessarily all share the environment and assumptions of one particular programming language. The guidelines do not directly or fully cover all aspects of service or interface specifications, but they do cover those aspects required to achieve language independence, i.e. required to make a specification neutral with respect to the language environment from which the service is invoked. The guidelines are primarily concerned with the interface between the service and the external applications making use of the service, including the special case where the service itself is already specified in a language-dependent way but needs to be invoked from environments of other languages. Language bindings, already addressed by ISO/IEC TR 10182, are dealt with by providing advice on how to use the two documents together.

This document provides technical guidelines, rather than organizational or administrative guidelines for the management of the development process, though in some cases the technical guidelines can have organizational or administrative implications.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
client
application (typically a program in some language) which makes use of a *service* (3.13)

Note 1 to entry: The term “service user” (3.15) is often used in a similar sense, where “client” more often implies the physical computer system on which the application is running, rather than just the application itself.

3.2
datatype
set of values, usually accompanied by a set of operations on those values

3.3
formal language
formal *specification language* (3.16)

3.4
interface
mechanism by which a *service user* (3.15) invokes and makes use of a *service* (3.13)

3.5
language
programming language, not *specification language* (3.16) or natural (human) language, unless otherwise qualified

3.6
language binding
specification of the standard interface to a *service* (3.13), or set of services, for applications written in a particular programming language

3.7
language-dependent
state of making use of the concepts, features or assumptions of a particular programming language

3.8
language-independent
state of not making use of the concepts, features or assumptions of any particular programming language or style of language

3.9
language processor
entire computing system which enables a programming language user to translate and execute programs written in the language, in general consisting both of hardware and of the relevant associated software

[SOURCE: ISO/IEC TR 10176:2003, 3.1, modified — the words “denotes the” at the beginning of the definition, and the two Notes to entry were deleted.]

3.10
mapping
defined association between elements (such as concepts, features or facilities) of one entity (such as a programming language, or a specification, or a standard) with corresponding elements of another entity

Note 1 to entry: Mapping when used as a verb is a process of determining or utilizing a mapping.

Note 2 to entry: Mappings are usually defined as being from one entity into another. A language binding of a language L into a standard S usually incorporates both a mapping from L into S and a mapping from S into L.

Note 3 to entry: Depending on what is being mapped, a mapping is not necessarily one-to-one. This means that mapping an element E from system A into an element E' of system B, followed by mapping E' back into system A, does not necessarily get back to the original E. In such situations, if a two-way correspondence is to be preserved, the execution of the mappings needs to include recording the place of origin and returning to it.

3.11
marshalling
process of collecting the actual parameters used in a procedure call, converting them if necessary, and assembling them for transfer to the called *procedure* (3.12)

Note 1 to entry: This process is also carried out by the called procedure when preparing to return the results of the call to the caller.

Note 2 to entry: Marshalling can be regarded as being performed by a service user when preparing input values for a service provider, and by a service provider when preparing results for a service user, the service concerned being regarded as the procedure being called.

3.12
procedure
subprogram which can return a value

Note 1 to entry: A procedure that returns a value is sometimes called a subroutine, a procedure that does not return a value is sometimes called a function.

Note 2 to entry: Some programming languages use different terminology.

3.13
service
facility, or set of facilities, made available to *service users* (3.15) through an *interface* (3.4)

3.14
service provider
server
computer system, or set of computer systems, that implements a *service* (3.13) and makes it available to *service users* (3.15)

Note 1 to entry: In this definition, "computer system" means a logical system, not a physical system; it can correspond to part of all of one or more physical computer systems.

Note 2 to entry: The term "server" is often used in a similar sense, though sometimes implying a physical computer system that has no other function than to provide its service

3.15
service user
application (typically a program in some language) which makes use of a *service* (3.13)

Note 1 to entry: The term "client" (3.1) is often used in a similar sense, though sometimes implying the physical computer system on which the application is running, rather than just the application itself.

3.16
specification language
formal language for defining the semantics of a service or an interface precisely and without ambiguity

3.17
unmarshalling
process of receiving and disassembling transferred parameters, and converting them if necessary, to prepare the values for further use

Note 1 to entry: This process is carried out by the called procedure on receipt of the actual parameters for the call, and by the caller on receipt of the returned results of the call.

Note 2 to entry: Unmarshalling can be regarded as being performed by a service provider when receiving input values from a service user, and by a service user when receiving results from a service provider, the service concerned being regarded as the procedure being called.

3.18
Z
<mathematics>complex numbers

Note 1 to entry: See ISO/IEC 10967-1.

3.19
Z
type of formal *specification language* (3.16)

Note 1 to entry: It is pronounced "zed".

Note 2 to entry: See ISO/IEC 13568.

4 Abbreviated terms

GPD general-purpose datatypes, as defined in ISO/IEC 11404:2007

LID language-independent datatypes, as defined in ISO/IEC 11404:1996; the LID specifications used in this document are identical to the corresponding specifications in ISO/IEC 11404:2007 (GPD)

LIPC language-independent procedure calling, as defined in ISO/IEC 13886

RPC remote procedure call, as defined in ISO/IEC 11578:1996

5 Overview

5.1 Services, interfaces, service providers and service users

The concept of a "service" is a very general one. In some contexts it is customary to use it in a restricted sense, e.g. when talking about "service industries" as contrasted with "manufacturing industries". Despite such usages, almost any activity or behaviour can be regarded as a "service", if it serves some useful purpose to do so (for example, manufacturing spoons can be regarded as a service for those needing spoons).

With the concept of a service come the concepts of a "service provider" and a "service user". The provider performs the activity that constitutes the service; the user is the customer or the client for the service, for whom the service is performed. In the information technology field, the "client-server model" incorporates these concepts: the server provides, the client uses.

Between the service provider and the service user is an interface that allows them to communicate. The service user communicates through the interface the requirement for the service, and any relevant information (e.g. not only the need for spoons, but the number and size of spoons required), and the service provider communicates through the interface the response to the order for the service, and any additional information or queries (e.g. the spoons can be delivered in six days, do you want silver spoons or plastic spoons?). In the information technology field, such interfaces are usually explicit, realized in hardware or software or both. In the world in general, they are sometimes explicit, but sometimes subsumed in more general human or other interactions.

This distinction between provider and user (client and server) should not be assumed to correspond to identifiable distinct entities. The distinction, and the service interface, can be purely notional, and possibly not normally thought of in that way. The service itself can similarly not correspond to a distinct, separate activity, and again possibly not normally thought of as such; it can be subsumed in some other activity or group of activities, and can possibly be implicit.

Hence, for example, in a transaction between two parties, each one can be providing a service for the other: each is a client, and each a server. In another context, the provider is providing the service to itself; the provider is also the user. Though it is possible to subdivide the provider/user into a provider part and a user part when considering provision of the service, this can be inconvenient in other respects.

In summary, "client" and "server", are roles that are carried out, rather than elements that necessarily need to be implemented separately. Though the term "client-server" is sometimes used in the information technology field in ways that are more specific than it is used here, it is important not to carry over assumptions from particular client-server models when reading this document. It is even more important not to assume that implementation of any service, in the sense used here, needs to be done using a client-server model.

5.2 Information technology services

The history of information technology has many instances of the technology, or a product, being used for very different purposes and in very different ways from those originally envisaged. The kinds of

service that information technology and products provide have continually expanded and diversified, and this is still continuing.

It is as common in information technology as in the outside world for the term "service" in particular contexts to be used in a rather specific way. The history of the technology suggests that, for the purposes of formulating guidelines about services, the term should instead be used as generally as possible.

This document has adopted this very general approach to the concept of "service". It is therefore important that, when using this document and the guidelines it contains, no presuppositions should be made about what a service is, or about how and by what it is provided or how and by what it is used. The guidelines should be interpreted and applied in that light.

This document does, however, carefully distinguish between the service itself, and the interface used to communicate with it. In some usages the term "service" includes the interface, and the interface can be embedded in the service and its specification (as in the phrase "all parts of the service"). However, logically they are distinct, and this logical distinction is maintained throughout this document.

Services in the most general sense often simply evolve naturally, but information technology services are usually consciously designed. They are also often built from explicit specifications, though some are developed ad hoc. Whichever the case, it is useful to make a clear logical distinction between service providers, service users, and the interface between them, even if, when implemented, one or both of these distinctions are purely notional, and are not embodied in identifiable and separable artefacts like particular hardware components or particular blocks of code. Indeed, thinking about service provision in such a way, in an environment that is normally regarded as a more integrated whole, can help to improve a specification, or at least to test it and verify its validity.

This is especially so in the increasing number of cases where information technology environments and services, though originally conceived as self-contained, need to interact with external environments and services, many of which need the distinction between providers and users to be made explicit. An instance of direct relevance to this document is where interacting entities are based upon different languages and hence different sets of underlying assumptions.

5.3 Services and language independence

The term "language-independent service (or interface) specification" means "language-independent specification of a service (or interface)", not "specification of a language-independent service (or interface)". Hence a language-independent specification of a service does not imply that the service itself is "language independent" in the sense intended here. The service specified can be relevant only to environments of particular languages.

NOTE 1 The implementation of a service which meets the specification uses some language or other, if only machine language, and so, is in a sense, "dependent" on that language, but that is not the sense intended here.

Also, a language-independent specification of an interface does not imply that the service interfaced to is either itself "language independent", or specified in a language-independent way (though it can be).

A trivial instance is that of a language processor for a particular language providing a service by executing a program in that language. For one of the long-established languages (like Cobol or Fortran) the interface is the provision of input data and the output of results. The language was designed for particular forms of input and output media, presumed under the control of a human user. However, a language-independent interface specification can define the input and output in such a way that the data can come from, and the results be returned to, some other system, in general using a different language.

In a simple case like that, the user system and the interface are distinct and not closely coupled. The interface can be implemented as a "black box" which acts in the same way that a human interpreter would for two people with different languages conversing: it takes input from the client and translates it into the equivalent input for the service, and takes the output from the service and translates it into the equivalent output for the client.

In the more general case, it is possible that the interface needs to be embedded in the client system so that it appears to be integrated in that host environment. That environment can require invocations of the service to be expressed in more meaningful terms, not limited to the data transmitted to it and the results expected from it.

NOTE 2 One example involves the functional standards for graphics. In some languages the most suitable invocation method is a procedure call to an external library, while in others the most suitable method is the use of additional commands (keywords).

Both the simple and the general case are referred to as "binding" to the interface, though the binding is much tighter in the general case. A "language binding" to the interface binds a particular programming language (in general not the same programming language as the one used by the server), so that programs written in that language can have access to the service. A good language binding allows language users to use a style of accessing the service which is familiar to them, and also accords with official standards for the language.

ISO/IEC TR 10182 provides guidance to those performing language bindings and writing standards for them. This document provides complementary guidance to those specifying service interfaces in a language-independent way, and writing standards for them.

A useful way of looking at language independence is by considering levels of abstraction. The various elements of programming languages can be regarded as existing at three levels of abstraction: abstract, computational, or representational, where the middle, computational level can be divided into two sublevels, linguistic and operational. The linguistic elements are regarded as instantiations at the computational level of the abstract concepts, while the operational level deals with manipulation of the elements, which inevitably looks "downwards" to the realization of the elements in actual, processible entities at the representational level.

NOTE 3 The representational level does not necessarily mean the physical hardware level, or the logical level of bits and bytes; see the discussion in 5.5.2.

5.4 Language-independent specifications

As the preceding discussion has shown, a language-independent specification can be a service specification, specifying the service itself, or be an interface specification, specifying how the service is accessed by clients. It can cover both.

This document is concerned primarily with specification of the interface to the service, rather than of the service itself. The service can be predefined in a language-dependent way. How a service is specified is likely to depend to some extent on the nature of the service and its application area, so guidelines on specification of the service are definitely outside the scope of this document. However, where it is wished to produce a language-independent specification of a service, so that it can be implemented in a variety of different languages, then the guidelines presented can be useful, directly or indirectly. For example, they draw attention to factors that should be borne in mind, and it can then be possible to adapt them to the particular circumstances.

This document therefore provides guidelines applicable in the following cases:

— specifying a service interface;

— specifying both of a service interface and the corresponding service itself, together;

— specifying from scratch (i.e. without anything pre-existing to base it on);

— specifying on the basis of an existing (probably language-dependent) service;

— specifying on the basis of an existing (language-dependent) binding.

Guidelines are grouped under various headings, dealing with different aspects. As far as possible each group is independent, in the sense that they can be referred to without necessarily working through preceding groups. Any necessary cross-references are provided.

5.5 Problems of language dependence and inbuilt assumptions

5.5.1 General

Producing a language-independent specification can present many problems, especially if starting from an existing service which was not originally designed to be language independent – typically, a service designed in and for a particular language environment. If a service is specified in the "wrong" way – it is possible that it was not "wrong" in its original context – it can make producing a language-independent interface very difficult. In particular, it can depend on explicit or (more likely) implicit assumptions about the language that applications using the service will be written in. Languages that are similar in character to the original one can have less problems, but a language-independent interface specification needs to be able to accommodate different language styles. This is one of the greatest challenges in developing language-independent specifications, whether for services or for interfaces.

NOTE Examples of styles of language are: procedural, declarative, functional, interpretive, object-oriented, etc., and these are not necessarily mutually exclusive.

Such problems can still occur even if the service concerned is a new one yet to be developed. Since most service developers tend to come from a particular language environment, it is all too easy, even when consciously attempting to produce a language-independent specification, to carry over implicit assumptions from that environment, simply because they are implicit and hence rarely questioned or even noticed.

5.5.2 Representational assumptions

An important class of language-dependent assumptions is that of representational assumptions. Some languages have explicit or implicit models of how language elements are represented at the hardware level, either physically or logically. Simple instances are storage of numerical values or of aggregate datatypes such as indexed arrays or character strings, or numbers of datatype Complex (assumed to be represented by two numbers of datatype Real, for the cartesian real and imaginary parts).

Such models tend to become implicit for those used to that language environment, even when the language definition makes the model explicit. Users of the language get so used to that model that they take it for granted. It is all too easy for such assumptions to get carried over into what is intended to be a language-independent specification.

Representational assumptions are not confined to the hardware level, but can occur at more abstract levels too; for example, a supposedly "language-independent" specification can use an integer datatype for a value which logically is not, or need not be, an integer. The fact that virtually all languages have an integer datatype or an equivalent is not relevant; it can be that the original language has used the integer datatype because it was the best or only choice, but other languages can have alternatives which the original language did not. A language-independent specification should avoid requirements that constrain how things should be represented, and concentrate upon what should be represented.

NOTE It is possible for a language-independent specification to be developed which is explicitly concerned with the representation of language elements. For such a specification, it is possible that not all the principles outlined above apply, although some can still be relevant.

5.5.3 Implementation assumptions

Representational assumptions are a specific form of implementation assumption, though not all implementation assumptions are language-dependent. Service designers make implementation assumptions when they take it for granted that a particular implementation approach will be adopted. A simple example is the assumption that the service will be invoked by a procedure call or, even more specifically, will use procedure calls using a parameter passing mechanism of a particular kind.

Implementors of language-independent service specifications should not be required to adopt a particular implementation approach. Instead, the specification should require only what is needed for

the service, or is needed to ensure that different implementations will be mutually consistent or (if interoperability is required) interact with one another correctly.

6 Guidelines on strategy

6.1 General

The discussion in Clause 5 above shows that a large number of factors need to be taken into account when producing a language-independent service specification. This clause provides guidance on how to go about the task.

The guidelines that follow are divided into general guidelines (see 6.2) and more specific ones. Some of the more specific guidelines are in fact similar to one another, appearing in various modified and specific forms under various headings, and can become "general" guidelines. The apparent duplication increases the length of the document, but is intended to reduce the amount of interpretation and adaptation that are needed in particular circumstances, and to emphasize the relevance in particular contexts. It also allows different Notes, specific to the context, to be appended.

6.2 General guidelines

6.2.1 Guideline: Dependence of the interface on the service

A service specification should be designed with the requirements for the language-independent interface in mind.

NOTE If a service is specified in the wrong way, it can make the production of a language-independent interface very difficult, in particular when explicit or implicit assumptions are made about the languages that applications that will use the service will be written in.

An example is assuming a particular method for invoking the service, e.g. the use of object classes, or the use of low-level procedure calls (i.e. using only simple datatypes for parameters).

6.2.2 Guideline: What to do when there are interoperability, concurrency, or time constraint issues

Issues relating to interoperability with other services, or concurrency, or time constraints of other kinds can affect language-independent service and interface specifications. If this is the case, the nature of such issues makes it vital that they be addressed first, with the remainder of the service being designed later, around the aspects handling those issues.

NOTE 1 Interoperability, concurrency, and time constraint issues can often cause difficulties, compared with which other issues are comparatively straightforward to deal with. They can also place requirements or constraints on other aspects of the service. It therefore aids the design process to address those issues first. For example, if a service is to have multiple clients, this is best taken into account very early on.

NOTE 2 Guidelines on interoperability appear in Clause 10, and guidelines on concurrency appear in Clause 11.

6.2.3 Guideline: Use of marshalling/unmarshalling

When specifying the way that values are communicated across the interface between the application using the language binding and the service, the marshalling/unmarshalling approach used in LIPC in relation to passing of parameters can prove useful.

NOTE The marshalling/unmarshalling concept for communicating values is sufficiently general to be of use even when the service and its interface do not involve explicit procedure calling.

6.2.4 Guideline: Recruiting expertise from a variety of backgrounds

When developing a language-independent specification, every attempt should be made to recruit the involvement of, or to obtain input from, language experts from a variety of backgrounds, and also experts in language-independence issues. In any event, before the language-independent specification is finalized, arrangements should be made to get a complete draft reviewed by experts of that kind from outside the group designing the specification.

NOTE Because of the particular nature of the problems involved in achieving language independence, it is preferable to choose language experts who have some experience of binding to language-independent specifications, and/or who are familiar with other languages than their own main language.

6.3 What to do if starting from scratch

6.3.1 General

It is rare for the designer of a language-independent service or interface specification to be able to "start from scratch", i.e. to be able to design without having to take into account an existing (and usually language-dependent) service or interface which is already in use (and with which compatibility is required, or expected even if not required). However, for completeness, this document does need to cover the possibility. Furthermore, guidelines on what ideally should be done can serve as a benchmark against which to measure what has actually been possible, given the constraints that a pre-existing service or interface can have placed upon the design. In principle, they can even establish that it would be preferable to treat the pre-existing version simply as a prototype to be discarded.

6.3.2 Guideline: Avoidance of implementation assumptions

When designing a language-independent service specification, representational or other implementation assumptions should be avoided.

NOTE 1 Languages differ greatly in character so a form of implementation suitable for one can be quite unsuitable for another. Furthermore some languages themselves make explicit or implicit representational or other implementation assumptions, not always consistent with those in other languages. Language-independence is therefore best assisted by avoiding all such assumptions, however attractive they can be in other respects.

NOTE 2 This guideline reappears in various more specific forms throughout this document and the general question has already been introduced in 5.5. This has been done deliberately, both to stress its importance and to aid in interpreting the guideline in various contexts.

6.3.3 Specifying the service in language-independent form

6.3.3.1 Guideline: Allowing for different approaches

When specifying the service in language-independent form, it should not be assumed that implementations in every language will use the same approach, and implementations should not be required to adopt a particular approach. Instead, the specification should require only what is needed for the service, or is needed to ensure that different implementations will be mutually consistent or (if interoperability is required) interact with one another correctly.

NOTE 1 It is not necessary to use an implementation model to specify requirements, whether these are needed to provide the service itself, to ensure mutual consistency, or to ensure interoperability. Such requirements are best expressed in an abstract, language-independent way.

NOTE 2 Guidelines on interoperability appear in Clause 10.

6.3.3.2 Guideline: Documenting external constraints and minimising their impact

If there are external constraints which the service is required to satisfy, these should be carefully examined to assess their impact, whether on implementation strategies for the service, or on the

interface. The relevant aspects of the service should then be specified in a way which minimizes the impact of the constraints. The external constraints (including the rationale for their presence), and the steps taken in the specification to cope with them, should be documented.

NOTE 1 Particular attention is needed in the case of constraints which seem to require things to be done in accordance with some implementation model. In many cases it is possible to avoid passing on these implementational requirements by absorbing them into the service, for example by internal conversions.

NOTE 2 In general, it is preferable to leave as much as possible to implementations to handle as best they can, provided this can be done without compromising either the integrity of the service or of language independence.

NOTE 3 Sometimes, the cost of an extra conversion interface is justified by gains elsewhere, for example in terms of resource, safety or reliability.

6.3.3.3 Guideline: Allowing for different binding methods

When specifying the service in language-independent form, it should not be assumed that the interface will use or specify a particular binding method; rather, the specification should be neutral with respect to binding methods.

6.3.4 Specifying the interface to the service in language-independent form

When specifying the interface to the service in language-independent form, it should not be assumed that a particular binding method will be used by every language, and use of a particular binding method should not be required. The specification should require of bindings only what is to be passed across the interface, not how it should be passed.

NOTE 1 Well-designed language bindings make maximum use of the facilities of the language. Assuming or requiring a particular binding method can lead to suboptimal bindings to the service and in extreme cases can make it impossible to specify an adequate binding.

NOTE 2 Language bindings are also designed for many different purposes, and it can create many problems if a binding to one service is required to be markedly different from other bindings.

6.4 What to do if starting from an existing language-dependent specification

6.4.1 General

The task of producing a language-independent service or interface specification from an existing language-dependent specification is one of "reverse engineering". In general, it can be expected that the original language-dependent specification treats the service, the interface, and the language binding as one, and does not, deliberately, keep the different aspects separate. For a language-independent specification, whether for a service or for an interface, it is necessary to ensure that these different aspects are kept separate. Guidelines on identifying significant language-dependent aspects are provided in 6.4.2. The conversion of language-dependent features to language-independent form is addressed in 6.4.3, and 6.4.4 addresses the consequences for language bindings. In 6.4.5, the situation is addressed where only the interface specification but not the service specification is to be made language independent.

NOTE If more than one language-dependent specification exists, the following guidelines still apply, but the results for each binding should be checked against each other. Inconsistencies can be very helpful in reaching an appropriate language-independent formulation

6.4.2 General guidelines

6.4.2.1 Guideline: Identifying implementation assumptions

Any representational or other implementation assumptions in the original language-dependent specification should be carefully reviewed, and any which are derived from the particular language used, rather than dictated by the semantics of the service, should be identified.

6.4.2.2 Guideline: Identifying language-dependent terminology

The terminology used in the original language-dependent specification should be carefully reviewed from the language-independent point of view, to see if it is derived from the terminology of the particular language rather than from the service.

6.4.2.3 Guideline: Identifying aspects specified at the wrong level of abstraction

The language-dependent specification should be carefully reviewed for features which are specified at a level of abstraction that is inappropriate for the language-independent version. The review should in particular search for those at too low a level which do not involve overt representational or implementation assumptions as discussed in 6.4.2.1, but arise from the way the service has been conceived in the original language environment. Attention should, however, also be paid to any at too high a level, which can take the form of features being left under-specified because the missing aspects are taken for granted in that language environment, or because the language definition leaves such aspects implementation dependent.

NOTE 1 The concept of levels of abstraction is discussed in 5.3.

NOTE 2 An example of too low a level is specifying the service in terms of independent entities when in fact they naturally form fields of a Record datatype.

NOTE 3 An example of too high a level is specifying a datatype without defining permitted or required ranges of values of the datatype.

NOTE 4 When rectifying inappropriate levels of abstraction, over-compensation is to be avoided.

6.4.2.4 Guideline: Identifying aspects derived from the language rather than inherent to the service

The language-dependent specification should be carefully reviewed for features which are not inherent to the service, but whose inclusion seems to have been prompted by the nature of the implementation language and its facilities. Particular attention should be paid to any such inessential features which can be difficult to provide in some other languages. Attempts should be made to discover how heavily users of the original specification use these features.

NOTE 1 Sometimes it is useful to include such features because they are useful elsewhere in the language, for purposes unrelated to the service itself.

NOTE 2 It is worthwhile considering including features of this kind in the specific language binding for the language concerned; though strictly inessential to the service, there can nevertheless be a continuing demand for them from that language community, which cannot readily be satisfied in another way (e.g. by the provision of separate services). If that is the case, it is assumed that the conformity rules permit bindings to include these supplementary features, though maybe not for all languages.

NOTE 3 However, it is possible that such features are rarely used by users of the original specification, in which case the opportunity can be taken to remove them, or to designate them as “obsolete”, to be removed at the next revision.

6.4.2.5 Guideline: Identifying desirable but absent features

The language-dependent specification should be carefully reviewed to see if there are any features which would be desirable, but which are in fact absent from the original (e.g. because it was not possible to provide it conveniently or efficiently in the original language, or where they are implicit in that language and did not need to be spelled out). Any such features should be studied, to see if they should now be added, either as options or as mandatory requirements.

NOTE 1 Such “absentee features” can occur because the original language can have been chosen for reasons other than being ideal for the purpose of providing the service.

NOTE 2 The original language can be subject to revisions which remove the previous difficulties in providing a feature.

NOTE 3 It is necessary to pay special attention to the binding to the original language.

6.4.3 Converting an existing language-dependent specification of the service into language-independent form

6.4.3.1 Guideline: Avoiding undue dependence on the original language-dependent version

While it is desirable and even necessary to use the original language-dependent specification as a guide when developing a language-independent specification from it, the detailed form and content should not necessarily be dictated by the detailed form and content of the original. In particular, changes that correct weaknesses in the original, and especially changes that enhance language independence, should be seriously considered, and if possible included in the specification, with due regard for the impact on existing implementations using the original specification. However, change should be avoided if what is in the original is adequate for the purpose, and does not adversely impact language independence, even if a change would appear to be an improvement.

NOTE 1 The guidelines in 6.4.2 show how to identify aspects of the original specification that are to be considered for changes.

NOTE 2 When assessing the impact of changes on existing implementations using the original specification, the guidelines on revisions in Clause 18 can be helpful. See 6.4.3.5.

NOTE 3 A change that does not correct a weakness but "would appear to be an improvement" can be contemplated, if the development of the language-independent specification is accompanied by a parallel revision of the original specification.

6.4.3.2 Guideline: Recasting scope of specification

In the light of the results of following previous relevant guidelines, the scope of the specification should, if necessary, be recast at as high a level of abstraction as is possible while remaining consistent with the nature of the service.

NOTE 1 It is possible that recasting the scope of the specification is not necessary: it can be sufficient to keep it at the same level of abstraction but to remove anything not at that level.

NOTE 2 Examples of too low a level of abstraction would be specifying a representational model of integers when a non-representational one is sufficient, or specifying use of an integer datatype for a value which logically is not, or need not be, an integer.

NOTE 3 An example of a level of abstraction higher than is consistent with the nature of the service would be specifying an integer datatype without stating a minimum range of values, when such a minimum range is needed by services for interoperability purposes.

6.4.3.3 Guideline: Revising language-dependent terminology

Language-dependent terms used in the original specification should be changed if necessary, e.g. if they are likely to be misinterpreted in a different language environment. If not changed, they should be clearly explained, for the benefit of those not familiar with the original language or specification.

NOTE 1 For the benefit of those familiar with the original language-dependent specification, any such changes of terminology are to be listed, and the reasons for the change explained.

NOTE 2 If a term is particular to the original language and not encountered elsewhere, confusion can still occur if language environments use a different term for the same or a similar concept.

6.4.3.4 Guideline: Conversion of datatypes and procedure calling

A suggested strategy for converting a language-dependent specification into language-independent form is to start by converting the datatypes of values used, together with all the required operations on the data, including input–output. If any procedure calling appears in the original specification, conversion of that should then follow. Conversions should be based on what the service needs, rather than what was chosen in the original specification, since those choices are likely to be language-dependent.

NOTE 1 Since all services handle data values of some kind, and many use procedure calling as a mechanism, converting these first can help the rest to fall into place more easily.

NOTE 2 It is not sufficient merely to use a binding of the original language to LID and leave it at that; it is possible that a particular choice of datatype has been dictated by what the language had available, and it is possible that it is not be the best language-independent choice (see Clause 12).

NOTE 3 For similar reasons, it is also insufficient to use a binding of the original language to LIPC; particular choices of procedure parameters and passing mechanisms have been limited to those the language had available.

6.4.3.5 Guideline: Documenting language-dependent aspects

The relationship between the original and the language-independent specifications should be fully explained (e.g. in an annex) and all language-dependent assumptions or features that have been recast or removed should be documented. A migration path to allow existing language-dependent implementations to be revised in line with the language-independent version should be provided.

NOTE With suitable adaptation, the revision guidelines in Clause 18 can be used to help in specifying a migration path for existing implementations.

6.4.4 Converting an existing implicit interface into an explicit language-independent interface

6.4.4.1 General

It is possible in some cases that the interface to an existing service (language-independent or not) has not previously been defined explicitly, but exists only in the form of a “binding” to one language, this binding itself probably being implicit rather than explicit. This subclause provides guidance on coping with that situation. Mostly, the guidelines below are simply reinterpretations of previous guidelines, adapted to suit those particular circumstances.

6.4.4.2 Guideline: Aspects derived from the language

Any aspects of the language binding which are derived from the particular language, rather than dictated by the need to interface to the service, should be identified, and replaced by language-independent equivalents where appropriate.

NOTE 1 It is likely that the revised binding, for the original language to the language-independent interface, is able to continue to include these aspects, if only as optional language-specific additions.

NOTE 2 Language-dependent aspects can include things like the structure of the binding document, as well as simply the features of the language concerned. Language independence can involve complete restructuring, including the revised binding for the original language. In that case, extra guidance can be needed, e.g. in the form of an informative annex.

6.4.4.3 Guideline: Absent features

The language binding should be carefully checked, or rechecked, to see if there are any aspects of the service, relevant to the interface, which are in fact absent from it (e.g. because it was not possible to

access them conveniently or efficiently from the language concerned, or because they were irrelevant for the language).

NOTE A feature can be absent from the binding simply because the language already contains that particular feature as part of its own service. The revised binding, for the original language to the language-independent interface, will still be able to continue to omit that feature, for the same reason.

6.4.4.4 Guideline: Identifying aspects not required by the service

Any aspects of the language binding which are inessential to providing an interface to the service should be identified, reviewed, and considered for removal from the language-independent interface specification.

NOTE Although, in some cases, this guideline and guideline 6.4.4.2 overlap, the presumption is normally that inessential features are removed. The aspects referred to here are not so much "derived from the particular language" but are service-related facilities seen to be of use to the language community concerned, or arise from inbuilt assumptions about how or why the service is used within that community. However, the possibility exists that these "inessential" features, in some form, nevertheless prove of value to users from other language communities, and are therefore not to be discarded without due consideration.

6.4.4.5 Guideline: Avoiding assuming the binding method

The language-independent interface specification should not be based on the assumption that the (explicit or implicit) binding method used for the original language will be used for all other languages.

NOTE 1 The binding method used for the original language is inevitably chosen to suit that particular language, and it is possible that it is not the most appropriate for all. In general, a well-written language-independent interface specification permits the use of any binding method.

NOTE 2 ISO/IEC TR 10182 provides guidance on binding methods.

6.4.5 Specifying a language-independent interface to a service whose specification is language-dependent

It is quite possible that the existing service for which a language-independent interface is needed is itself specified in one particular language and is therefore, at least potentially and possibly necessarily, language dependent. This subclause provides guidance on coping with that situation. The guidelines below are primarily logical extensions or adaptations to others elsewhere in this document.

NOTE A service can be necessarily language-dependent when it depends on specialist facilities which are available only in one specialist language (for example the database facilities in SQL) and which in practical terms cannot sensibly be simulated in another available language. It can be language-dependent in a less restrictive sense when only a small minority of languages have suitable facilities (for example, knowledge-based systems that can be implemented readily in languages such as Prolog or Lisp but only with great difficulty in others).

6.4.5.1 Guideline: Protecting bindings from language dependence

The language-independent interface should be specified in a way that protects language bindings as much as possible from the language dependence of the service. This can be done by specifying the limitations and assumptions arising from the language of the service, and providing the necessary conversions within the interface separately, rather than propagating them to the bindings.

7 Guidelines on document organization

7.1 General

A language-independent service specification can be a very complex document, depending on the complexity of the service and the scope of the specification. This subclause provides guidance on how to organize the material needing to be covered.

NOTE 1 If the document structure of bindings is intended to follow that of the language-independent service specification, it is necessary to consider how to keep them in line during revision.

NOTE 2 Guidelines on document organization for language bindings are in 17.4 and 17.5.

7.2 Guideline: The general framework

7.2.1 General

The language-independent service specification should be designed to include the parts in the checklist that follows in 7.2.2 (though it should not necessarily be confined to only to the parts listed). Where a particular part seems not to be necessary in a given case, allowance should still be made for its possible future inclusion, e.g. as a result of a later change in the scope of the specification, or of a development of the service concerned.

NOTE Here the term "part" is used in the everyday general sense; it does not imply the need for a separate "Part" of a standard in the formal sense. See 7.3.

7.2.2 Checklist of parts for inclusion

1) If the scope of the specification includes the semantics of the service, a definition of those semantics, including rules for conformity of implementations.

2) If the scope of the specification does not include the semantics of the service, an explanation of how the semantics relate to the content of the document.

   NOTE It is necessary to include a reference to the definition of the semantics, and perhaps also to include a brief summary of the semantics, e.g. in an informative annex.

3) If the scope of the specification includes the interface to the service, a definition of that interface, including rules for conformity of implementations.

4) If the scope of the specification does not include the interface to the service, an explanation of how the interface relates to the content of the document.

5) In the case of implementations of the interface, a specification of requirements on name correspondence between names used in the interface specification and names used in a calling program.

   NOTE 1 This part entails requirements on language bindings to the interface.

   NOTE 2 Even when the application of LID and LIPC is sufficient to cover all functionality, name correspondence requirements are still likely to be needed.

   NOTE 3 A normative annex can be appropriate for specifying name correspondence requirements.

6) The specification of all further requirements on standard-conforming implementations (such as fault detection, reporting and handling; provision of implementation options to the user; documentation; validation; etc.), and of rules for conformity.

   NOTE It is probably necessary to specify such further requirements separately for implementations of the service and for implementations of the interface.

7) The conformity rules of the language bindings to the language-independent service specification.

8) A description, as well as a reference, and if necessary a complete specification, of any formal specification language used in a) or c), and for each case an annex containing a summary of the formal definitions.

9) One or more annexes containing an informal description of the service and of the interface, a glossary, guidelines for service users (on implementation-dependent features, documentation available, etc.), and a cross-referenced index to the document.

NOTE 1 In general, each informal description should appear even if its full definition is within the scope of the specification and is included in the document. However, it is possible that it is not necessary for some simple services.

NOTE 2 Where the full definition of either the service or the interface is not within the scope of the specification, and hence does not appear in the document, an informative clause can be more appropriate than an annex, if only to emphasize its importance. This is particularly the case for the specification of the interface when the specification of the service appears elsewhere, and in the case of language bindings.

NOTE 3 A normative annex can be appropriate for specifying name correspondence requirements.

10) An annex containing one or more checklists of any implementation-defined features.

11) An annex containing guidelines for implementors, including short examples where appropriate.

12) An annex providing guidance to users of the language-independent service specification on questions relating to the validation of conformity, and any specific requirements relating to validation contained in 1), 3), 5) and 6) above.

13) In the case where the language-independent service specification is a revision of an earlier version, an annex containing a detailed and precise description of the areas of incompatibility between the old version and the new version.

14) Material that forms a tutorial commentary containing examples that illustrate the use of the service can optionally be included as an annex or be published as a separate document.

7.3 Guideline: Production and publication

Although 7.2 does not imply that the "parts" of the specification should be in a number of physically separate documents, for a very complex service this should be considered, especially if different aspects (the service, the external service, language bindings, etc.) are likely to be implemented separately.

NOTE 1 This means that a language-independent service specification published as an International Standard would be published as a set of separate Parts.

NOTE 2 Publication in a set of separate documents implies a need for careful cross-referencing, possibly by including in each one informative summaries or extracts from others that are relevant, or needed for understanding. This implies some duplication, the need to keep changes and revisions consistent across the set, and consequently an increase in overall length and of effort involved. Such costs are to be carefully weighed against the advantages of dividing the whole into more manageable pieces.

7.4 Guideline: Document organization when starting from a language-specific specification

Where a language-independent service specification is being developed, which is based on an existing language-specific specification, and changes to the original document organization seem desirable in the language-independent case, the benefits of such changes should be weighed against the value of maintaining a close correspondence between the two, to aid comparison and review.

NOTE 1 Factors to be considered include the extent of use of the original language-specific specification and hence the volume of review expected from those familiar with the original version, and how soon the original specification is replaced by a binding of the language-independent specification to the language concerned.

NOTE 2 It can help to apply similar criteria to those in Clause 18 on revisions, when deciding whether and by how much to change the document organization from the original.

NOTE 3 See also 6.4.4.2 (in particular NOTE 2), 17.4 and 17.5.

8 Guidelines on terminology

8.1 General

The careful and precise use of terminology is important for any kind of specification, particularly a standard specification, but it is especially important for language-independent service specifications.

8.2 Guideline: The need for rigour

The terminology used in a language-independent service specification should be defined rigorously, even where it is believed that a term is generally well understood. Different usages of the same terminology (and unspoken assumptions that possibly do not hold) commonly encountered in language communities should be pointed out.

NOTE 1 Languages vary greatly in terminology, using same or similar words for very different things, or for slightly different things, and different words for same or slightly different things, which make it critically important to be very precise in their use.

NOTE 2 Slight variations of meaning can cause more trouble than large ones, simply because they are easy to overlook, so rigour, in the sense of completeness as well as accuracy, is especially important where these can occur.

NOTE 3 The use of a formal specification language can help to eradicate insufficiently rigorous definition of terminology, even if not used normatively, since the formal definitions can be used to check the interpretation of natural language terms.

NOTE 4 Annex B contains a glossary of language-independent terms that can be used as a starting point.

8.3 Guideline: The need for consistency

All normative terms and phrases, once defined rigorously in accordance with 8.2, should be used consistently throughout the language-independent service specification, with the precise meaning as defined.

NOTE See NOTE 3 of 8.2 also.

8.4 Guideline: Use of undefined terms

All uses of undefined terms in the language-independent service specification should be carefully checked to ensure that they cannot lead to normative ambiguity.

NOTE 1 In any specification, undefined terms whose meaning is assumed to be understood, need to be used at some point, but providing definitions, either directly or by reference, is not necessary when the resulting lack of rigour is not relevant to the specification.

NOTE 2 Again, the use of a formal specification language (NOTE 3 of 8.2) can be helpful.

8.5 Guideline: Use of ISO 2382

As far as possible, the language-independent service specification should use the terminology given in the appropriate parts of ISO 2382, taking into account common practice in the community providing and using the service, and in the various language communities concerned, and also the possible costs of transfer to new terminology. ISO 2382 terminology should nevertheless be used in preference to terminology specific to one particular implementation or binding language. Additional terms not covered by ISO 2382 should be defined in a specific section of the standard.

8.6 Guideline: Use of definition by reference

Though definition of terms can be by normative reference rather than detailed exposition, the language-independent service specification should in general include the text of the referenced definitions, at least for information, and it should be made clear that, since usages vary, users of the language-independent service specification should not assume (without having explicitly checked) that their habitual use of a term is identical to that given.

NOTE This applies to all referenced terms, including those in ISO 2382.

8.7 Guideline: Terminology used in bindings

Language bindings should be explicitly required to address and explain fully any differences of terminology between the language and the language-independent service specification.

9 Guidelines on use of formal specification languages

9.1 Guideline: Use of a formal specification language

Serious consideration should be given to the use of a formal specification language to define the service semantics.

NOTE 1 The use of a formal specification language reduces the risk of divergence and incompatibility between bindings and implementations in different languages, arising from differing connotations and underlying assumptions in the use of natural-language terms.

NOTE 2 A formal specification language often makes it possible to carry out automatic checks for errors, omissions and inconsistencies in definitions, which can be difficult to spot with natural-language methods (e.g. an inconsistency between two requirements that are widely separated in the document text). It remains a human responsibility to ensure that the resulting complete, consistent and precise definition is of the semantics intended.

NOTE 3 It can be worthwhile producing complete or partial formal semantics even if the final specification is completely non-formal, focusing discussion, helping to avoid misunderstandings among the specification team, improving the quality of the final specification, and possibly saving time.

NOTE 4 Great care is needed in using more than one formal language for specifying semantics, especially for parts of the specification that are not well separated. This applies equally to the use of formal languages mentioned here and the IDL as defined in LID and LIPC (see Clauses 12 and 13).

9.2 Checklist of formal specification languages

9.2.1 General

Services vary so greatly that it would serve no useful purpose for this document to recommend the use of one formal specification language or even one style of formal specification language. However, to assist those using this document, there follows a list of formal specification languages which have been made the subject of International Standards, together with a brief indication of their style and of their range of applicability.

9.2.2 Estelle

Estelle (ISO 9074) is a standardized formal specification language. The Estelle language is based on a stripped-down version of Pascal that has been extended with a notion of modules and communication between those modules. Estelle semantics are based on extended finite state automata. A system is modelled by a set of module instances that communicate messages asynchronously over given channels.

Modules are defined by a body and a header. The body defines, in a Pascal-like way, the behaviour of the module, and the header defines the external interface. Channels are defined by two roles (one for each end of the channel), where each role definition defines the messages that can be sent.

Although Estelle was originally designed to specify communication services and protocols, it can be used to specify other systems. The module definition appears to map well onto the notion of specifying internal services and external interface separately. However, the Pascal-like nature of the module body language is liable to exert strong bias on any implementations developed from the specification, because of the detailed nature of those specifications.

9.2.3 Lotos

Lotos (ISO 8807) is a standardized formal specification language. The Lotos language is based on a combination of the ACT-ONE specification language with a process algebra based on the concepts of CCS and CSP. Lotos semantics are based either on abstract datatype specifications or on process algebras, depending on which part of the language was employed. A system is modelled as a collection of processes that communicate potentially complex data objects synchronously over given ports.

Data objects can be defined in the ACT-ONE part of Lotos. Their definition consists of an interface defining the syntax of the operations used to manipulate the data object, and a set of rules that define how the operations interact with each other. Processes are defined by a header that names the ports through which the process can communicate with other processes, and a behaviour expression that defines the allowable behaviours (as seen through interactions on the ports) of the process.

Lotos was originally designed to specify communication services and protocols, but has been used to specify other types of system. That Lotos comprises both an abstract datatype component and a process algebra component means that it can be used in any circumstance where either would be appropriate.

9.2.4 VDM-SL

The Vienna Development Method Specification Language (VDM-SL, ISO/IEC 13817-1) is based on a rich set of basic and compound types with syntax that allows the definition of functions, global state, and operations that can modify the state. VDM-SL semantics are based on denotational style lambda calculus. A system is modelled as a collection of global state variables and the operations used to modify the state and other functions.

Global state variables model the state of the system and are defined by constructions of the basic or compound types of VDM-SL. Invariants can be added to the types of the state variables providing appropriate constraints. Operations and functions are defined by a header that defines which state variables will be accessed, and a pre-condition and a post-condition that define the behaviour of the operation.

In addition to specification by pre- and post-conditions, VDM-SL has programming-language–like constructs to describe iteration and assignment, and has a well-defined refinement process that can be used to refine datatypes or function or operation definitions.

VDM-SL was first developed for the specification of compiler semantics, but has been used for many sequential (and some concurrent) system specifications.

9.2.5 Z

Z is a specification language currently undergoing international standardization as ISO/IEC 13568. Z is based on a typed set theory where the types of the values are well defined and can be relations. Z semantics are provided in a denotational style using a specially developed relational algebra. A system is modelled in Z as a collection of schemas that define desirable properties that the system needs to exhibit.

Schema definitions have two parts, either of which can be optional. The first part is declarative and introduces the variables for which a relationship is to be defined. The second part is the predicate part, and defines the relationship that needs to hold between the variables defined in the scope of the

schema. Schemas can be used to define complex data objects, or types with or without invariants, or to define operations or functions. Z provides powerful mechanisms for composing schemas to construct larger specifications from smaller components.

Z has been used in a variety of software and system specifications such as transaction processing systems or heart pacemakers. Z has also been used to specify the semantics of some of the POSIX standards.

9.2.6 Extended BNF

A syntactic metalanguage is a notation for defining the syntax of a language by a number of rules. The concepts are well known, but many slightly different notations are in use.

A syntactic metalanguage can also be sensibly used whenever a clear formal description and definition is required, e.g. the format for references in papers submitted to a journal, or the instructions for performing a complicated task.

Extended BNF (ISO/IEC 14977) is general purpose, and its adoption saves time by avoiding the need to choose one of several suggested notations, which then needs to be amended to overcome their limitations.

9.3 Guideline: Using formal specifications from the outset

Once the decision has been made to use a formal specification language, and the particular language has been selected, the chosen specification language should be used from the outset, all participants in the project being required to submit proposals and drafts using formal rather than informal semantics.

NOTE Formal methods, especially for defining semantics, are at present not widely known or used among IT practitioners, despite their known advantages. Experience has shown that using the agreed formal specification language from the start is much easier than trying to introduce one later on when participants have become used to discussing issues relating to the project in informal terms. Early progress can be slower than it has been, through initial unfamiliarity with the formal language, but any lost time is usually recovered in due course since ambiguities and errors are less likely to occur and are easier to detect.

9.4 Guideline: Use of operational semantics

Care should be taken if using operational semantics for formal definition of a language-independent service specification, to avoid appearing to provide an implementation specification.

NOTE 1 In operational semantics, the definition of the semantics is made in terms of the operation of an "abstract machine" which implements that semantics. There is a consequent danger of providing a detailed implementation model, especially with a formal specification language capable of operational semantics with translation into an executable (even if inefficient) actual implementation. Other formulations, such as axiomatic or denotational semantics, do not rely on such an implementation model.

NOTE 2 If the specification is derived from an original implementation based on a particular programming language, there is the added danger of the "abstract machine" reflecting the character and inbuilt assumptions of that language.

NOTE 3 Depending on the nature of the service being defined, it can be possible to provide an operational semantics at a sufficiently high level of abstraction to avoid these dangers.

NOTE 4 There can be cases where external constraints or other considerations mean that the use of operational semantics, at a level implying an implementation model, is nevertheless indicated. In such cases, the assumptions of the model are spelled out, and guidance is given on alternative forms of implementation where those assumptions are invalid or inappropriate. It is especially important that such a specification be reviewed by experts familiar with a variety of language environments and implementation strategies, to minimize the risk of inbuilt bias.

10 Guidelines on interoperability

10.1 General

The term interoperability is used in many contexts, sometimes in a very vague and general way, and is sometimes confused with the related but distinct concept of portability. This introduction is intended to clarify the concept of interoperability in the context of service specifications, as a preliminary to the associated guidelines for language-independent service specifications.

10.1.1 Interoperability with what?

Interoperability issues arise when a service is required to interoperate with other services in the course of providing its own services to an external user. The other services concerned can be other instantiations of the same service, or be different services, or both.

If interoperability is with other instantiations of the same service, that becomes one of the design requirements of the language-independent service specification, and while this can add to the difficulty of defining the specification, it is a relatively straightforward situation to deal with.

NOTE Questions of interoperability can be very complicated for a distributed system, which can allow different implementations on the same or different types of computers (supporting interfaces in the same or different languages) interoperating at various levels, e.g. exchanging data, sharing a database, or invoking each other. In such a case, it is important to be clear and agreed on just what forms of interoperability are required.

If interoperability is with different services, then the extent of the difficulty of defining the specification depends upon whether these are being specified at the same time, or pre-existing services that are already specified.

In general, the effect of interoperability requirements is to add constraints to the specification, which is why the strategic guideline 6.2.2 recommends that they be dealt with first, along with any concurrency requirements (see Clause 11), when developing a specification. When constraints arise in connection with other instantiations of the same service, or different services being specified at the same time, though they exist, they are not necessarily especially troublesome. Constraints are likely to be much more severe when interoperability is required with an existing, already specified service, since the possibility to alter the specification of that service to make interoperability easier is unlikely. Even if the specification of that service is being revised, the scope for adjustment to ease interoperability can be limited, or even non-existent.

In the worst cases, the constraints can create pressure to compromise the aims of the language-independent service specification, for example if another service makes representational assumptions about exchange values, or makes other implementation assumptions which have an impact on interoperation. They can even create pressure to compromise the aim of language independence. This needs handling with great care, and preservation of language independence can require some ingenuity. This subclause provides some guidelines on dealing with such situations.

Severe constraints can also occur if there is a need for synchronicity, or at least some guaranteed response time. If the service being specified needs to meet such a requirement for an external user, the need to interact with some other service can create complications. Alternatively, if the other service demands synchronicity or other forms of time constraint, this can potentially affect the ability of the service to respond to its own external users. In general, how services can handle time constraints is outside the scope of this document. It should be noted however that languages vary very greatly in their ability to handle synchronicity and time constraints, which can place severe difficulties in the way of defining the service itself, or language bindings, in a truly language-independent way.

Though the nature of the other services is the most important factor affecting interoperability, two other factors can be important: the nature of the interoperation, and how it is invoked.

10.1.2 The nature of the interoperation

Interoperation can be master–slave, slave–master, or peer–peer. In a master–slave relationship, the language-independent service invokes the other service, but needs to do no more than state its requirements, expecting the other service to deliver what is required. In a slave–master relationship, the language-independent service is invoked by the other service, and needs to deliver what that other service requires. In a peer–peer relationship, the services cooperate to deliver what the external user requires.

The master–slave situation should cause relatively little difficulty, the first because it is a matter only of invoking the other service and being able to handle its various responses. The slave–master situation can be treated as if the other, "master", service is another external user, with its own interface, the main problems arising if the other service is pre-defined and imposes requirements involving severe constraints. The peer–peer situation can also involve severe constraints if the other service is pre-defined, and can pose tricky design problems.

10.1.3 How interoperation is invoked

Interoperation can be invoked by the external user (e.g. by exercising an option or selecting a parameter) or take place in the background, solely within the service, so that the external user is not directly concerned with it (and can even not be aware of it).

Neither of these situations should present too many problems, provided that it is clearly understood which form of interoperation is involved, and it is handled in the appropriate way. Where interoperation is invoked by the external user, this can be treated as part of the interface like any other feature of the service. Where interoperation takes place solely in the background, depending on the nature of the interoperating service it can be appropriate to define an explicit further interface, separate from the interface to the external user, to handle the interactions. Difficulties are likely to occur only when these two situations are confused, or not kept clearly separate.

10.2 Guidelines on interoperability with other instantiations of the same service

Where interoperability is required with other instantiations of the same service, it is probable that the relationship will be peer–peer. The guidelines that follow are therefore devised on that assumption. Circumstances can be envisaged in which this is not the case. In this case, the guidelines in 10.3 for the master–slave relationship need to be appropriately adapted.

10.2.1 Guideline: Identifying features affecting interoperability

All aspects of the service that affect interoperability with other instantiations of the service should be identified, and the specification should ensure that these are clearly distinguished from other aspects.

10.2.2 Guideline: Precise definition and rigorous conformity requirements

All aspects of the service that affect interoperability with other instantiations of the service should be precisely defined, and conformity requirements should be made rigorous enough to ensure that the ability to interoperate is always maintained, whatever combination of options and implementation-defined choices are used by this and the other instantiations.

NOTE 1 Experience shows that interoperability between standard-conforming implementations is often prevented because conformity rules are not strong enough to ensure it.

NOTE 2 Over-specification of the requirements – e.g. making rigid representational requirements – simply to make absolutely sure that interoperability is always possible, is not necessary. It is sufficient to keep the scope of the specification and its level of abstraction clearly in mind, and that strict adherence to the conformity rules is necessary.

NOTE 3 The use of formal definitions to eliminate ambiguity is particularly useful in relation to interoperability requirements.

10.2.3 Guideline: Importance of exchange values

In specifying interoperability requirements, particular attention should be paid to the datatypes used for exchange values, and to the exact ranges of validity of data values needed for interaction.

NOTE 1 The LID standard includes facilities for specifying precise ranges of values in a language-independent way, so representational requirements are not needed unless the service itself is at a representational level of abstraction.

NOTE 2 It is possible, without breaching this guideline, to allow values outside the specified range of validity for interaction to be used in situations where that value is actually not acted upon, but only used "for completeness sake". This, however, is a risky practice, and is probably best avoided unless a strong rationale exists to permit it.

10.3 Guidelines on interoperability with other services

10.3.1 General

Where interoperability is required with other services, it is possible that the relationship will be peer-peer, but more likely that it will be master–slave or slave–master. A peer–peer relationship is most likely to occur when specifications for the two (or more) services are being developed together. Master–slave or slave–master relationships can occur with specifications being developed together, but it is more likely that the "slave" service will be defined first and the "master" service is specified later to make use of it. However it can happen that a "master" service is defined and then (for example at a revision which extends the service) requires a "slave" service which it can invoke.

The guidelines that follow therefore cover the two main cases, of services (whatever the relationship) being developed together, and of a service being developed to interoperate (whatever the relationship) with some pre-defined service.

NOTE Where more than two services are involved, it is possible that there is a "mixed" situation where two or more services are being developed together to interoperate with one or more services already defined. Those faced with that task are encouraged to apply the guidelines below as much as possible, to meet the needs of the particular case.

10.3.2 Guideline: Interoperability with other services being defined at the same time

Where interoperability is required with other services which are also being defined at the same time, the services should be regarded as a single "super-service" in respect of the interoperability aspects, and the guidelines in 10.2 should then be applied to that "super-service".

NOTE 1 Care is needed, with duplication of definitions if necessary, to ensure consistency across the different components of the "super-service", taking into account the possibility that these different component services can be implemented using different languages.

NOTE 2 Treating two or more services as a single "super-service" for the purposes of specification is simpler to arrange if the same group is responsible for all of them. A high level of liaison, cooperation, and mutual trust is called for if more than one group is involved.

10.3.3 Guideline: Interoperability with a pre-defined service

Where interoperability is required with another service which is already defined, all aspects of the pre-defined service that affect interoperability with the service now being defined should be identified, particular note being made of those which impose pre-defined requirements. If these requirements are specified in a language-dependent way, they should be re-specified in language-independent form. An interface should then be defined which allows the language-independent service to appear to the pre-defined service as if it were a service in the same language. All definitions should be made precise, and conformity rules should be made rigorous, especially for this interface, particular attention being paid to exchange values.

11 Guidelines on concurrency issues

11.1 General

Concurrency issues, i.e. issues concerning whether actions should take place serially or in parallel, can arise within the specification of the service, in the way the interface with users of the service operates, or in what the service, through its interface, requires from the user.

In general, the processes of a service can be divided into three groups: essentially serial, optionally concurrent, and essentially concurrent. A process that is essentially serial needs to have its parts carried out in a specified sequence if it is to function correctly. A process that is essentially concurrent can have its parts carried out in parallel if it is to function correctly (though the parallelism can often be simulated by a serial process achieving the same end result, if there are no external constraints to make that impossible). A process that is optionally concurrent functions correctly, whether or not its parts are carried out in parallel (since the parts are not interdependent in any way that affects the process).

NOTE 1 The logical fourth possibility, optionally serial, is identical to optionally concurrent.

NOTE 2 The replacement of the values a,b,c of the sequence (a,b,c) by the values x,y,z, giving (x,y,z), is an example of an optionally concurrent action. However, their replacement by the values c,a,b to give (c,a,b) is an example of an essentially concurrent action, since changing the values one at a time does not, in general, produce the required result. The essentially concurrent action can, however, be simulated serially by making copies of values that would otherwise be lost, and using them when needed.

In this clause, the term “concurrency requirement” is used to denote any requirement relating to any of these three possibilities; in particular it can mean either or both of a requirement to perform tasks in a given sequence or of a requirement to perform tasks in parallel.

In general, the effect of concurrency requirements is to add constraints to the specification, which is why the strategic guideline 6.2.2 recommends that they be dealt with first, along with any interoperability requirements (Clause 10), when developing a specification.

11.2 Guidelines on concurrency within the service specification

11.2.1 Guideline: Avoidance of unnecessary concurrency requirements

A language-independent service specification should avoid concurrency requirements other than any which are absolutely necessary to provide the service; in particular it should not require serial processing when parallel processing can achieve the required, nor should it require actual parallel processing (as opposed to simulated parallel processing) unless this is demanded by external constraints or the nature of the service. Every apparently necessary concurrency requirement should be examined closely, to see if there is a way of avoiding it, so that the number that eventually remains in the specification is kept to a minimum.

NOTE 1 Unnecessary concurrency requirements, whether for serial processing or parallel processing, is an example of over-specification arising from the inclusion of implementation assumptions.

NOTE 2 Requirements to handle a number of service users simultaneously (see 11.3) can entail some degree of parallelism, but this guideline still applies.

11.3 Guidelines on concurrency of interaction with service users

11.3.1 General

Depending on the nature of a service, it can be necessary to deal with one user at a time. In that case, it can be that the service is not able to accept a request from another user until the current one has been dealt with, or it can be able to support a queuing system, where incoming calls on the service are

accepted as they arise, but are still dealt with one at a time. Alternatively, it can be able to handle a number of service users simultaneously.

11.3.2 Guideline: Handling of concurrent service requests

A language-independent service specification should explicitly state whether an implementation needs to be capable of handling concurrent service requests and, if so, whether the services need to be provided concurrently, or can be queued and the service provided to the users in the queue one at a time.

11.3.3 Guideline: Number of concurrent service requests handled

Where handling of concurrent service requests is required, either through simultaneous provision or by a queuing system, the language-independent service specification should state the minimum number of such requests that it needs to be possible to handle simultaneously, and require the maximum number that can be handled to be documented.

NOTE It is possible that a service is able to handle a certain number of users simultaneously, but also to maintain a queue of those unable to be dealt with immediately. This guideline then applies, separately, both to the simultaneous provision and to the queue.

11.3.4 Guideline: Order of processing of service requests

A language-independent service specification should explicitly state whether an implementation needs to handle service requests in order of arrival, or can prioritize requests, or needs to prioritize requests.

11.3.5 Guideline: Criteria for prioritizing service requests

Where a service needs to or can prioritize requests, the language-independent service specification should explicitly define the criteria which need to or can govern prioritizing decisions, or at a minimum specify constraints which need to be met.

NOTE An example of a constraint is: whatever other criteria apply, no request is made to wait for more than a specified period of time.

11.4 Guidelines on concurrency requirements on bindings

11.4.1 General

In a language-independent service specification, requirements on users are expressed as requirements on language bindings, through the service interface. This subclause provides guidelines for handling concurrency issues that can arise in the specification of such requirements.

11.4.2 Guideline: Avoidance of concurrency requirements

A language-independent interface of a service specification should explicitly be neutral with respect to concurrency, i.e. it should place no requirements on whether an implementation of a language binding uses a serial or a parallel approach or any combination of the two.

NOTE 1 Languages vary greatly in their capability for handling parallelism. Requiring concurrency of a binding can create severe problems for implementing it efficiently, or even at all, or can force implementors to adopt solutions which do not conform to the language standard.

NOTE 2 Some languages are very well equipped to handle parallelism. Requiring a serial form of implementation can place unnecessary constraints on implementors using that language; a far more efficient and natural form of binding can be possible using language features supporting parallelism, and is not to be precluded.

NOTE 3 This guideline does not imply that a particular language binding should not impose concurrency requirements on implementations of that binding; the semantics of certain language features can mean that, for an implementation to be correct, certain concurrency requirements need to be met.

11.4.3 Guideline: Specification of unavoidable concurrency requirements

Where a language-independent interface of a service specification cannot be neutral with respect to concurrency, e.g. through external constraints, the unavoidable concurrency requirements should be specified fully, but in as general a way as possible so as to place the fewest possible constraints on bindings. In particular, no explicit or implicit assumptions should be made about how a language can support parallelism.

12 Guidelines on the selection and specification of datatypes

12.1 General

Probably without exception, all service specifications make use of the concept of data, and any specification needs to define the data to which it applies. The clearest and most common way of doing this is by means of datatypes: a data value which belongs to the relevant defined datatype is covered by the specification, while a data value which does not belong to the relevant defined datatype is not so covered (and in general an attempt to use such an incorrect value constitutes an error).

This clause provides guidelines for the selection and use of datatypes in language-independent service specifications.

12.2 Guideline: Use of ISO/IEC 11404 General-Purpose Datatypes (GPD)

The datatypes used in the language-independent service specification should be selected from those defined in ISO/IEC 11404 either by direct adoption, or using the methods it defines for generating further datatypes from those directly provided in the standard.

NOTE LID provides a wide variety of primitive datatypes suitable for direct adoption, and a variety of methods for generating further datatypes from these. These methods include various forms of aggregation (including Set, List, Array, Record, and Table datatypes) and of other forms of derivation (including "new" for clone datatypes which have the same set of data values but are logically distinct from the original, Choice to merge separate datatypes into a single one, Pointer for indirect referencing, and various forms of subsetting – see 12.3).

12.3 Guideline: Specification of datatype parameter values

For each selected datatype for which LID defines parameters, the language-independent service specification should specify all of the parameter values, either directly, or by placing a requirement to do so upon a conforming implementation of the specification.

NOTE 1 For most datatypes, the set of data values is either potentially infinite or, though finite, of arbitrary size. (For example there are by definition an infinite set of values of datatype Integer, but a logically finite though arbitrarily large set of values of an Enumerated datatype.) LID defines parameters for such types, to allow subtypes to be defined, including subtypes of known finite size. The language-independent service specification needs either to select the parameter values (e.g. specify what range of data values is to be supported) or require a conforming implementation to do so (in which case it is desirable to specify constraints, e.g. "at least..." and/or "at most..." as appropriate).

NOTE 2 LID allows arbitrary sets of data values to be constructed, as well as contiguous ranges (where that concept has meaning), by explicit inclusion or exclusion of specified values. (An example is Range –10:+10 of Integer excluding 0 but extended to include values –20 and +30.) While allowing for such "unusual" values is generally best avoided, because of the implementation overhead, where they are critical to the application this can be a suitable means for the language-independent specification to highlight their presence in a rigorous way.

NOTE 3 Aggregate datatypes in LID have parameters governing the structure and size of the aggregate, e.g. number of dimensions and index ranges for Array datatypes, number and datatypes of fields for Record datatypes, length of CharacterString datatypes, etc.

12.4 Guideline: Treatment of values outside the set defined for the datatype

The language-independent service specification should specify how a conforming implementation is required to handle data values encountered which are outside the defined set of values for the relevant datatype.

NOTE 1 The way disallowed values are handled needs careful thought, and would be precisely and unambiguously defined in the language-independent service specification, since much depends on the consequences. In some contexts, it can be sufficient simply to ignore such values, as if they were not there. More usually, at least a warning is needed, and commonly some exception handling mechanism need to be invoked. In critical cases, the presence of such a value indicates a breakdown, and a need to abort the service.

NOTE 2 Where the language-independent service specification takes a "permissive" attitude to ranges of supported values of given datatypes, and allows the implementation to decide, there is a danger that users who are familiar with an implementation allowing a wide range of values can encounter this problem when moving to a more limited one. (This can become critical when interworking between implementations is likely.) Hence, this consequence of a "permissive" approach when defining the specification is always to be borne in mind, and the exception handling designed accordingly.

12.5 Guideline: Specification of operations on data values

For each datatype, the language-independent service specification should specify the operations on its data values required to be supported, and, where relevant, whether and under what conditions further operations are permitted in a conforming implementation of the specification.

NOTE 1 LID defines "characterizing operations" for its datatypes, which are not normative and also not necessarily exhaustive (they are included to aid identification of the most suitable match for language binding purposes). Hence, a complete language-independent service specification lists which operations, whether "characterizing" in LID or not, are required for the application.

NOTE 2 In general, a complete definition for the operation is needed, since details can vary from language to language. For example, it is not enough to say that addition is needed for a Range of Integer datatype; it is also necessary to specify things such as what happens if the result of an addition is "out of range". (Note that LID does include a Modulo datatype, if that is the result required.)

12.6 Guideline: Recommended basic set of datatypes

The datatypes used in the language-independent service specification should, as far as practicable, be selected from the following basic set, which are generally supported directly, or able to be simulated without major binding problems, by a wide variety of languages: the primitive datatypes Boolean, Bit, Integer, Character, and the aggregates Array, Record and CharacterString.

12.7 Guideline: Specification of arithmetic datatypes

With respect to arithmetic datatypes, the language-independent service specification should take into account the provisions of the ISO/IEC 10967 series, in particular ISO/IEC 10967-1.

NOTE 1 Problems can occur with arithmetic datatypes, especially because of the approximate nature of values and operations on values of datatype Real (in LID) and its counterpart in actual programming languages. ISO/IEC 10967-1 gives precise specifications for these arithmetic operations, to predictable accuracy, and its study will help determine how important the accuracy of arithmetic is to the application.

NOTE 2 For many applications the correct functioning of the service does not depend on, or is insensitive to, the detailed behaviour of Real values and operations. In such cases, it suffices to rely on the native arithmetic of the host language and implementation environment. However, even in this case a statement of the arithmetic requirements, however modest, would be included, since a future revision of the specification can become more demanding, and mention of the arithmetic requirements is a safeguard against these being overlooked or unwisely taken for granted.

NOTE 3 For other applications the correct functioning of the service is dependent on arithmetic being carried out to given accuracy requirements. Though it can be believed that all likely host languages and implementation environments meet those requirements, by including them here there can be no doubt over what the requirements are.

NOTE 4 For some applications the correct functioning of the service is critically dependent on arithmetic being carried out to high accuracy. In such cases, it is strongly recommended that the requirements be rigorously specified, preferably using the ISO/IEC 10967 series by direct citation or, failing that, by using the same techniques, and that meeting those requirements be made formally mandatory for conformity to the specification.

12.8 Guideline: Approach to language bindings of datatypes

The language-independent service specification should provide clear guidance on the approach that a language-independent service language binding should take to binding the various datatypes required. Particular care should be taken in the case of complicated abstract datatypes which many languages need to represent through simpler and less abstract datatypes, and, in some cases, it is possible that mandatory requirements need to be included to ensure that the integrity of the language-independent service is protected.

NOTE 1 It is desirable to allow maximum flexibility for the language-independent service to be implemented efficiently and in a way which fits in well with the host language or environment. Hence, requirements on language bindings are best kept to the minimum needed to protect the integrity of the language-independent service. On the other hand, implementors are helped if requirements are explicit and the limitations on flexibility are made clear, rather than a matter for interpretation.

NOTE 2 In many cases, it can be that the use of abstract datatypes are helpful to implementors and those defining language bindings, in that it increases flexibility to bind to the most appropriate datatype available.

12.9 Guideline: Avoidance of representational definitions

When defining the datatypes used in the language-independent service specification and the operations to be supported, any explicit or implicit dependence on or assumptions about the form of representation of the datatype values should be avoided as far as possible, the definitions being in terms of abstract properties only. When dependence on some form of representation is unavoidable (e.g. because the language-independent service entails interworking with some other service which does require a particular representation) then the representation requirements should be made explicit, and kept to the minimum necessary for the language-independent service to be performed correctly. The language-independent service specification should also address the issues relating to conversions from and to other forms of representation.

NOTE 1 In this subclause, the term “representational” includes indirect forms as well as direct forms. An example of an indirect form is assuming that a value of datatype Complex, approximating to a value in the mathematical complex domain, is represented as a pair of values of datatype Real, approximating to values in the mathematical real domain, which are approximations to the (mathematical) real and imaginary parts of the corresponding (mathematical) complex number.

NOTE 2 In the case of datatype Real, for most applications it is usually safe to make the assumption that values are represented in floating point form. Use of ISO/IEC 10967-1 entails that assumption, without any further assumptions e.g. about the radix used, or machine representation for storage or transmission of values. For some applications, however, it can be desirable for the language-independent service specification to address issues relating to the use instead of some other form, such as fixed point (datatype Scaled in LID).

13 Guidelines on specification of procedure calls

13.1 General

This clause provides guidelines on how to specify procedure calls in language-independent service specifications. Many (perhaps most) service specifications find it convenient to define certain actions or functions of the service in terms of procedure calls; indeed, some services can conveniently be

defined entirely in terms of procedure calls and the data on which they act. In particular, any required operations on data values (see 12.5), which not all actual languages necessarily provide, can well be best defined in that way.

13.2 Guideline: Avoidance of unnecessary operational assumptions or detail

The language-independent service specification should define each procedure call in terms of the overall effect achieved by the call in relation to the service, not in terms of how that overall effect is to be achieved.

NOTE 1 The concept of "procedure call", though a very general one and by no means confined to so-called "procedural languages", is often interpreted in terms of specifying things procedurally, i.e. how an effect is to be achieved. To make the service specification truly language independent, it is necessary to bear in mind that many languages are "non-procedural" (i.e. the underlying procedural aspects do not appear at the level of the source code, and hence are not directly under the control of the programmer). It is worth noting also that even the procedural languages can vary in the way that they achieve certain effects, and if the specification defines the "how" as well as the "what" of a procedure call, the binding for languages who achieve the same effect differently are not the most effective or efficient one.

NOTE 2 This guideline implies that it is undesirable that the language-independent service specification requires or expects that a language binding necessarily implements a procedure call in the language-independent service specification as a procedure call in the language.

NOTE 3 This guideline implies that it is undesirable that the language-independent service specification specifies higher-level procedures in terms of calls of lower-level procedures, unless this is unavoidable because of external constraints (i.e. requirements imposed by the environment in which the service operates).

NOTE 4 This guideline encapsulates the concept sometimes referred to as "the right level of abstraction": i.e. that the right level of abstraction is not the operational level, but a higher and more abstract level which leaves out the operational detail.

NOTE 5 This guideline implies that it is undesirable to specify a language-independent service in terms of operational semantics, since this can easily imply an implementation model for the service, including its constituent procedures (see Clause 9).

13.3 Guideline: Use of ISO/IEC 13886 procedure calling model

The language-independent service specification should, for its procedure calling model, use ISO/IEC 13886.

NOTE The advantage of using LIPC as the model is that any LID datatype can be used as a parameter (or for the returned value) of any LIPC procedure, which greatly simplifies the language-independent service specification and reduces the chance of clashes with other related language-independent service specifications. Use of LIPC and the very wide range of datatypes available for parameters also maximises freedom to implement the service in the way that best suits the language used, using the relevant language bindings to LIPC and LID.

13.4 Guidelines on the use of ISO/IEC 13886

13.4.1 General

ISO/IEC 13886 defines a language-independent model of procedure calling of sufficient abstraction to allow the procedure calling facilities of many languages to communicate by mapping them to and from the LIPC facilities, the mapping being defined by the bindings of the languages concerned to LIPC. The model allows for various modes of parameter passing.

An LIPC parameter can be of any datatype definable via ISO/IEC 11404. No distinction is made between "function" procedures that return a value through the procedure name (and hence can be called to provide values to expressions directly), and "subroutine" procedures that do not return a value in such a way (though they can return results through setting values of suitable parameters). The language syntax used to invoke a procedure is a matter for the language concerned, and of no relevance to LIPC. If the language allows for "function" procedures, the language binding maps the return through the

procedure name of the evaluated value of the function into an additional parameter of the corresponding LIPC invocation.

NOTE 1 Hence, the LIPC equivalent of the square root function sqrt(x), provided in many languages, would be of the form sqrt(x,y), y being set to the square root of x.

LIPC acts as a bridge between the procedure calling facilities of different languages. A language processor offering LIPC server facilities for a procedure (i.e. on the service provider side, in the present context), maps the LIPC procedure call definition, including the number and datatypes of formal parameters, into the form of the corresponding procedure call in the language on its side, using the LIPC binding for that language.

NOTE 2 The semantics of the procedure once called (sometimes termed the procedure body as distinct from the procedure head) can be defined in various ways, e.g. in language-independent form, or by program source code in the language on the service provider side.

A language processor offering LIPC client facilities (i.e. on the service user side) can then invoke that procedure, in terms of the language on that side. The actual parameters are converted by the LIPC client facilities from the local datatypes to the LID datatypes required for the formal parameters, using the LID binding for the language; this process is termed marshalling.

Transmission of the procedure invocation and parameters to the service provider side is a system implementation matter outside the scope of LIPC. Once received by the service provider side, the LIPC server facilities unmarshal the marshalled actual parameters from the LID datatypes into the local datatypes used by service provider mapping of the LIPC procedure call definition. Return of results is performed by a reverse process of marshalling on the service provider side and unmarshalling on the service client side.

LIPC specifies four abstract modes of parameter passing (see 13.4.3), and explains how the common forms of parameter passing found in languages can be expressed with them. As a guide for those defining and implementing LIPC services, an abstract model of the execution of a procedure call is provided. There is no requirement to implement this execution model itself, which is provided solely to aid understanding and reduce the risk of incompatibilities through differences in assumptions being made by people with different language backgrounds.

13.4.2 Guideline: Selection of datatypes of parameters

Care should be taken in the selection of datatypes of parameters, to avoid breaking down complicated datatypes used for definitional purposes into simpler ones for specifying operations, simply because it is known that most languages are unable to pass as a parameter a value of a complicated datatype. The definition of the procedure should be made at the highest level of abstraction possible (see 13.2) to enable bindings of the language-independent service specification to exploit language features to their best effect.

NOTE This does not preclude the language-independent service specification from including, e.g. as an informative annex, advice to implementors and those defining language bindings of the language-independent service specification on possible methods of breaking down complicated datatypes for parameter passing purposes.

13.4.3 Guideline: Selection of parameter passing modes

As far as possible, the language-independent service specification should use the LIPC parameter passing mode “call by value sent on initiation” for each parameter.

NOTE 1 The LIPC parameter passing mode “call by value sent on initiation” is what is commonly called simply “call by value”, or “in”.

NOTE 2 Experience shows that “call by value” is the parameter passing mode which causes least confusion to users, and is least prone to unwanted side effects.

NOTE 3 Special rules, or at least guidance, might be considered to be included for bindings of the language-independent service specification to languages which do not specify "call by value", or "in", as a required parameter passing mode, unless these are already covered adequately for the language-independent service specification in an LIPC binding standard.

NOTE 4 LIPC permits the use of "call by reference" in this mode, by passing the value of a "pointer" datatype. The advantage of this is that it makes precise the concept of "call by reference", by making it clear what is allowed to change and what is not. This is the recommended method for returning values to the calling environment. Special rules, or at least guidance, might be considered to be included for bindings of the language-independent service specification to languages which are less precise in this respect, unless these are already covered adequately for the language-independent service specification in an LIPC binding standard.

NOTE 5 An alternative, for the return of results, to the use of a pointer datatype called "by value", is use of another LIPC parameter passing mode, "call by value return on termination", i.e. "out". This can be preferred in some cases. In fact, if the specification is at a high enough level of abstraction, it can be possible to use of either method, depending on available language features.

13.4.4 Guideline: Use of bindings to LIPC

The language-independent service specification should require that any implementation of the service or language binding to the language-independent service specification standard conforms in respect of procedure calling to the relevant requirements of the relevant binding to LIPC, where one exists.

NOTE Special rules for implementors are needed to cover cases where a binding to LIPC to the language used does not exist.

13.5 Interfacing via remote procedure calling (RPC)

13.5.1 General

A service for which a language-independent specification is required can be intended for use wholly or primarily across a network. In consequence, any procedure call from the service user is transmitted to the service provider through a communication channel. Thus, the values corresponding to the actual parameters supplied by the service user, and results returned by the service provider, need to be encoded in the appropriate transmission protocol. The need for encoding of values for transmission imposes constraints on the datatypes of the encoded values, and the parameter passing mode used.

However, the service still needs to be capable of interoperating with other (non-remote) services, meaning that it is still appropriate for the language-independent specification of the service to be expressed in the most general terms possible, using LIPC for procedures as recommended earlier. The constraints imposed by the need for service users to call procedures remotely, across a network, are therefore reflected wholly in the language-independent specification of the interface. There is no logical problem in this, because the remote procedure call constraints apply only to the interface.

A specific standard deals with procedure calls operating under such constraints: ISO/IEC 11578. The constraints referred to above are reflected in the facilities this standard provides.

This subclause provides some general guidelines for specifying an interface under remote procedure call constraints, including use of the RPC standard.

13.5.2 Guideline: Avoid limiting the service specification because of constraints on the interface specification

Remote procedure call constraints on the interface should not be carried over into the language-independent service specification.

NOTE 1 The language-independent service specification give the formal definition of the procedures, and the datatypes and passing modes for parameters that are the most appropriate for the service are specified. This allows implementors of the service maximum freedom to exploit the facilities of the language used for implementation.

NOTE 2 For some services the LIPC datatypes and passing modes for parameters equivalent to those in RPC are in fact the most appropriate. In such cases, this problem does not arise.

NOTE 3 The LIPC standard contains an annex summarizing its relationship with RPC, which can be consulted if necessary.

13.5.3 Guideline: Specification of RPC interface

A language-independent interface specification with remote procedure call constraints should define the usual LIPC marshalling and unmarshalling appropriate to the case and then a further stage of marshalling, from LIPC into RPC form, and an additional stage of unmarshalling from RPC into LIPC form before the usual unmarshalling from LIPC to the service provider language form is carried out.

NOTE 1 Where calling can take place non-remotely as well as remotely, the usual LIPC-based interface is needed. Even when calling only takes place remotely, specification of marshalling and unmarshalling in two stages is desirable for documentation purposes: for example, it makes it clear which transformations are caused by moving between language-dependent and language-independent forms, and which are the results of remote procedure call constraints.

NOTE 2 This does not imply that implementations of the interface are required to use a two-stage process. On the contrary, implementations are free to optimize the marshalling and unmarshalling transformations within the limitations of the languages concerned and the remote procedure call constraints.

NOTE 3 Language bindings to LIPC can also provide for the case when remote procedure call constraints are present.

13.5.4 Guideline: Use of subsets

In some cases, a language-independent service specification is best defined in terms of an LIPC-based full service, with a more restricted RPC-based service as a subset. If this is done, it should be ensured that the restricted service is a fully conforming subset of the full service.

NOTE 1 It is better to avoid using the subset approach for speed or simplicity, because of the well-known attendant disadvantages of including levels and options in standards.

NOTE 2 If possible, it is advisable to use a formal specification method to demonstrate the full conformity of the subset.

13.5.5 Guideline: Use of ISO/IEC 11578

For some services where remote procedure call constraints apply, it can be necessary or desirable to make use of features in ISO/IEC 11578 in addition to those with corresponding features of ISO/IEC 13886. In general, any such language-independent interface specification should be reviewed in the light of ISO/IEC 11578, so that the maximum benefit is obtained from use of that International Standard.

NOTE RPC is defined at a more concrete level of abstraction than LIPC, and hence addresses issues outside the scope of the LIPC standard which can still be relevant to the service being specified.

13.6 Guideline: Guidance concerning procedure calling to those defining language bindings to the language-independent service specification

Procedure calling is a well-known concept, and exists in some form in all programming languages, or at least all those likely to wish to bind to a language-independent service specification. It is also a simple concept, at least at the level of provision of functionality. However, because procedure calling interacts with other parts of the language concerned, there are many detailed variations between one language and another, not just at the syntactic level, which is relatively easy to deal with, but in terms of the underlying operational model used. Languages vary, sometimes greatly, in their underlying structure and design assumptions, and this can often be reflected in the procedure calling model.

The difficulty for those defining a language binding to an external service is that the service probably does not have exactly the same procedure calling model. There is obvious danger of mismatches, faults, and unexpected behaviour arising because those defining the service made unspoken assumptions about procedure calling while those defining the binding made different unspoken assumptions. This is especially true when those concerned are particularly expert in one language but not very familiar with others.

Hence the first necessity for those defining a language binding is to examine carefully the underlying assumptions of the procedure calling model of the "target" language, and compare these to the underlying assumptions of that used for the service. It is recommended that this be done in each case by first separating the three concepts of defining the procedure; invoking it; and the delivery of its results back to the point of call, and then doing detailed comparison of each of the three.

However, this document recommends throughout that relating the procedure calling models of the service and the target language by using the common reference point provided by LIPC, which was designed for that purpose. If the language-independent service specification was produced in accordance with the guidelines in this document, it, and the interface to which the binding will relate, uses the LIPC model, at least as a reference if not for the detail of the specification (see 13.3). Even if the service specification is language-dependent, a language-independent interface produced in accordance with the guidelines presents the service using the LIPC model – in other words, it acts as a buffer between the service's internal assumptions, and the outside world in which those assumptions do not necessarily hold.

If the target language already has a defined LIPC binding, in a standard or, failing that, an authoritative document, then those defining the language binding can use that. If not, they need to define one of their own, covering at least enough to meet the needs of binding to the service concerned. LIPC has an annex, titled *How to do an LIPC binding for a language,* which provides guidance for that. Before embarking on this, a search should be made to determine if others in a similar situation have produced partial bindings.

NOTE The danger that different partial LIPC bindings are incompatible shows the importance for a language community of agreeing on a definitive standard binding to LIPC.

For some services, however, an existing LIPC binding is not enough, since it primarily covers invocation and return of results. While this is adequate in many cases, there can be a need for a common understanding of what happens during execution – for example, when questions of interoperability arise (see Clause 10). In such cases, ISO/IEC 13886:1996, Clause 6 should be studied, because that provides an abstract formal model of procedure execution for use in such situations.

14 Guidelines on specification of fault handling

14.1 General

During the invocation or execution of a service, faults can occur. The term "fault" is used here in a broad sense, to mean any occurrence which prevents, delays or changes the way in which the service would normally operate. For a service to be reliable, it is important to foresee the kinds of fault that can arise, and to determine how they should be handled when they do.

Some faults are internal to the service itself (i.e. they occur on the service side of the interface). If a fault can be completely handled on the service side so that, with appropriate fault recovery procedures, the service is unaffected, it is not visible outside, so a language-independent service specification covering only the service interface need not address it. However, if the service is affected, e.g. is stopped, or delayed, or acts in a way other than expected, then the fault, or a manifestation of it, is propagated through the interface, and it can be necessary for the language-independent service specification to address it.

Some faults occur at the interface, e.g. arising from a mismatch between what the service user invokes and what the service provider expects. The language-independent service specification needs to address such faults.

Some faults are specific to the binding. It can be necessary for the language-independent service specification to address such faults, for example, by identifying them, and saying that they are a matter for the binding.

This clause provides guidelines on how to approach all these aspects of a language-independent service specification.

14.2 Guideline: Fault detection requirements

Requirements should be included covering fault detection, reporting and handling, with appropriate conformity clauses. The language-independent service specification should specify a minimum set of faults which a conforming implementation needs to detect (in the absence of any masking faults); minimum level of accuracy and readability of fault reports; whether a fault is fatal or non-fatal; and, for non-fatal faults, the minimum recovery action to be taken.

When considering requirements in this area, it is possible that drafters of language-independent service specifications need to take execution overhead into account, which can be considerable for some services, some implementations, or some languages to which the language-independent service specification is bound. A possible way of dealing with conflicting priorities (e.g. between speed and safety) for differing uses of the service can be to specify that implementation options should be available to allow the level and extent of fault checking to be controlled.

14.3 Checklist of potential faults

The following is a list of typical faults which can arise in the invocation and execution of a service. Drafters of language-independent service specifications should check all of the following for relevance to their service, and the specification produced should address all that are appropriate, plus others specific to the service concerned. This list is not to be considered either as exhaustive or as prescriptive.

In all cases the language-independent service specification should specify whether the fault concerned is fatal or non-fatal. Depending on the nature of the service, it can occur that a particular fault does not constitute a problem (whereas it would in another service) but that users of the service would nevertheless benefit from the availability of a warning message from the implementation.

14.3.1 Invocation faults

a) unknown or misspelt command.

b) duplicate user-defined name.

c) invalid syntax of numerical value (e.g. two decimal points).

d) call for unknown service function or other named facility.

e) wrong number of parameters supplied in call.

f) wrong datatype of parameter supplied in call.

g) symbol supplied not in supported character repertoire.

14.3.2 Execution faults

a) attempt to divide by zero.

b) numeric overflow on arithmetic operation (any numeric datatype, Real, Scaled or Integer).

c) numeric underflow on operation yielding datatype Real value.

NOTE It is possible to specify an implementation option, to permit the service user to treat such an exception as non-fatal, replacing the underflow value by zero and continuing, or as fatal, which would be the default.

d) invalid string or list operation, e.g. overflow upon concatenation, attempt to perform an operation undefined for an empty string or list.

e) operation undefined for value.

f) attempt to perform operation on an undefined value.

g) attempt to delete a non-existent item.

h) unable to execute call (e.g. named service function unavailable).

i) attempt to open file that cannot be found.

j) attempt to open file that is already open.

NOTE Perhaps non-fatal though it can indicate incorrect file naming by the service user.

k) illegal file name.

NOTE File names can be generated dynamically.

l) attempt to access (for input or output) file to which access is unauthorized.

NOTE It is advisable not to require in the language-independent service specification the provision of an unnecessary amount of information or lower levels of security than provided by the host environment. Any resulting message is only meaningful for a legitimate user who has merely omitted to unlock a protected file for read or write access, and who is able to obtain the needed information and take the necessary action without direct assistance from the implementation.

m) attempt to input from an output-only file (e.g. printer stream) or to output to an input- only file (e.g. keyboard).

n) attempt to create an item that already exists.

o) attempt to replace a non-existing item.

p) attempt to close file already closed.

q) insufficient system resource (e.g. memory) available for specified operation.

r) time limit exceeded.

s) limit on complexity (e.g. depth of recursion) exceeded.

t) use of non-standard dynamic implementation-defined extension.

14.4 Guideline: Recovery from non-fatal faults

Where the specification permits recovery mechanisms from fault conditions, the required results of the actions to be taken by the implementation (when such a recovery mechanism is invoked) should be defined as fully as are defined the normal features of the service.

15 Guidelines on options and implementation dependence

15.1 General

Options present a special problem for service specifications in general (as indeed for other things), and for language-independent service specifications in particular. Optional aspects of the service, which some implementations have but others do not, can harm interoperability and create difficulties for applications using the service. These applications can need the options and rely on them to be present, or at least need to know (possibly in advance) whether they are there or not. Options in the service

interface, or in bindings, can create uncertainties and difficulties, and damage interoperability of applications using different implementations of the service.

From that point of view, the best course of action is to eliminate options altogether. All implementations, interfaces and bindings therefore provide the identical service. However, in some environments, providing everything can be technically difficult, prohibitively expensive or actually impossible. A partial service can be offered but it would not conform to the specification and (if the specification is embodied in a standard) users of it would hence not be able to rely on the assurance that conformity to the standard (possibly backed up by testing and certification) would bring. In some environments or for some purposes, providing everything can be indeed unnecessary; parts would never be used yet the costs of them would remain.

As for implementation dependence, removing it altogether can make implementation impossible on some systems. In contrast, it is possible that some systems are able to provide a better level of service (as measured by, say, speed or capacity) than that specified, but would be prevented from doing so. Where characteristics so measured need to be identical to permit interoperability, this can be unavoidable. Yet, a situation can arise in which implementation, interface, binding and application all operate at a higher level, without affecting anything outside, but doing so would be precluded.

Services, the way they are used, and the circumstances in which they are used, vary so much that there can be no general rule covering every case. Those responsible for developing the specification, such as a standards committee, need to weigh all the various factors and make a decision – probably a compromise – accordingly. The guidelines in this clause are aimed at helping them to do that.

15.2 Guidelines on service options

15.2.1 Guideline: Optional service features

Inclusion within the language-independent service specification of optional features of the service, as offered through the interface to users whether as optional additions or as optional alternatives, should be minimized. Ideally, the aim should be to have no optional features at all.

NOTE In general, optional features harm interoperability, and they can create difficulties for applications using the service, especially those that deal with different implementations of the service.

15.2.2 Guideline: Avoidance of assumptions about the use of the service

When determining whether to allow a feature of a service to be optional, the fewest assumptions as possible should be made about how or for what purpose the service will be used.

NOTE 1 Experience shows that options often arise because it is assumed that a certain feature is rarely used, or is required only for certain expected uses of the service. Problems are thereby created during the transitional period where many implementations omit the option until experience has shown that it is more widely used than anticipated by the designers of the service.

NOTE 2 Experience can also show that features made mandatory are in fact used only rarely. In this case, a possible course of action is to make the feature optional in a future revision (see Clause 18).

15.2.3 Guideline: Use of query mechanism

The language-independent service specification should recommend that that every implementation of that service should provide a mechanism by which the application can determine which optional features are available.

15.2.4 Guideline: Management of optional service features

Where complete avoidance of service options is impracticable:

— they should be organized in a consistent manner, and the number of different levels should be minimized;

— if an implementation provides a conforming optional service feature that is not required for the subset for which conformity to the language-independent service specification is claimed, then the specification should require that, nevertheless, the implementation provides that feature in accordance with the requirements of the specification;

— the language-independent service specification should require that every implementation omitting any optional features provides, either internally or through the interface, a defined response to any service user request for a feature not provided by that implementation.

15.2.5 Guideline: Definition of optional features

If at all possible, any optional (or higher level) features of the service should be defined functionally in terms of mandatory (or lower level) features.

15.3 Guidelines on interface options

15.3.1 Guideline: Completeness of interface

The service interface specification should allow bindings and user applications access to all the features of the service that are offered externally.

NOTE Even when the interface is conceived to be for a given, limited purpose not seeming to require certain features of the service, some applications can have unforeseen uses of those features, and a well-designed interface will allow for such uses (see 15.2.2).

15.3.2 Guideline: Interface to service with options

Where the interface is to a service that has optional features, the interface specification should reflect this but still be able to handle invocations of those features by user applications, and provide an appropriate response.

NOTE Adopting this strategy makes it easier to update the interface in case these unimplemented options become available later.

15.4 Guidelines on binding options

15.4.1 Guideline: Completeness of binding

Unless unavoidable, a language binding to the service interface specification should allow user applications access to all the features of the service, whether mandatory or optional, that are available through the interface. If omissions are unavoidable, they and the reasons for them should be fully documented.

NOTE 1 A well-designed language binding makes no assumptions about how or for what purpose applications written in the language will use the service (see 15.2.2).

NOTE 2 Omitting features can be unavoidable though, for example, if the interface assumes the presence of a facility that is not available in that language. (This is at least as likely to be the result of the service specification not being sufficiently language independent as it is to be the result of a shortcoming in the language.)

15.4.2 Guideline: Binding to a service with options

Where the language binding is to a service that has optional features, the following possibilities should be considered:

— reflecting the service options in options in the binding;

— requiring certain options to be available before binding is possible;

— using suitable language facilities to provide enquiry functions, allowing language users to determine, in a given environment, whether a given option is available or not.

15.4.3 Guideline: Binding to a language with optional features

Where the language binding is to a language that has optional features, the binding should make use of the full power of the language, provided

— it does not require syntax additional to that allowed by the standard for the full language;

— an alternative binding, where possible, is provided in cases where the preferred binding uses an optional language facility which is absent from a standard-conforming language processor which omits that option (e.g. a subset language implementation);

— such alternative binding is totally equivalent to preferred binding as far as providing access to the service feature is concerned.

NOTE 1 For some languages, the "full power" includes the ability to add user-defined datatypes, operations, modules etc. Exploitation of this full power gives maximum benefit for those language users needing access to the service.

NOTE 2 Strictly speaking, this guideline does not belong in this document but in ISO/IEC TR 10182. The guideline is however kept in the spirit of this document.

15.5 Guidelines on implementation dependence

15.5.1 Guideline: Completeness of definition

The number of aspects within its scope that the language-independent service specification leaves not completely defined should be minimized (and preferably eliminated altogether). Where full definition is impracticable, in general such aspects should be required to be implementation-defined, subject where appropriate to specified minima or other limits, rather than left as implementation-dependent or undefined. In this case, a complete checklist should be provided of all such implementation-defined features, guidance should be provided for implementors, required limits, as appropriate, should be specified, and the documentation accompanying the implementation should be required to provide for the user a full specification of the implementation definitions used.

NOTE The crucial phrase above is "within its scope". It is important to avoid over-specifying requirements by going beyond the scope of the specification by specifying how results are achieved as well as what results are achieved. Such over-specification often means that, for languages with facilities other than those envisaged, either instead or in addition, implementations are pointlessly constrained, and can be less efficient than they can be.

15.5.2 Guideline: Provision of implementation options

The language-independent service specification should specify implementation options required to be provided by a conforming implementation, including in each case a specification of standard default settings of the option and the form or forms in which the implementation options are to be made available to the service interface, bindings, and user applications.

NOTE There is also the possibility of the language-independent service specification leaving some things undefined to be defined by the binding, or for the binding to decide whether to define something or leave it to the implementation to do so. All such things should be explicitly stated in the language-independent service specification.

15.5.2.1 Checklist of potential implementation options

Writers of language-independent service specifications should consider all of the following features as potential areas for specifying implementation options, and the specification produced should address

all that are appropriate for the service and for the kinds of implementation and binding language covered:

— the handling of non-standard features;

— the use of system-dependent or implementation-dependent features;

— the type(s) of optimization;

— the handling of faults and warning messages;

— the handling of overflow and similar range checking;

— operating modes;

— the use of pre-connected files and their status on termination;

— the rounding or truncation of arithmetic operations;

— the precision and accuracy of representation and of arithmetic, as appropriate;

— the default settings of service parameter values;

— in the case where the specification is a revision of an earlier specification, the detection and reporting, of usage incompatible with the old specification.

15.5.3 Guideline: Implementation-defined limits

Minimum guaranteed service levels to be supplied by conforming implementations should provide advice on choice of actual levels and be specified in appropriate circumstances, namely where:

a) it is probable that user service demands encounter implementation-defined limits during execution; and

b) such limits can be expressed in terms of the nature of the demand (rather than service implementation issues which can be unpredictable or variable, such as resource capacity).

15.5.3.1 Checklist of potential implementation-defined limits

Examples of features for which it can be appropriate to specify minimal limits in specifications are:

— length of user-supplied or internally-generated character strings handled;

— range of integers;

— internal precision of values of datatype Real;

— magnitude of values of datatype Real;

— number of user files which can be open simultaneously;

— complexity of service requests that can be handled.

15.5.3.2 Actual values of limits

When advising implementors on considerations involved in setting the actual values of implementation-defined limits, note that such advice can do one or more of:

— recommending specific values;

— recommending minimum useful values;

— recommending maximum useful values;

— recommending that limits should depend on implementation thresholds where efficiency changes sharply;

— recommending that limits should depend on resource availability, which can fluctuate during execution;

— setting forth other criteria appropriate to the specific service.

In each case, the reasons for the recommendations should be explained. Different recommendations can be appropriate for different limits.

It should be noted that appropriate implementation-defined limits need to be made accessible to users, in particular for those performing conformity testing, as well as being documented. Where this is not available through service facilities (such as user "help" facilities), appropriate guidance to implementors should be provided.

16 Guidelines on conformity requirements

16.1 General

If the specification (of the service, the interface, or a binding) is to be included in a formal standard, then formal conformity rules are required. This clause provides guidelines on how to do this.

NOTE 1 Much of this clause is based on the guidelines in ISO/IEC TR 10034.

Even if the specification is not being designed with formal standardization in mind, any document containing it needs to make clear what is required to meet the specification. However, it can feel unnecessary to make the conformity requirements as strict as those in a formal standard should be.

The guidelines here are based on the assumption that strict and formal conformity rules are required. If the circumstances are regarded as less demanding, then some relaxation can be possible. However, in general, a rigorous definition of what is required is helpful to both implementors and users of any service, standardized or not, since there cannot then be any doubt.

What kinds of entity are expected to conform to the specification, and the rules to be laid down for such conformity, depends greatly upon the nature of the service being specified, and in that respect these guidelines can do no more than indicate general principles. In respect of the language-independent nature of the specifications, the consequent relationship with actual languages, and what this implies for conformity rules, these guidelines can be more specific.

There are three kinds of possible relationship between a programming language and a language-independent service specification, which can be summarized as implementation, invocation, and incorporation.

Implementation means that the language is used to implement the service. At its simplest, the service is provided to the user as an executing program. The user, in general, does not know, and should not need to know, what language the program is written in.

NOTE 2 This relationship is relevant for a specification of the service. Though the specification of services in general is outside the scope of this report, specification in respect of language independence is in scope, and is covered in 16.2.

Invocation means that the user of the language is able to call upon the service from a program in that language. The term reflects the familiar simple case of a user invoking a procedure from a procedure library. The essence in this case is that the service is logically external to the language, but language users can invoke it. The implementation language for the service is not necessarily the same as the language from which it is called.

Incorporation means that the service is provided by the language. That is, the service is internal to the language – it is included in the language definition. This does not mean that the language is also

the implementation language: languages designed for particular applications domains are often implemented in other languages designed for systems implementation.

Both invocation and incorporation require language bindings to the service, which, in general, are rather different in style. Furthermore, for a particular service and language, invocation and incorporation are not necessarily totally mutually exclusive. Normally, a language has only one binding to the service, which is an invocation-style binding, an incorporation-style binding, or a mixture (since, for a particular service and language, invocation and incorporation are not necessarily totally mutually exclusive).

Conformity rules need to cover all these three cases, and are different in general. Those covering implementation conformity are the ones which most depend on the nature of the service. The guidelines for specifying conformity of implementations (see 16.2 and 16.3) are confined to general principles and how to avoid making undue assumptions about the nature of the implementation language. Conformity rules covering invocation and incorporation are rules for conformity of the language bindings. The guidelines for specifying conformity of language bindings (see 16.4) address how to avoid making undue assumptions about the nature of the binding or the language being bound to, and how to make the language bindings as simple as possible. The general guidelines for specifying the service address those questions, but they also need to be addressed here, since it is possible for a good language-independent specification to be subverted by undue assumptions in the conformity rules.

16.2 Guidelines for specifying conformity of implementations of the service

16.2.1 Guideline: Avoidance of assumptions about the implementation language

The conformity requirements on implementations should not make assumptions about the style or mode of provision of facilities of the implementation language.

16.2.2 Guideline: Avoidance of representational assumptions

The conformity of implementations should not depend on representational requirements, or requirements that make representational assumptions.

16.2.3 Guideline: Avoidance of implementation model

The conformity requirements on implementations should not assume or require an explicit implementation model.

16.2.4 Guideline: Requiring end results rather than methods

The conformity clauses for implementations of the service should make very clear that is the end result that matters, not how it is achieved.

NOTE The same holds, of course, for the normative text of the specification.

16.3 Guidelines for specifying conformity of implementations of the interface

16.3.1 Guideline: Requirements on implementation-defined aspects

Conformity requirements for implementations of the interface should address implementation-defined aspects of the service (e.g. maxima or minima of implementation- defined values), even if the specification of the service does not.

NOTE Such requirements assist in defining language bindings to the interface, since they help to determine the minimum level of service that a language user can expect from a binding.

16.4 Guidelines for specifying conformity of bindings

16.4.1 Guideline: Propagating requirements to conforming bindings

The conformity rules for the service should include requirements on the conformity rules that language bindings apply to their implementations, in order to propagate requirements of the interface to conforming bindings and ensure an adequate level of consistency between bindings for different languages.

16.4.2 Guideline: Adherence to defined semantics

The conformity rules of a language-independent service specification should require that any conforming language binding shall adhere strictly to the defined semantics of the service.

NOTE 1 See NOTE 1 in 17.3.

NOTE 2 Other guidelines in this document make it clear, however, that conformity rules are discouraged from imposing on bindings inflexible requirements that are inessential to the correct functioning of the service.

17 Guidelines on specifying a language binding to a language-independent interface specification

17.1 General

Guidelines on the development of language bindings can be found in ISO/IEC TR 10182. The following additional guidelines are recommended for use when binding to a language-independent interface specification.

17.2 Guideline: Use of bindings to LID and LIPC

Language bindings to a language-independent service specification should make maximum use of existing bindings for the language to the language-independent standards ISO/IEC 11404 and ISO/IEC 13886.

NOTE 1 See Clauses 12 and 13.

NOTE 2 For some services whose invocation can be expressed completely in terms of procedure calls and associated parameters and for languages whose bindings to LID and LIPC are complete, it can be possible to produce a very simple, near-automatic binding which simply lists the service's datatypes and procedures and the corresponding bindings.

NOTE 3 However, it is possible that name correspondence requirements still be necessary: see 7.2.2 e).

17.3 Guideline: Adherence to defined semantics

A language binding to a language-independent service specification should adhere strictly to the defined semantics of the service, even when the conformity rules for the service specification do not make such a requirement.

NOTE 1 If different semantics exist for different bindings, this causes confusion among users, possibly resulting in errors that are difficult and expensive to put right. Strict adherence to the defined semantics is clearly important when interoperability is required between applications using different languages and language bindings, but even when interoperability between one language platform and another is not an issue, portability and consistency of the same application between different language platforms is jeopardized if the defined semantics are departed from.

NOTE 2 It can be appropriate to include in the language binding specification the specification of the service itself (clearly shown as informative, not normative), either as a separate section or annex, or interleaved with related binding definitions.

NOTE 3 A binding which redefines the semantics of a service, hence contrary to this guideline, has sometimes been termed a "thick" binding, as opposed to a "thin" binding which adheres to the defined semantics as this guidelines recommends. However, the terms "thick" and "thin" in relation to bindings have tended to cause much confusion and misunderstanding, and are best avoided.

17.4 Guideline: Binding document organization

A language binding to a language-independent interface specification should be designed to include the following parts (though it should not necessarily be confined to only to the parts listed).

NOTE In general, a binding to a language-independent interface specification is a simpler document than the specification itself, however, the relation of the intended structure of the bindings to that of the language-independent service specification is to be carefully considered. There are advantages in keeping them as similar as possible. The list below is therefore a shortened and adapted form of the checklist in 7.2.2.

1) A definition of the binding of the language to the interface, including rules for conformity of implementations.

2) The specification of all further requirements on standard-conforming implementations of the binding (such as fault detection, reporting and handling; provision of implementation options to the user; documentation; validation; etc.), and of rules for conformity.

3) One or more annexes containing an informal description of the service and of the interface, a glossary (including an explanation of any differences between the terminology used in the language-independent interface specification and that used in the language standard), guidelines for service users (on implementation-dependent features, documentation available, etc.), and a cross-referenced index to the document.

4) An annex explaining the name correspondence between names used in the interface specification and names used in a calling program.

5) An annex containing one or more checklists of any implementation-defined features.

6) An annex containing guidelines for implementors, including short examples where appropriate.

7) An annex providing guidance to users of the binding on questions relating to the validation of conformity, and any specific requirements relating to validation contained in (1) and (2) above.

8) In the case where the binding is a revision of an earlier version, an annex containing a detailed and precise description of the areas of incompatibility between the old version and the new version.

   NOTE Where the revision has been prompted by a revision of the language standard rather than of the service, it can be sufficient to summarize the relevant equivalent information given in the revised language standard, together with a summary of any new language features used in the binding.

9) An annex which forms a tutorial commentary containing examples that illustrate the use of the service from within the language.

17.5 Guideline: "Reference card" binding documents

Consideration should be given to production of a "reference card" style of binding document, consisting simply of a listing of the elements of the interface and the corresponding syntax in the language concerned.

NOTE 1 This can be provided in various ways: as a separate document for purposes of quick reference, as an informative annex to a full binding, as a normative binding with the detailed material in informative annexes, or even (e.g. in very simple cases) as the entire binding document.

NOTE 2 The reference card form of documentation has been shown to be popular and effective for commercial products, and the semantics can always be found by reference to the language-independent specifications.

18 Guidelines on revisions

18.1 General

The revision of any specification can create problems for users accustomed to the previous version, and always needs to be carried out with care. In the case of a language-independent service specification, a revision of the service specification, interface specification, language binding, or language of a language binding can occur.

Only a revision of a language binding can be done without, in principle, potentially affecting the others. Revision of a language can require revision of the corresponding language binding. Revision of the interface specification can require revision of some if not all language bindings. Revision of the service specification can require revision of the interface specification and hence to language bindings.

The guidelines in this clause are intended to assist in the planning of revisions in each of these categories.

NOTE Since the principles that apply to the revision of standards and specifications are much the same whatever the detailed subject matter, many of these guidelines are based on ISO/IEC TR 10176:2003, 4.5.4, with appropriate adaptation.

18.2 Kinds of change that a revision can introduce

18.2.1 General

In this subclause, “feature” is used neutrally, to denote any aspect of the specification, which is visible to the service user, and/or service implementor.

18.2.2 Addition of a new feature

A new feature is added to the specification without affecting existing features.

NOTE In the service specification, such a change can for example add a further facility to the service. In the interface specification, it can offer to users a facility of the service not previously visible (e.g. additional fault information). In a language binding, it can offer to users from that language community a facility of the service not previously made available.

18.2.3 Change to the specification of a well-defined feature

A change is made to the specification of a feature that is defined reasonably precisely in the previous version. The feature remains available, but has changed in some way.

NOTE In the service specification, such a change to the semantics of a feature can mean that invoking the feature through the interface can produce results different from before. In the interface specification, it can specify that a particular service facility (which can have remained unchanged) is now invoked in a different way. In a language binding, it can alter the way that the service is invoked, or the context in which such invocation is permitted.

18.2.4 Deletion of a well-defined feature

A feature which was well defined in the previous version is rendered invalid by the new specification.

NOTE Deletion of such a feature can imply that attempts to invoke it, now produce a fault condition, in whichever specification (service, interface, binding) it appears.

18.2.5 Deletion of ill-defined feature

A feature, which was not well defined in the previous version, is rendered invalid by the new specification.

18.2.6 Clarification of ill-defined feature

A feature, which was not well defined in the previous version, so that its interpretation was open to question, is properly defined in the new specification.

NOTE Though this can be regarded as correcting a defect in the previous version of the specification, it is possible that some past interpretations are not compatible with that in the revised version and so have a similar effect – in those cases – to changing the specification of a well-defined feature as in 18.2.3.

18.2.7 Change or deletion of obsolescent feature

A feature designated in the previous version as obsolescent is deleted or changed in the new specification.

18.2.8 Change of level definition

A level of service previously defined in the specification (whether service, interface, or binding) is altered in the new version.

18.2.9 Change of specified limit to implementation-defined value

A specified limit (maximum or minimum) in the previous version, for a value left implementation-defined by the specification, is changed in the new version.

18.2.10 Change of other implementation requirement

An implementation requirement in the previous version is deleted or changed in the new specification, or a new implementation requirement is added in the new version.

18.2.11 Change of conformity clause

A change in the conformity clause can introduce a change in the behaviour of an implementation.

18.3 General guidelines applicable to revisions

18.3.1 Guideline: Revision compatibility

For each proposed addition, deletion or modification that represents a potential incompatibility from an earlier version of the specification:

— the rationale for the proposed change should be stated;

— the way in which the proposed change affects the original feature should be determined, in accordance with the classifications in 18.2 above;

— the difficulty of converting any affected clients of the service should be assessed;

— an attempt should be made to determine how widely the affected feature is used;

— all the above should be documented, and conversion guidance should be provided in the relevant part of the language-independent service specification.

18.4 Guidelines on revision of the service specification

18.4.1 Guideline: Determining impact on interface and language bindings

Before any revision of the service specification is undertaken, potential changes should be reviewed and the consequent effect on the interface and on the dependent language bindings should be determined.

18.4.2 Guideline: Minimising impact on interface and language bindings

Any changes resulting from a revision of the service specification should be specified in such a way as to minimize the difficulty of revising the specifications of the interface and of the dependent language bindings.

18.4.3 Guideline: Use of incremental approach to revision

Because of possible effects on interface and dependent bindings, the revision of a service specification should if possible be carried out in small steps, correcting, incrementing or modifying a part of the specification at a time, to allow similar correcting, incrementing or modifying of the dependent specifications.

18.5 Guidelines on revision of the service interface

18.5.1 Guideline: Buffering unrevised bindings from changes

If possible, a revision of the service should be specified in order to absorb the changes, to ensure that unrevised language bindings still work.

NOTE 1 Techniques include the provision of conversion facilities or optional forms, which can be marked as "obsolete", to be removed at the next revision. This allows time for the language binding specifications to catch up with the revised service.

NOTE 2 It is advisable to revise the interface to provide special help to calls from applications still using outdated bindings.

18.5.2 Guideline: Use of incremental amendments

Where the revision of the service takes the form of additional features, an incremental amendment for language bindings should be specified, to allow the new features to be accessible to applications.

NOTE An incremental amendment to a binding is in general quicker and simpler to provide than a complete revision. The additional features can be integrated with the others at some future time when a full revision is warranted.

18.6 Guidelines on revision of language bindings following revision of the service interface

18.6.1 Guideline: Buffering application programs from changes

When a language binding is revised in response to a revision of the service interface, it should as far as possible shield application programs dependent on the service from any changes to the interface.

NOTE A change to an implementation of a language binding which allows application programs using it to remain unaltered can avoid costly and time-consuming modifications to many such programs.

18.6.2 Guideline: Use of incremental amendments

Where revision of the service interface adds to the functionality available to application programs, an incremental amendment to the language binding to accommodate invocation of the new features, without invalidating invocation not using those features, provides the benefits to new or revised application programs without requiring modification of all such programs.

18.7 Guidelines on revision of a language binding following revision of the language

18.7.1 Guideline: Use of new language features

When a language binding is revised in response to a revision of the language specification, relevant new language features should be exploited to allow new or revised application programs full advantage of those new features.

18.7.2 Guideline: Buffering "legacy" application programs from changes

Where application programs not yet revised are expected to continue in use, the revised language binding should where necessary make use of the "backward compatibility" provisions of the language revision to allow old "legacy" programs to continue to work.

NOTE It is common practice when revising language definitions either to make them fully compatible with the previous definition, or to make special provision for legacy programs.

18.7.3 Guideline: Buffering application programs by use of options

If necessary, the revised language binding should specify a user option to allow application programs to continue using the old form of invocation, even if the language revision itself makes no provision for legacy programs.

NOTE 1 This implies that implementation of the binding contains the necessary conversions that are absent from the revised language definition.

NOTE 2 Since it is undesirable for language bindings to remain out of step with the language definition for any longer than necessary, it is desirable to mark such a user option as obsolescent, to be removed at the next revision of the binding.

Annex A
(informative)

Brief guide to language-independent standards

A.1 Language-independent arithmetic

ISO/IEC 10967-1 (*Language independent arithmetic Part 1, Integer and floating point arithmetic*, hereafter "the LIA-1 standard") provides rigorous definitions of the basic arithmetic operations on integer and floating point values. It is at a higher level of abstraction than IEC 559 (IEEE 754), which specifies an abstract representational and implementation model; though for mathematical reasons it adopts a floating-point notation for definitions relating to "real" (i.e. approximate) values, it does not require or assume that a floating-point representation will be used in implementations. The relationship with IEC 559 is clearly described, for convenience of use with systems that make use of that implementation model. IEC 559 systems can conform to the LIA-1 standard, and many do, but the requirements of IEC 559 are less rigorous, primarily because of the presence of options, so conformity with IEC 559 does not guarantee conformity with LIA-1.

ISO/IEC 10967-2 covers elementary numerical functions. ISO/IEC 10967-3 covers complex arithmetic and procedures).

A.2 Language-independent datatypes

The purpose of ISO/IEC 11404 is to define a set of datatypes suitable for use as "common ground" between a wide variety of programming languages, and other things which use the concept of "datatype" explicitly or implicitly. The set defined is very rich, far wider than is usual in programming languages, to accommodate the needs of many very different languages and allow them to find LID datatypes corresponding to their own datatypes with the minimum distortion. For the similar reasons, operations on the values are "characterizing", i.e. typical of the datatype concerned, but are not normative, since which operations are provided in a given context, and which are not, are very much dependent on the envisaged field of applications. Again, in almost all cases the datatypes are "purely computational", i.e. concerned with pure values, without any particular semantic or applicational connotations, and capable of use in a wide variety of application fields. In this, LID follows the mainstream tradition of general-purpose languages, as can be seen from the choice of primitive datatypes, though provision is made for the construction of further datatypes which, if necessary, can be application-oriented.

The LID standard follows the common practice of starting with a number of primitive datatypes and then using these to construct others. There are three main kinds of constructed datatypes: subtypes, generated datatypes, and aggregates. (In fact aggregates are technically also generated datatypes, but important enough to deserve separate classification.)

Primitive datatypes are datatypes whose values are regarded fundamental - not subject to any reduction. Many primitive LID datatypes are also generic, in the sense that they have an unlimited number of values, and hence the datatypes often used in practice are confined to a finite subset of them. The reason that they are used, rather than "actual" achievable datatypes, is threefold: it is a convenient way to identify a class of datatypes which is infinite in extent; language definitions commonly use them, thus simplifying the binding between the LID datatypes and the ones used by specific languages; and it allows for the possibility of actually supporting them if a language is designed to do so.

The LID primitive datatypes are Boolean, State, Enumerated, Character, Ordinal, Date-and- time, Integer, Rational, Scaled, Real, Complex and Void – a much longer list than most languages support, for the reasons stated above. Note that Date-and-time is the only one with particular semantic connotations; the exception was made partly because time is so universal a concept, and partly because some mainstream general-purpose languages include it.

Subtypes are created by modifying the value-space of a "base" datatype in various ways – specifying a range or size; selecting values; excluding values; extending the value-space; or defining explicitly how the value-space is constructed from that of a "base" datatype. Any combination of these is possible too. This carries the implication that a subtype can have a wider value space than its base datatype. However, any datatype can be used as the base, not just the primitive ones, and in that context extension is a useful subtype constructor, e.g. you can make a new subtype by extending an existing one.

The (non-aggregate) generated datatypes in the LID standard are Pointer, Procedure, and Choice datatypes, produced from other datatypes by the methods familiar from languages that include them.

The main kinds of aggregate in the LID standard are Bag, Set, Record, Sequence, Array, and Table. Bag is the most general form of aggregate datatype, capable of containing anything; additional properties or constraints are then used to identify various kinds of aggregate datatype that are encountered computationally. These properties and constraints are not all mutually orthogonal; they can interact with others, in various ways. The particular mix of properties used for a given aggregate datatype depends on the envisaged computational uses of the datatype and its values. The LID standard provides a way of constructing aggregates as needed, by appropriate mixing and matching of a relatively small number of properties. These include homogeneity, size, and the ability for components to be extracted by tagging, keying or indexing. Multidimensionality is allowed for.

Finally, the LID standard allows for new datatypes to be produced from existing ones (copies, or "clones") and for further datatypes and datatype generators to be derived from the basic primitive and generated ones - Tree, CharacterString and BitString are examples. Non-aggregate derived datatypes include Bit, Modulo, and TimeInterval.

NOTE Some of the above text has been adapted from the article A taxonomy of datatypes published in Reference [18].

A.3 Language-independent procedure calling

ISO/IEC 13886 defines a language-independent model of procedure calling of sufficient abstraction to allow the procedure calling facilities of many languages to communicate. An LIPC parameter can be of any datatype definable via ISO/IEC 11404. No distinction is made between "function" procedures that return a value through the procedure name (and hence can be called to provide values to expressions directly), and "subroutine" procedures that do not return a value in such a way. If the language allows for "function" procedures, the language binding maps the return through the procedure name of the evaluated value of the function into an additional parameter of the corresponding LIPC invocation.

A language processor offering LIPC server facilities for a procedure maps the LIPC procedure call definition, including the number and datatypes of formal parameters, into the form of the corresponding procedure call in the language on its side, using the LIPC binding for that language. A language processor offering LIPC client facilities can then invoke that procedure, in terms of the language on that side. The actual parameters are converted by the LIPC client facilities from the local datatypes to the LID datatypes required for the formal parameters, using the LID binding for the language; this process is termed marshalling.

Transmission of the procedure invocation and parameters to the service provider side is outside the scope of LIPC. Once received by the service provider side, the LIPC server facilities unmarshal the marshalled actual parameters from the LID datatypes into the local datatypes used by service provider mapping of the LIPC procedure call definition. Return of results is performed by a reverse process of marshalling on the service provider side and unmarshalling on the service client side.

LIPC specifies four abstract modes of parameter passing: call by value sent on initiation; call by value sent on request; call by value returned on termination; and call by value returned when available. In combination with the datatyping facilities in LID, these four modes cover all of the logical possibilities that binding standards or those implementing standards-conforming language services are likely to need. The standard explains how the common forms of parameter passing found in languages can be expressed with them.

The standard also provides an abstract model of the execution of a procedure call. There is no normative requirement for this execution model to be implemented; it is included as a guide for those defining and implementing LIPC services, to aid understanding and reduce the risk of incompatibilities between language bindings and implementations of client-side and server-side facilities in language processors.

Annex B
(informative)

Glossary of language-independent terms

B.1 General

This glossary is derived from the terminology used in the language-independent standards described in Annex A, together with other standards of importance to language-independent specifications, for example those relating to character sets and coding. The definitions are interleaved to assist comparison; sources are indicated as described in A.1. For convenience of comparison, the definitions from Clause 3 are repeated.

B.2 Source indications

GLB ISO/IEC TR 10182:2016, *Guidelines for language bindings*

LIA ISO/IEC 10967-1:2012, *Language independent arithmetic, Part 1, Integer and floating point arithmetic*

LID ISO/IEC 11404:2007, *General-Purpose Datatypes (GPD)*

LIPC ISO/IEC 13886:1996, *Language-Independent Procedure Calling (LIPC)*

LISS ISO/IEC TR 14369, *Guidelines for the preparation of language independent service specifications* (i.e. this document; for these entries the Notes from the full definitions in Clause 3 are omitted)

B.3 Index of terms

For consistency of presentation, minor editorial changes have been made to the original formats, but otherwise the definitions appear as published. No omissions have been made, even when the definitions appear to be remote from the concerns of this document; this is to avoid possible misunderstanding, and to assist potential users who can be unsure whether the referenced document is relevant to a project. The only omissions have been the explanations of abbreviations that are included with definitions of terms in the same Clause of ISO/IEC TR 10182.

Where Notes are appended, they are marked as either original, i.e. appearing in the document referenced, or additional, i.e. added for the purposes of this document. The two kinds of Notes are kept separate.

abstract service interface (GLB)

An interface having an abstract definition that defines the format and the semantics of the function invoked independently of the concrete syntax (actual representation) of the values and the invocation mechanism.

actual parameter (LIPC)

A value that is bound to a formal parameter during the execution of a procedure.

actual parametric datatype (LID)

A datatype appearing as a parametric datatype in a use of a datatype generator, as opposed to the formal-parametric-types appearing in the definition of the datatype generator.

actual parametric value (LID)

A value appearing as a parametric value in a reference to a datatype family or datatype generator, as opposed to the formal-parametric-values appearing in the corresponding definitions.

aggregate datatype (LID)

A generated datatype each of whose values is made up of values of the component datatypes, in the sense that operations on all component values are meaningful.

alien syntax (GLB)

Syntax of a language other than the host language.

annotation (LID)

A descriptive information unit attached to a datatype, or a component of a datatype, or a procedure (value), to characterize some aspect of the representations, variables, or operations associated with values of the datatype which goes beyond the scope of this International Standard.

approximate (LID)

A property of a datatype indicating that there is not a 1-to-1 relationship between values of the conceptual datatype and the values of a valid computational model of the datatype.

arithmetic datatype (LIA)

A datatype whose values are members of ***Z***, ***R***, or ***C***.

NOTE (original) - This standard specifies requirements for integer and floating point datatypes. Complex numbers are not covered by this standard, but will be included in a subsequent part of this standard.

association (LIPC)

Any mapping from a set of symbols to values.

axiom (LIA)

A general rule satisfied by an operation and all values of the datatype to which the operation belongs. As used in the specifications of operations, axioms are requirements.

bounded (LID)

A property of a datatype, meaning both bounded above and bounded below.

bounded above (LID)

A property of a datatype indicating that there is a value *U* in the value space such that, for all values *s* in the value space, *s* <= *U*.

bounded below (LID)

A property of a datatype indicating that there is a value *L* in the value space such that, for all values *s* in the value space, *L* <= *s*.

box (LIPC)

A model of a variable or container that holds a value of a particular type.

characterizing operations (LID)

(of a datatype) A collection of operations on, or yielding, values of the datatype, which distinguish this datatype from other datatypes with identical value spaces;

(of a datatype generator) A collection of operations on, or yielding, values of any datatype resulting from an application of the datatype generator, which distinguish this datatype generator from other datatype generators which produce identical value spaces from identical parametric datatypes.

client interface binding (LIPC)

The possession by the client procedure of an interface reference.

client procedure (LIPC)

A sequence of instructions which invokes another procedure.

complete procedure closure (LIPC)

A procedure closure, all of whose global symbols are mapped.

component datatype (LID)

A datatype which is a parametric datatype to a datatype generator, i.e. a datatype on which the datatype generator operates.

configuration (LIPC)

Host and target computers, any operating system(s) and software used to operate a processor.

continuation value (LIA)

A computational value used as the result of an arithmetic operation when an exception occurs. Continuation values are intended to be used in subsequent arithmetic processing. (Contrast with exceptional value).

NOTE 1 (original) The infinities and NaNs produced by an IEC 559 system are examples of continuation values.

NOTE 2 (additional) Here, "IEC 559 system" means a system conforming to IEC 559 (IEEE 754).

datatype (LIA)

A set of values and a set of operations that manipulate those values.

NOTE 1 (additional) The purpose of the LIA-1 standard is to provide rigorous definitions of the basic arithmetic operations on integer and floating point datatype values. Hence, in the context of usage in that standard, the term "datatype" naturally includes the operations.

NOTE 2 In the LIA standard, the first to be published, "datatype" is spelled with a space, i.e. "data type". (The same is true for the GLB Technical Report.) For consistency in this (LISS) document, "datatype" is substituted throughout.

datatype (LID)

A set of distinct values, characterized by properties of those values and by operations on those values.

NOTE (additional) This definition is essentially identical to that in this document, though emphasizing the "characterizing" role of operations in helping to identify corresponding LID datatypes to those in a particular language.

datatype (LISS)

A set of values, usually accompanied by a set of operations on those values.

datatype declaration (LID)

(1) The means provided by this International Standard for the definition of a language-independent datatype which is not itself defined by this International Standard;

(2) An instance of use of this means.

datatype family (LID)

A collection of datatypes which have equivalent characterizing operations and relationships, but value spaces which differ in the number and identification of the individual values.

datatype generator (LID)

An operation on datatypes, as objects distinct from their values, which generates new datatypes.

defined datatype (LID)

A datatype defined by a type-declaration.

defined generator (LID)

A datatype generator defined by a type-declaration.

denormalization loss (LIA)

A larger than normal rounding error caused by the fact that denormalized values has less than full precision. (See float-rounding for a full definition.)

denormalized (LIA)

Those values of a floating point type F that provide less than the full precision allowed by that type.

embedded alien syntax (GLB)

Statements in a special language for access to a system facility, included in a source program written in a standard programming language.

error (LIA)

(1) The difference between a computed value and the correct value. (Used in phrases like “rounding error” or “error bound”.)

(2) A synonym for exception in phrases like “error message” or “error output”. Error and exception are not synonyms in any other context.

exact (LID)

A property of a datatype indicating that every value of the conceptual datatype is distinct from all others in any valid computational model of the datatype.

exception (LIA)

The inability of an operation to return a suitable numeric result. This might arise because no such result exists mathematically, or because the mathematical result cannot be represented with sufficient accuracy.

exceptional value (LIA)

A non-numeric value produced by an arithmetic operation to indicate the occurrence of an exception. Exceptional values are not used in subsequent arithmetic processing.

NOTE 1 (original) Exceptional values are used as part of the defining formalism only. With respect to this international standard, they do not represent values of any of the datatypes described. There is no requirement that they be represented or stored in the computing system.

NOTE 2 Exceptional values are not to be confused with the NaNs and infinities defined in IEC 559. Contrast this definition with that of continuation value above.

execution sequence (LIPC)

A succession of global states s1, s2, ... where each state beyond the first is derived from the preceding one by a single create operation or a single write operation.

exponent bias (LIA)

A number added to the exponent of a floating point number, usually to transform the exponent to an unsigned integer.

external identifier (GLB)

An identifier that is visible outside of a program.

formal parameter (LIPC)

The name symbol of a parameter used in the definition of a procedure to which a value will be bound during execution.

formal-parametric-type (LID)

An identifier, appearing in the definition of a datatype generator, for which a language-independent datatype will be substituted in any reference to a (defined) datatype resulting from the generator.

formal-parametric-value (LID)

An identifier, appearing in the definition of a datatype family or datatype generator, for which a value will be substituted in any reference to a (defined) datatype in the family or resulting from the generator.

functional interface (GLB)

The abstract definition of the interface to a system facility by which system functions are provided.

functional specification (GLB)

The specification of a system facility. In the context of this document, the functional specification is normally a potential or actual standard. For each system function the specification defines the parameters for invocation and their effects.

generated datatype (LID)

A datatype defined by the application of a datatype generator to one or more previously-defined datatypes.

generated internal datatype (LID)

A datatype defined by the application of a datatype generator defined in a particular programming language to one or more previously-defined internal datatypes.

generator declaration (LID)

(1) The means provided by this International Standard for the definition of a datatype generator which is not itself defined by this International Standard;

(2) An instance of use of this means.

global state (LIPC)

The set of all existing boxes and their currently assigned values.

global symbol (LIPC)

Symbol used to refer to values that are permanently associated with a procedure.

helper function (LIA)

A function used solely to aid in the expression of a requirement. Helper functions are not visible to the programmer, and are not required to be part of an implementation. However, some implementation-defined helper functions are required to be documented.

host language (GLB)

The programming language for which a standard language binding is produced; the language in which a program is written.

identifier (GLB)

Name of an object in an application program that uses a system facility.

implementation (of this standard) (LIA)

The total arithmetic environment presented to a programmer, including hardware, language processors, exception handling facilities, subroutine libraries, other software, and all pertinent documentation.

implementation-defined (GLB)

Possibly differing between different processors for the same language, but required by the language standard to be defined and documented by the implementor.

implementation defined (LIPC)

An implementation defined feature is a feature that is left implementation dependent by this International Standard, but any implementation claiming conformity to this standard shall explicitly specify how this feature is provided.

implementation-dependent (GLB)

Possibly differing between different processors for the same language, and not necessarily defined for any particular processor.

implementation dependent (LIPC)

An implementation dependent feature is a feature that shall be provided by an implementation claiming conformity to this standard, but the implementation need not to specify how the feature is provided.

implementor (GLB)

The individual or organization that realizes a system facility through software, providing access to the system functions by means of the standard language bindings.

input parameter (LIPC)

A formal parameter with an attribute indicating that the corresponding actual parameter is to be made available to the server procedure on entry from the client procedure.

input/output parameter (LIPC)

A formal parameter with an attribute indicating that the corresponding actual parameters are made available to the server procedure on entry from the client procedure and to the client procedure on return from the server procedure.

interface (LISS)

The mechanism by which a service user invokes and makes use of a service.

interface closure (LIPC)

A collection of names and a collection of procedure closures, with a mapping between them.

interface execution context (LIPC)

The union of the procedure execution contexts for a given interface closure.

interface reference (LIPC)

An identifier that denotes a particular interface instance.

interface type (LIPC)

A collection of names and a collection of procedure types, with a mapping between them.

interface type identifier (LIPC)

An identifier that denotes an interface type.

internal datatype (LID)

A datatype whose syntax and semantics are defined by some other standard, language, product, service or other information processing entity.

invocation association (LIPC)

The invocation association of a procedure closure *<Image, Association>* applied to a set of actual parameter values is the association of the closure augmented by a mapping of all local symbols to values and all formal parameter symbols to the corresponding actual parameter values. Thus it is a binding to values of all symbols in the procedure image for the duration of the invocation.

invocation context (LIPC)

For a particular procedure call, the instance of the objects referenced by the procedure, where the lifetime of the objects is bounded by the lifetime of the call.

inward mapping (LID)

A conceptual association between the internal datatypes of a language and the language-independent datatypes which assigns to each in datatype either a single semantically equivalent internal datatype or no equivalent internal datatype.

language (LISS)

Programming language, not specification language or natural (human) language, unless otherwise qualified.

language binding (LISS)

A specification of the standard interface to a service, or set of services, for applications written in a particular programming language.

language binding of *F* to *L* or

***L* language binding of *F* (GLB)**

A specification of the standard interface to facility F for programs written in language L.

language committee (GLB)

The ISO technical subcommittee or working group responsible for the definition of a programming language standard.

language-dependent (LISS)

State of making use of the concepts, features or assumptions of a particular programming language.

language-independent (LISS)

State of not making use of the concepts, features or assumptions of any particular programming language or style of language.

language-independent datatype (LID)

(1) A datatype defined by this International Standard, or

(2) A datatype defined by the means of datatype definition provided by this International Standard.

NOTE (additional) The LID standard abbreviates this term to "LI datatype"

language processor (LISS)

The entire computing system which enables a programming language user to translate and execute programs written in the language, in general consisting both of hardware and of the relevant associated software.

lower bound (LID)

In a datatype which is bounded below, the value *L* such that, for all values *s* in the value space, *L <= s.*

mapping (LID)

(of datatypes) A formal specification of the relationship between the (internal) datatypes which are notions of, and specifiable in, a particular programming language and the (language-independent) datatypes specified in this International Standard;

(of values) A corresponding specification of the relationships between values of the internal datatypes and values of the language-independent datatypes.

mapping (in general)

(noun) A defined association between elements (such as concepts, features or facilities) of one entity (such as a programming language, or a specification, or a standard) with corresponding elements of another entity. Mappings are usually defined as being *from* one entity *into* another. A language binding of a language *L* into a standard *S* usually incorporates both a mapping from *L* into *S* and a mapping from *S* into *L*.

(verb) The process of determining or utilizing a mapping.

NOTE (additional) This is essentially the same definition as for the LID standard, though necessarily made more general.

marshalling (LIPC)

A process of collecting actual parameters, possibly converting them, and assembling them for transfer.

NOTE (additional) The definition in this document is essentially identical, though spelled out more in the absence of the full context of the LIPC standard, and extended (in a Note) to preparing input values for a service.

marshalling (LISS)

The process of collecting the actual parameters used in a procedure call, converting them if necessary, and assembling them for transfer to the called procedure. (This process is also carried out by the called procedure when preparing to return the results of the call to the caller.)

normalized (LIA)

Those values of a floating point type F that provide the full precision allowed by that type.

notification (LIA)

The process by which a program (or that program's user) is informed that an arithmetic exception has occurred. For example, dividing 2 by 0 results in a notification.

operation (LIA)

A function directly available to the user, as opposed to helper functions or theoretical mathematical functions.

order (LID)

A mathematical relationship among values.

NOTE (additional) The LID standard also makes a cross-reference to its subclause 6.3.2.

ordered (LID)

A property of a datatype which is determined by the existence and specification of an order relationship on its value space.

output parameter (LIPC)

A formal parameter with an attribute indicating that the corresponding actual parameter is to be made available to the client procedure on return from the server procedure.

outward mapping (LID)

A conceptual association between the internal datatypes of a language and the language-independent datatypes which identifies each internal datatype with a single semantically equivalent language-independent datatype.

parameter (LIPC)

A parameter is used to communicate a value from a client to a server procedure. The value supplied by the client is the actual parameter, the formal parameter is used to identify the received value in the server procedure.

parametric datatype (LID)

A datatype on which a datatype generator operates to produce a generated datatype.

parametric value (LID)

(1) A value which distinguishes one member of a datatype family from another, or

(2) A value which is a parameter of a datatype or datatype generator defined by a type-declaration.

NOTE (additional) In relation to type-declaration the LID standard also makes a cross-reference to its subclause 9.1.

partial procedure closure (LIPC)

A procedure closure, some of whose global symbols are not mapped. Procedure closures may be complete, with all global symbols mapped, or partial with one or more global symbols not mapped.

precision (LIA)

The number of digits in the fraction of a floating point number.

primitive datatype (LID)

An identifiable datatype that cannot be decomposed into other identifiable datatypes without loss of all semantics associated with the datatype.

primitive internal datatype (LID)

A datatype in a particular programming language whose values are not viewed as being constructed in any way from values of other datatypes in the language.

procedural binding (GLB)

The definition of the interface to a system facility available to users of a standard programming language through procedure calls.

procedural interface definition language (GLB)

A language for defining specific procedures for interfacing to a system facility as used, for example, in ISO 8907 Database Language NDL.

procedure (GLB)

A general term used in this document to cover a programming language concept which has different names in different programming languages - subroutine and function in Fortran, procedure and function in Pascal, etc. A procedure is a programming language dependent method for accessing one or more system functions from a program. A procedure has a name and a set of formal parameters with defined datatypes. Invoking a procedure transfers control to that procedure.

procedure (LIPC)

The procedure value.

procedure (LISS)

In this document, the term “procedure” is used in the generic sense to cover both those (sometimes called subroutines) which do not return a value associated with the procedure name, and those (sometimes called functions) which do, and hence can be called from within expressions.

procedure call (LIPC)

The act of invoking a procedure.

procedure closure (LIPC)

A pair <*procedure image, association*> where the association defines the mapping for the image's global symbols and no others.

NOTE (original) Procedure closures are the values of procedure type referred to in ISO/IEC 11404:2007, *General-Purpose Datatypes (GPD)*.

procedure execution context (LIPC)

For a particular procedure, an instance of the objects satisfying the external references necessary to allow the procedure to operate, where these objects have a lifetime longer than a single call of that procedure.

procedure image (LIPC)

A representation of a value of a particular procedure type, which embodies a particular sequence of instructions to be performed when the procedure is called.

procedure invocation (LIPC)

The object which represents the triple: procedure image, execution context, and invocation context.

procedure name (LIPC)

The name of a procedure within an interface type definition.

procedure return (LIPC)

The act of return from the server procedure with a specific termination.

procedure type (LIPC)

The family of datatypes each of whose members is a collection of operations on values of other datatypes. Note, this is a different definition from procedure value.

procedure value (LIPC)

A closed sequence of instructions that is entered from, and returns control to, an external source.

processor (GLB)

A system or mechanism that accepts a program as input, prepares it for execution, and executes the process so defined with data to produce results.

processor (LIPC)

A compiler or interpreter working in combination with a configuration.

programming language extensions with native syntax or native syntax binding (GLB)

The functionality of the system facilities is incorporated into the host programming language so that the system functions appear as natural parts of the language. The compiler processes the language extensions and generates the appropriate calls to the system facility functions.

representation (LID)

(*of a language-independent datatype)* The mapping from the value space of the language-independent datatype to the value space of some internal datatype of a computer system, file system or communications environment;

(*of a value)* The image of that value in the representation of the datatype.

rounding (LIA)

The act of computing a representable final result for an operation that is close to the exact (but unrepresentable) result for that operation. Note that a suitable representable result may not exist.

rounding function (LIA)

Any function *rnd:* ***R*** *-> X* (where *X* is a discrete subset of ***R***) that maps each element of *X* to itself, and is monotonic non-decreasing. Formally, if *x* and *y* are in ***R***,

$$x \in X \Rightarrow rnd(x) = x$$

$$x < y \Rightarrow rnd(x) <= rnd(y)$$

Note that if $u \in \boldsymbol{R}$ is between two adjacent values in *X*, *rnd*(*u*) selects one of those adjacent values.

round to nearest (LIA)

The property of a rounding function *rnd* that when *u* in ***R*** is between two adjacent values in X, *rnd(u)* selects the one nearest *u*. If the adjacent values are equidistant from *u*, either may be chosen.

round toward minus infinity (LIA)

The property of a rounding function *rnd* that when *u* in ***R*** is between two adjacent values in X, *rnd(u)* selects the one less than *u*.

round toward zero (LIA)

The property of a rounding function *rnd* that when *u* in ***R*** is between two adjacent values in X, *rnd(u)* selects the one nearest 0.

server procedure (LIPC)

The procedure which is invoked by a procedure call.

service (LISS)

A facility or set of facilities made available to service users through an interface.

service provider (LISS)

A computer system or set of computer systems that implements a service and makes it available to service users.

service user (LISS)

An application (typically a program in some language) which makes use of a service.

shall (LIA)

A verbal form used to indicate requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted.

should (LIA)

A verbal form used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that (in the negative form) a certain possibility is deprecated but not prohibited.

signature (of a function or operation) (LIA)

A summary of information about an operation or function. A signature includes the operation name, the minimum set of inputs to the operation, and the maximum set of outputs from the operation (including exceptional values if any). The signature

add_I: I x I -> I ∪ {integer_overflow}

states that the operation named *add_I* shall accept any pair of *I* values as input, and (when given such input) shall return either a single *I* value as its output or the exceptional value *integer_overflow*. A signature for an operation or function does not forbid the operation from accepting a wider range of inputs, nor does it guarantee that every value in the output range will actually be returned for some input. An operation given inputs outside the stipulated input range may produce results outside the stipulated output range.

specification language (LISS)

A formal language for defining the semantics of a service or an interface precisely and without ambiguity.

subtype (LID)

A datatype derived from another datatype by restricting the value space to a subset while maintaining all characterizing operations.

symbol (LIPC)

A program entity used to refer to a value.

system facility (GLB)

A coherent collection of services to be made available in some way to an application program. The system facility may be defined as a set of discrete system functions with an abstract service interface.

system facility committee (GLB)

The ISO technical subcommittee or working group responsible for the development of the functional specification of a system facility.

system function (GLB)

An individual component of a system facility, which normally has an identifying title and possibly some parameters. A system function's actions are defined by its relationships to other system functions in the same system facility.

termination (LIPC)

A predefined status related to the completion of a procedure call.

unmarshalling (LIPC)

The process of disassembling the transferred parameters, possibly converting them, for use by the server procedure on invocation or by the client procedure upon procedure return.

NOTE (additional) The definition in this document is essentially identical, though spelled out more in the absence of the full context of the LIPC standard, and extended (in a Note) to receipt of input values by a service.

unmarshalling (LISS)

The process of receiving and disassembling transferred parameters, and converting them if necessary, to prepare the values for further use. This process is carried out by the called procedure on receipt of the actual parameters for the call, and by the caller on receipt of the returned results of the call.

upper bound (LID)

In a datatype which is bounded above, the value U such that, for all values s in the value space, s <= U.

value (LIPC)

The set Value contains all the values that might arise in a program execution.

value space (LID)

The set of values for a given datatype.

variable (LID)

A computational object to which a value of a particular datatype is associated at any given time; and to which different values of the same datatype may be associated at different times.

Z (LISS)

(1) (mathematics, e.g. ISO/IEC 10967-1:2012) the complex numbers

(2) (pronounced “zed”) a formal specification language, see ISO/IEC 13568.

Bibliography

[1] ISO 2382 (all parts), *Information technology — Vocabulary*

[2] ISO 8807, *Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour*

[3] ISO/IEC 90741,[1)]*Information technology — Open Systems Interconnection — Estelle: A formal description technique based on an extended state transition model Amendment*

[4] ISO/IEC/TR 10034, *Guidelines for the preparation of conformity clauses in programming language standards*

[5] ISO/IEC/TR 10176, *Information technology — Guidelines for the preparation of programming language standards*

[6] ISO/IEC/TR 10182, *Information technology — Programming languages, their environments and system software interfaces — Guidelines for language bindings*

[7] ISO/IEC 10967-1, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*

[8] ISO/IEC 10967-2, *Information technology — Language independent arithmetic — Part 2: Elementary numerical functions*

[9] ISO/IEC 10967-3, *Information technology — Language independent arithmetic — Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions*

[10] ISO/IEC 11404, *Information technology — General-Purpose Datatypes (GPD)*

[11] ISO/IEC 11578, *Information technology — Open Systems Interconnection — Remote Procedure Call (RPC)*

[12] ISO/IEC 13568, *Information technology — Z formal specification notation — Syntax, type system and semantics*

[13] ISO/IEC 13719, *Information technology — Portable Common Tool Environment (PCTE)*

[14] ISO/IEC 13817-1, *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language*

[15] ISO/IEC 13886, *Information technology — Language-Independent Procedure Calling (LIPC)*

[16] ISO/IEC 14977, *Information technology — Syntactic metalanguage — Extended BNF*

[17] ISO/IEC/IEEE 60559, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

[18] A taxonomy of datatypes published in ACM Sigplan Notices (Vol **29** No. 9, September 1994, pp 159–167)

1) Withdrawn.

ICS 35.060

Price based on 64 pages