ISO/IEC JTC1/SC22
Languages
Secretariat: CANADA (SCC)

ISO/IEC JTC1/SC22
N 1251

SEPTEMBER 1992

TITLE: Draft Paper on: Programming Language Independent Specification Methods

SOURCE: Secretariat ISO/IEC JTC1/SC22

WORK ITEM: N/A

STATUS: New

CROSS REFERENCE: N/A

DOCUMENT TYPE: Information document

ACTION: For information to SC22 Member Bodies.

Address reply to: ISO/IEC JTC1/SC22 Secretariat
J.L. Côté
Treasury Board Secretariat
140 O'Connor St., 10th Floor, Ottawa, Ontario, Canada, K1A 0R5
Tel.: (613)957-2496 Telex: 053-3336 Fax: (613)996-2690

WG11/N284

ISO/IEC JTC 1/SC22/WG15 N173

TCOS-LIS Draft 4

to SC22 for info

Draft TCOS-SSC Technical Report — Programming Language Independent Specification Methods

Sponsor

Technical Committee on Operating Systems and Application Environments
of the
IEEE Computer Society

Work Item Number: JTC 1.22.21.x.y

Abstract: *Programming Language Independent Specification Methods* provides guidance to the working groups in the TCOS Standards Subcommittee who are preparing language-independent specifications and language bindings for approval as IEEE standards and eventually ISO/IEC standards. It describes an interface model with abstract datatypes and abstract procedure calls, specifies the notational conventions associated with the model, and provides guidelines for the use of the model in programming-language-independent specifications and language bindings to them. Although this document has been purposely designed to look somewhat like a real standard, it is not an official IEEE or ISO publication.

Keywords: language-independent specifications, language bindings, datatype, procedure call

TCOS-LIS / D4
May 1991

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017, USA

May 1991

SHxxxxx

Contents

TABLES

Foreword

This TCOS-SS Technical Report describes methods for producing programming-language-independent functional specifications, and for producing programming language bindings to those functional specifications. These methods include abstract models and notations, together with guidelines for their use. These guidelines are intended to be followed by all functional specifications and language binding standards developed within the IEEE Technical Committee on Operating Systems.

The models and guidelines described in this document are adapted from current ISO/IEC JTC1/SC22/WG11 draft technical reports and guidelines concerning language-independent specifications and language bindings. Because there are no adopted standards or guidelines in this area, this document provides normative text to be included within all TCOS-SS functional specifications, concerning **Definitions**, **Conventions**, and **Conformance**. When ISO/IEC standards for language-independent specifications are adopted, these TCOS-SS guidelines can be amended to conform to those standards, where appropriate, and the normative text can be replaced by references to those standards. In the interim, successive revisions of these guidelines will be informed by changes in the ISO/IEC draft guidelines, as these become available.

Document Status

This document is a draft version of a proposed TCOS-SS Technical Report containing guidelines to be used by the TCOS-SS working groups to develop functional specifications in language-independent form and language bindings to those functional specifications.

Draft 4 is the first draft produced using the document build tools developed by Hal Jespersen and Donn Terry. Shaded margins are used in this draft to indicate Rationale or Editor's Notes. Text introduced or substantially modified in this draft is indicated by a small numeral (the draft number) in the right-hand margin. A topical index is provided in this draft, but is it buggy and substantially incomplete. The identifier index is empty. These shortcomings will be remedied in future drafts. 4

Section 6 (Conventions) has been revised to show the notation actually used in 1003.1LIS/D1. New material has been added to Sections 5, 8, and 9. The Annex A examples from previous drafts have been removed, and replaced by sample text from Section 5 of 1003.1LIS/D1. The list of open issues in Annex B has been removed, since it is no longer current. In some cases, relevant text has been added to the appropriate sections in the body of the document. Further material addressing previously identified issues will be added in subsequent drafts. 4

Table 1 – Revision History

Date	Revision Number	Author(s)	Summary of Changes
88-12-07	0.01	Barker, Connors	Initial Draft
89-03-17	0.02	Connors, Kimmel	First revision, based on feedback received during January 1989 POSIX 1003 meetings. Added front matter.
89-08-03	0.03	Kimmel	Updated for use by other TCOS-SS working groups. Content specific to the conversion of 1003.1 has been removed.
90-07-05	1.0	Rabin, Walli	Reorganized, revised, and enlarged. The datatype model was replaced and adapted from JTC1/SC22/WG11 N162 Guidelines on Language-Independent Datatypes.
90-10-15	2.0	Rabin, Walli	Revised, based on feedback received during the July, 1990 POSIX meetings.
91-01-04	3.0	Rabin, Walli	Minor revisions, including some alignment of terminology with WG11.
91-05-10	4.0	Rabin	Revised, based on experience developing 1003.1LIS/D1.0.

Please direct written comments to the author using the following address:

Paul Rabin
Open Software Foundation
11 Cambridge Center
Cambridge, Massachusetts
USA 02142

rabin@osf.org or uunet!osf.org!rabin
FAX: +1 617 225 2782
TEL: +1 617 621 8873

Programming Language Independent Specification Methods

Section 1: Background

> "Since in principle there is no reason why a particular system facility should not be used by a program, regardless of the language in which it is written, it is the practice of system facility specifiers to define an 'abstract' functional interface that is language independent. In this way, the concepts in a particular system facility may be refined by experts in that area without regard for language peculiarities. An internally coherent view of a particular system facility is defined, relating system functions to each other in a consistent way and relating the system functions to other layers within the system facility, including protocols for communication with other objects in the total system."[1)]

Language-independent specifications are components of an architectural approach to the specification of open systems. This approach seeks to promote the free interoperation and substitution of system components by providing rigorous and public specifications of functional interfaces.

The ISO, together with its committees and subcommittees, defines the highest precedence standards for open systems, and also provides guidelines for the development of standards for open systems. The ISO has mandated that language-independent specifications be provided together with language bindings for all

---

1) ISO/IEC JTC1/SC22 document N754 - Guidelines for Language Bindings, page 2.

system services. These language-independent specifications play a central role in the standards arena.

- A language-independent service specification specifies those requirements that are common to all language bindings to that service, and hence provides a functional specification to which language bindings may conform.
- A language-independent service specification is a re-usable component for constructing language bindings.
- A language-independent service specification provides an abstract specification of a service, and hence facilitates more rigorous modelling of services.
- A language-independent service specification facilitates interoperability between applications using different languages to share a common service implementation.
- A language-independent service specification facilitates the specification of relationships between one service and another.

All TCOS-SS interface specifications that are forwarded to ISO/IEC JTC1/SC22/WG15 for adoption as International Standards will consist of a language-independent functional specification plus one or more language bindings.

Because a functional specification is abstract, a degree of divergence among its language bindings is permitted, even encouraged to the extent that the languages themselves differ from each other. It is precisely the purpose of a functional specification to define those aspects in which the various language bindings should conform to each other and those aspects in which they may differ. The methods described in this document aim to provide the greatest possible flexibility for language bindings.

New features may arise in a functional specification first, or as language-specific extensions in one of the language bindings. Once a new feature is incorporated in a revision of a functional specification, it will be propagated to revisions of the language bindings. Development should be coordinated to prevent unnecessary divergence of the language bindings from each other.

The specification scheme described in this document assumes that a language binding is dependent on prior specifications for both the language and the service, and has the obligation of conformance to them, rather than the reverse. In the case of TCOS-SS services, the language binding has come first, and been the source for the specifications of the language and the service. This naturally causes a conflict between the authority of the primary language binding and that of the language-independent service specification. Initially, the primary language binding will be authoritative. However, as other language bindings are developed and used, the language-independent service specification will take on a more independent role.

Section 2: Scope and Purpose

This document is addressed to members of TCOS-SS working groups developing functional specifications or language bindings to functional specifications. Its purpose is to assist and coordinate the development of functional specifications and language bindings by defining an abstract model, and providing guidelines for the use of that model in the development of new functional specifications, the derivation of a base standard from an existing language binding, and the development of new language bindings to a functional specification.

The model presented in this document is not intended to formalize all parts of an interface specification, but only those aspects that are of concern to language bindings. The semantics of the underlying services are not specifically addressed, although they may be addressed by extensions of the current methods.

The model is primarily intended for use in developing language-independent specifications for operating system and related services, and language bindings for procedural programming languages. Whether the same or other methods are suitable for other types of services and programming languages has not been investigated.

This document provides technical guidelines for the development of language-independent specifications and language bindings. This document does not provide organizational or administrative guidelines for the management of the development process.[2)]

2.1 Goals

The proposals contained in this document have the following objectives:

1. To meet requirements to provide language-independent functional specifications, and to conform to ISO guidelines for the development of these specifications and of language bindings to them.
2. To facilitate the development of language-independent functional specifications that have sufficient richness and precision to ensure common, recognizable base functionality across all language bindings for application developers.

---

2) See [ISO1], Guidelines 1-5.

3. To facilitate the development of language bindings to these functional specifications and to allow these language bindings the greatest possible flexibility to exploit the strengths of each particular language.

2.2 Non-Goals

The following related objectives are consciously excluded as being outside of the scope of the immediate requirements:

1. To ensure interoperability between programs or program modules written in different programming languages. Language-independent specifications should identify those system datatypes whose interchange is required for a minimum level of interoperability. However, the mechanism to be used to support interchange of system of other datatypes will not be specified.

   Interoperability is a difficult and poorly understood problem. At least parts of this problem seem to fall within the scope of language and networking concerns rather than operating system concerns. The appropriate division of labor needs to be explored more fully. As the solutions become better understood, and the requirements that properly belong to operating system interface specifications become better defined, these requirements will be added to the scope of this document.

   Briefly, there are three related problems under the umbrella of interoperability that seem to be most closely tied to language-independent specifications:

   - First, the system exports private datatypes to applications, mainly for use as identifiers or attributes of system objects. A distributed application needs to be able to share values of these system datatypes among its separate parts.
   - Second, applications define their own datatypes for internal use. A distributed application needs to export these datatypes to the system, when it uses system services to store or transport values of these application datatypes among its separate parts.
   - Third, applications need to use standard datatypes defined by an external authority, including characters, time values, mathematical structures, etc. Values of such standard datatypes need to be freely transportable between the application and the system, and among different parts of an application.

2. To define complete service or interface semantics through the use of formal, verifiable specification languages.

Some formalization of functional specifications is needed in order to facilitate both the derivation and the validation of language bindings. This requirement primarily concerns interface syntax and a thin layer of semantics for the datatypes passed across the interface. Beyond this, it is certainly desirable that the underlying semantics (for instance, of the objects implemented by the service) be clear, explicit, and unambiguous. For the forseeable future, functional specifications will have a "formalization gradient"; that is, the level of formalization will decrease with depth from the surface (interface). Language bindings that attempt to map to features below the surface in order to gain flexibility, will have greater difficulty the deeper they go.

3. To ensure the portability of language or binding implementations, for instance by defining an actual system interface.

   A functional specification defines an abstraction of the same application interface that the language binding defines more concretely, not a lower-level interface on top of which language support may be implemented.

4. To permit direct binding to the language-independent specification, by specifying interfaces in an implemented (or easily implementable) notation that might permit binding through embedded alien syntax or other methods.

   A functional specification defines an abstraction of an API, not an API. Each language binding will define a concrete API that uses native syntax.

The first two excluded objectives are simply beyond the scope of the current project, but do not conflict with the models or guidelines described in this document. Future work should extend the first tentative steps taken here in these areas. The second two excluded objectives do fundamentally conflict with the approach taken here, and are not possible extensions of this work.

Note that this document does not specifically address the related issue of internationalization, which shares a concern for the treatment of character datatypes. The term "language-independent" in this document always refers to programming languages and not to so-called natural languages.

Note also that this document is not a standard, and cannot be referenced by normative text within a standard. Functional specifications need to specify their own service interfaces, including datatypes and procedure calls. The model presented in this document describes the information needed to specify datatypes and procedure calls, together with sample specifications for the datatypes likely to be needed for the specification of TCOS-SS interfaces. These should be imitated rather than referenced by functional specifications. When ISO standards for Language-Independent Datatypes and Procedure Calls become available, functional specifications may reference these as appropriate.

Section 3: References

[POSIX.1] IS 9945-1: 1990
IEEE Std 1003.1-1990
Portable Operating System Interface for Computer Environments

[PSSG] POSIX Standards Style Guide
Draft 4

[SO1] ISO/IEC JTC1/SC22 N754
Work Item JTC1.22.14
Proposed DTR 10182 on: Information Processing Systems - Guidelines for Language Bindings

[SO2] ISO/IEC JTC1/SC22 N842
Work Item JTC1.22.17
ANSI X3T2/90-087
Common Language-Independent Datatypes: Working Draft #4

[SO3] ISO/IEC JTC1/SC22/WG11 N188
Work Item JTC1.22.16
ANSI X3T2/90-074
Common Language-Independent Procedure Calling Mechanism: Working Draft Version 2.1

[SO4] ISO/IEC JTC1/SC22/WG11 N194R
Language-Independent Standards: Second Draft

[SO5] ISO/IEC JTC1/SC21 N4927
Information Processing Systems - Open Systems Interconnection - Remote Procedure Call

[ECMA] ECMA-127 Remote Procedure Call Using OSI

[PCTE] ECMA-xxx Portable Common Tool Environment
Draft 6 - October, 1989
= ISO/IEC JTC1/SC22/WG15 N70

[OM] OSI Object Management API Specification
Version 2.0 Draft 5
X.400 API Association and X/Open Company Limited

[ASN.1] ISO 8824: 1987
OSI Abstract Syntax Notation One

Section 4: Definitions

[Some or all of these definitions should appear in each language-independent functional specification.]

Abstract Datatype: A datatype defined by its properties, rather than by the representation of its values.

Base Datatype: A datatype whose definition is used in the definition of a derived datatype, especially a datatype whose values are, or are components of, the values of a constructed datatype.

Concrete Datatype: A datatype directly supported by a programming language.

Constructed Datatype: A datatype defined by a standard construction method from previously defined or primitive datatypes.

Datatype: A collection of distinguished values, together with a collection of characterizing operations on those values.

Datatype Family: A set of related datatypes sharing a common, parameterized datatype definition.

Derived Datatype: A datatype that is defined in terms of one or more previously defined base datatypes.

Functional Specification: A specification, using a language-independent notation, of a service, or set of services, that are available to applications written in several programming languages.

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

Language Standard: A specification of the syntax and semantics of applications written in a particular programming language.

Procedure: A program abstraction with formal input and output parameters that are bound to objects or values in the application that invokes it.

Overloading: The encoding of multiple service features in a single interface feature.

Status Datatype: An abstract datatype whose values may be bound to "control" values as well as "data" values.

Value: A member of the value space of a datatype.

Value Space: The set of values associated with a datatype.

Section 5: The Model

The model described in this section provides a conceptual foundation for the terms and notation used to specify procedural interfaces in a language-independent manner. Because the model will be used to specify a variety of interfaces that will in turn have language bindings from a variety of programming languages, it is subject to seemingly conflicting requirements. The model must be rich enough to allow the specification of all service interfaces of interest, and also weak enough so that it can be easily mapped onto the facilities provided by a wide range of programming languages. Of course, in any particular case, only a limited binding may be possible between a service interface and a programming language. The purpose of the model is to facilitate language bindings to the extent that they are possible, and to provide a tool that may be used during the definition of new services and languages to anticipate the effect of various choices on future language bindings.

The model includes those interface features that are intended to be shared by all language bindings to a language-independent specification, and excludes those features that may differ. The features included in the model are abstract datatypes and abstract procedure interfaces. All other features of language bindings, such as memory allocation, parameter passing, exception handling, implementation of datatypes, choice of identifiers, and packaging and visibility control are excluded from the model.

The model does not include elements whose values can be altered by assignment. The definition of objects, and of associated allocation and reference mechanisms are features delegated to the language binding.

The model does not include mechanisms for grouping interface elements and controlling their visibility to applications. Such mechanisms may be defined by particular language bindings.

5.1 Using the Model

The model is intended to provide an conceptual framework in which language-independent functional specifications can be defined. In this framework, interface features are abstract, that is, they are defined in terms of their properties and behavior rather than their implementation. Each language binding binds these abstract features to concrete features that are expressible in the language, and hence implementable by the language implementation. If the language itself supports deferred binding, some interface features may remain abstract in the language binding. Such features will be finally bound to concrete features at

compile time, or even at run time.

Interoperability requires the ability to transfer values between program modules written in different programming languages. This seems to be precluded by having abstract datatypes separately bound by each language binding. Instead, the binding must occur in the functional specification itself, or in the implementation. Specifying concrete datatypes in the functional specification defeats the purpose of allowing maximum autonomy to each language binding, yet many programming languages cannot defer value and datatype binding to the implementation.
The model is intended to be used to specify the interfaces to services, but not to specify the semantics of these services. The model does not impose a means for specifying the semantics of the underlying objects accessed via the abstract interfaces defined by a language-independent specification. Various informal or formal methods may be used. Developers of new functional specifications are encouraged to approach the problem of specifying semantics with the same techniques of functional and datatype abstraction on which the interface model is based.

To what extent is it possible to specify a service interface without specifying the underlying object model implemented by the service? Should different models and notations be used to specify interface syntax and semantics?

5.2 Execution Sequence and Concurrency

The model specifies abstract procedural interfaces. Unless otherwise specified, a procedure call transfers a single thread of control from the application to the implementation and back to the application. In certain interfaces, the thread of control does not return to the application, or returns together with a new thread of control. In other cases, the thread of control passes from the implementation to the application and then back to the implementation.

In certain cases, there may be restrictions on the order in which operations may be performed sequentially on certain objects. In certain cases, operations by multiple threads of execution on the same object are guaranteed to be serialized. In other cases, the application is responsible for ensuring serialization. Since the model only concerns the interface itself, it places no restrictions on the concurrency behavior above or below the interface. Hence, all requirements relating to execution sequence or concurrency must be specified explicitly in each functional specification.

This is an area where the model should be extended. Programming languages may have difficulty binding to certain control structures, so functional specifications should be as explicit as possible about their requirements in this regard.

5.3 Datatypes

The datatype model presented here is adapted from the model presented in JTC1/SC22 N842 Common Language-Independent Datatypes Working Draft #4 [SO2]. Features were omitted that were not needed for developing language-independent specifications of TCOS-SS interfaces. That document should be referenced for further explanation of the model presented here, and for guidance in extending this model.

The datatype model is intended to include all datatypes needed to specify TCOS-SS interfaces, and to support the range of abstractions needed to permit flexible language binding, while preserving the essential characteristics of each datatype.

For the purpose of this model, a datatype is considered to be a set of values together with a set of operations on those values. The identity of values of a datatype is dependent on the operations associated with the datatype. The identities of the operations themselves is assumed to be unproblematic. All values of a datatype must be the result of some set of operations that do not themselves depend on any value. For any set of operations, identical values yield identical results.

The set of all possible abstract datatypes is itself a complex datatype (fortunately outside the scope of this datatype model!), with mappings between the value spaces and operations of related datatypes. There are many possible ways to describe these relationships, and hence to categorize and specify particular datatypes. The current approach is partly analytical and partly descriptive. A variety of datatype attributes are defined in order to permit flexible specification of datatypes.

Because the model does not include datatype-valued expressions, a simple approach to datatype identity is possible: Datatypes with distinct names are distinct datatypes, regardless of their definitions (or implementations). Each value is considered to be associated with a unique datatype. In certain cases the natural mapping between the value spaces of two or more related datatypes is sufficiently obvious that references to corresponding values may be substituted for each other without harm.

5.3.1 Value Spaces

The value space of an abstract datatype has characteristics that are independent of the relationships between values. These include:

- Atomicity: A datatype is atomic if its values have no visible components that can be operated on independently. An atomic datatype value appears simple, regardless of its possible implementation. Non-atomic datatypes are constructed from atomic datatypes by standard construction operations.

- Nameability: Completely opaque datatypes have no names for any of their values. Other datatypes may have names for some or all values. Names may be defined by external standards or conventions, or may be declared in a functional specification.

The following fundamental datatype construction methods are available:

- Array: An array datatype is derived from two datatypes: a base datatype and an index datatype. A value of an array datatype is a mapping from the index datatype to the base datatype.
- Choice: A choice datatype is derived from one or more base datatypes. A value of a choice datatype is a set of mappings from names to base datatypes, exactly one of which is valid.
- Record: A record datatype is derived from one or more base datatypes. A value of a record datatype is a set of mappings from names to base datatypes, all of which are valid.

Identity of values of constructed datatypes is defined as identity of corresponding component values, recursively applied. For choice datatypes, corresponding component values must be either both defined or both undefined.

5.3.5 Common Datatypes

The model defines a set of commonly used datatypes and datatype families, both simple and constructed. The further information needed to define a particular member of a datatype family, including base datatypes or numeric parameters, is described in each case. The syntax associated with each of these datatypes is defined in Section 6 - Conventions.

The following list is provisional. A rationale needs to be provided for the particular selection of basic datatypes. The definitions for the basic datatypes should be structured according to the previously defined concepts. The character and time datatypes, in particular, may require additional concepts.

5.3.5.1 Boolean

A datatype with two values, named "TRUE" and "FALSE", and a set of logical operations: NOT, OR, AND, XOR, etc.

Parameters: None

5.3.5.2 Bit

A datatype with two values, named "0" ("ZERO") and "1" ("ONE"), and a set of bitwise operations: NOT, OR, AND, XOR, etc. The values are ordered, with 0<1.

Parameters: None

5.3.5.3 Opaque

This is a family of datatypes, with no order or other operations defined. An opaque datatype may have associated names that identify distinguished values.

Parameters: Number of distinct values

5.3.5.4 State

This is a family of unordered, finite datatypes. Each value is identified by an associated name.

Parameters: List of names

The name of this datatype family was changed from "Enumerated" to "State" for alignment with [ISO2]. Previous drafts raised the issue of whether ordered or unordered datatypes (or both) should be supported. This draft includes the (unordered) State datatype family, but not the (ordered) Enumerated datatype family, because no requirement for an ordered type with defined names has been found in [POSIX.1].

5.3.5.5 Ordinal

This is a family of discrete, ordered datatypes, each of which is a bounded subrange of the (ideal) infinite ordinal datatype. Ordinal datatypes are pure order datatypes. Normally, the first value of an ordinal datatype has the name "1", and successive values have the same names as the successive integers.

Parameters: Number of values

5.3.5.6 Integer

This is a family of ordered datatypes, each of which is a bounded subrange of the (ideal) infinite integer datatype. Each integer datatype inherits the operations of the base datatype: add, subtract, multiply, divide.

Parameters: Lower bound, Upper bound

Although arbitrary subranges are possible, in practice particular datatypes are specified as signed or unsigned, and by a number of values equal to 2^n. Integer datatypes that map efficiently onto hardware-supported datatypes are the most commonly used. In most cases the range includes zero as midpoint or endpoint. A recent exception is the datatype ssize_t (defined in [POSIX.1]), which has the range [-1, SSIZE_MAX].

5.3.5.7 Array

This is a family of datatypes constructed from a base datatype and an index datatype. The base datatype may be any datatype. The index datatype must be a finite, ordered datatype. A value of an array datatype contains a value of the base datatype corresponding to each value of the index datatype.

Parameters: Base datatype, Index datatype

5.3.5.8 Choice

This is a family of datatypes constructed from a sequence of base datatypes, each associated with a name. A value of a choice datatype contains, for exactly one name, a value of the corresponding base datatype.

Parameters: List of <name, datatype> pairs

5.3.5.9 Record

This is a family of datatypes constructed from a sequence of base datatypes, each associated with a name. A value of a record datatype contains, for each name, a value of the corresponding base datatypes.

Parameters: List of <name, datatype> pairs

5.3.5.10 List

This is a family of datatypes constructed from a base datatype. A value of a list datatype contains a sequence of zero or more values of the base datatype. Applying an operation to all members of a list may be supported through either of two datatypes of programming paradigm. In the first, the sequencing control is provided by the application. In the second, it is provided by the implementation.

Parameters: Base datatype, optional maximum length

Is there a need for a datatype that supports concatenation like a list, but supports indexing like an array?

5.3.5.11 Set

This is a family of datatypes constructed from a base datatype. A value of a set datatype contains an unordered collection of values of the base datatype, each occurring at most once. The is_member operation returns a boolean value that depends on whether the specified value is a member of the set. Applying an operation to all members of a set may be supported through either of two datatypes of programming paradigm. In the first, the sequencing control is provided by the application. In the second, it is provided by the implementation.

Parameters: Base datatype

5.3.5.12 Character

This is a family of datatypes whose values are either "glyphs" or special non-printing control characters. All character datatypes used in TCOS-SS functional specifications include the set of Portable Filename Characters, defined in POSIX.1.

Character datatypes are considered unordered; all collation properties are external to the model.

Parameters: Character Set

A character set can be specified by designating an externally specified character set, or by enumeration, or by a combination of these.

Character datatypes depend only on the character set, and not on the encoding of the character set.

5.3.5.13 Character String

This is a family of datatypes constructed from a base character datatype. A value of a character string datatype is a sequence of values of the base character datatype.

Parameters: Base character datatype, optional maximum length

5.3.5.14 Time

This is a family of scaled, one-dimensional vector datatypes used to measure time intervals. Time units are specified by designating an externally specified time unit (e.g., "second", "day"). Time scales are specified as multiples or fractions of the time unit.

Parameters: Unit, Scale, Maximum value

5.4 Value Names

The model includes symbolic names that are used to represent specific values of specific datatypes. The model does not specify when a name is bound to its value: at compile time, link time, or load time. The use of a value name as a parameter in a datatype specification does not imply that its value is defined at compile time.

5.5 Procedures

Procedures are modelled as abstract interfaces. Unlike most real programming languages, the model allows input and output parameters of any datatype. All parameters are passed by value. There are no in-out parameters, or parameters passed by name or by reference. Information communicated between a caller and a service is contained wholly within the procedure parameters (except for the status value). There is no use of "global variables" in the model. Of course, real language bindings may choose other means for passing values across the interface.

Procedures do not have a distinguished return parameter. All returned values are modelled as output parameters.

Procedures also return an abstract status value. In a language binding, the status value may be mapped in various ways, including to an output parameter, a global status variable, or an exception mechanism. Status values are also typed, so that each procedure can be associated with a particular status value space. All status datatypes will have a distinguished value indicating successful completion, and other values indicating specific reasons for failure. Output parameters are valid if the status value indicates success, and are invalid otherwise. There is no provision for informational status. If the status value is not the unique value

indicating success, the procedure is assumed to have failed with no side effects, unless specified otherwise.

This procedure model follows the success/failure procedure model used in [POSIX.1]. On success, all output parameters have defined values and all expected side effects occur. On failure, no output parameters have defined values and no side effects occur, except that a diagnostic status value is made available to the application. This status value is not considered an output parameter because language bindings might map the status value to control features such as exception handlers, as well as to data features. How the success or failure of each procedure is indicated to applications is defined by each language binding.

Members of WG11 have suggested a more general model that does not distinguish between success and failure, and allows arbitrary subsets of the output parameters to have defined values for each status.

The interface model includes procedures exported by the application as well as those exported by the service implementation. Examples of the former from the C language include "main()" and signal handlers.

Section 6: Conventions

[This Conventions section will be included in every language-independent functional specification.

The POSIX Standards Style Guide [PSSG] describes macros that can be used to format the notation defined below.]

This subclause defines the notation used within this standard to specify the information that is communicated through interfaces to the defined services. The notation assumes a model of an abstract procedural interface, although language bindings to this standard may take other forms.

The key elements of the notation are abstract data datatype definitions, value name definitions, and procedure definitions.

It is expected that in each case the notations specified here will be supplemented by additional descriptions using other notational schemes. For instance, for the specification of each procedure, the notation described below is to be used in the **Synopsis** section, while the semantics are described in the **Description** section.

6.1 Notation

A modified BNF notation is used in this section. The characters ":", "=", "<", ">", "|", "{", and "}" are always part of this notation and are never used in terminal strings.

Non-terminal elements consist of a "<" character followed by an identifier followed by a ">" character. Each production consists of a non-terminal followed by the characters "::=" followed by a right-hand-side expression. In a right-hand-side expression, the "|" character separates alternate elements, and the "{" and "}" characters delimit optional elements.

6.2 Identifiers

<identifier> ::= <initial-char> <identifier-body>
<initial-char> ::= <upper-case-letter> | <lower-case-letter>
<identifier-body> ::= <body-char> { <identifier-body> }
<body-char> ::= <upper-case-letter> | <lower-case-letter> |
<digit> | <separator>
<lower-case-letter> ::= a | b | c | d | e | f | g | h | i |
j | k | l | m | n | o | p | q | r | s | t |
u | v | w | x | y | z
<upper-case-letter> ::= A | B | C | D | E | F | G | H | I |
J | K | L | M | N | O | P | Q | R | S | T |
U | V | W | X | Y | Z
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
<separator> ::= - | _

The following typographic conventions are used as a mnemonic aid to indicate the syntactic datatype of identifiers.

1. Lowercase identifiers in the Helvetica Italic font are used for datatype and procedure names. Datatype names are distinguished by the suffix "_type".

Examples: *directory_handle_type,open_directory*

2. Roman uppercase identifiers are used for value names.

Examples: TRUE, MAX_DIRECTORY_HANDLE

3. Lowercase identifiers in the New Century Schoolbook Italic font are used for procedure parameters.

Examples: *directory_name*

6.3 Datatypes

The following notation is used to specify a datatype synopsis:

Datatype: *<datatype-name>*
Definition: <datatype-definition>
Description:
where

<datatype-name> ::= <identifier>
<datatype-definition> ::= <atomic-type> | <constructed-type>
<atomic-type> ::= boolean_type/<opaque-type>
<ordinal-type> | <integer-type>
<opaque-type> ::= opaque[<size>
<state-type> ::= state[<state-list>
<state-list> ::= <state-value> { , <state-value> }
<state-value> ::= <identifier>
<ordinal-type> ::= ordinal_type[<size>
<size> ::= <number>
<integer-type> ::= integer_type[<minimum>
<minimum> ::= <signed-number>
<maximum> ::= <signed-number>

<constructed-type> ::= <array-type> | <choice-type> | <record-type> 4
<array-type> ::= array*<index-type>*of
<index-type> ::= <ordinal-type> | <integer-type> | <state-type>
<base-type> ::= <type>
<choice-type> ::= choice[<member-list> 4
<record-type> ::= record[<member-list>
<member-list> ::= <member-definition> { , <member-definition> }
<member-definition> ::= <member-name> : <member-type>
<member-name> ::= <identifier>
<member-type> ::= <type>

This notation scheme is incomplete. The notations for specifying character datatypes, time datatypes, and for specifying a datatype "from scratch" have not yet been formulated. Suggestions are welcome.

6.4 Value Names

The following notation is used to specify a value name synopsis: 4

Name: <value-name> 4
Definition: <datatype> 4
Description: 4

where

<value-name> ::= <identifier>
<datatype> ::= <datatype-name> | <datatype-definition>

6.5 Procedures

The following notation is used to specify a procedure:

Procedure: <procedure-name>
Status Type: <datatype>
Input: <parameter-name> : <datatype>
<parameter-name> : <datatype>
Output: <parameter-name> : <datatype>
<parameter-name> : <datatype>
Description:
where

<procedure-name> ::= <identifier>
<datatype> ::= <datatype-name> | <datatype-definition>
<parameter-name> ::= <identifier>

<datatype-name> ::= <identifier>
<datatype-definition> ::= <atomic-type> | <constructed-type>
<atomic-type> ::= boolean_type/<opaque-type>
<ordinal-type> | <integer-type>
<opaque-type> ::= opaque[<size>
<state-type> ::= state[<state-list>
<state-list> ::= <state-value> { , <state-value> }
<state-value> ::= <identifier>
<ordinal-type> ::= ordinal_type[<size>
<size> ::= <number>
<integer-type> ::= integer_type[<minimum>
<minimum> ::= <signed-number>
<maximum> ::= <signed-number>

<constructed-type> ::= <array-type> | <choice-type> | <record-type> 4
<array-type> ::= array<*index-type*>of
<index-type> ::= <ordinal-type> | <integer-type> | <state-type>
<base-type> ::= <type>
<choice-type> ::= choice[<member-list> 4
<record-type> ::= record[<member-list>
<member-list> ::= <member-definition> { , <member-definition> }
<member-definition> ::= <member-name> : <member-type>
<member-name> ::= <identifier>
<member-type> ::= <type>

This notation scheme is incomplete. The notations for specifying character datatypes, time datatypes, and for specifying a datatype "from scratch" have not yet been formulated. Suggestions are welcome.

6.4 Value Names

The following notation is used to specify a value name synopsis: 4

Name: <value-name> 4
Definition: <datatype> 4
Description: 4

where

<value-name> ::= <identifier>
<datatype> ::= <datatype-name> | <datatype-definition>

6.5 Procedures

The following notation is used to specify a procedure:

Procedure: <procedure-name>
Status Type: <datatype>
Input: <parameter-name> : <datatype>
<parameter-name> : <datatype>
Output: <parameter-name> : <datatype>
<parameter-name> : <datatype>
Description:
where

<procedure-name> ::= <identifier>
<datatype> ::= <datatype-name> | <datatype-definition>
<parameter-name> ::= <identifier>

Section 7: Conformance

[The following clauses should be included in every language-independent functional specification.]

Editor's Note: This section will define the general requirements for conformance of a language binding to a language-independent functional specification.

Conformance depends on properties of the mapping between the datatypes, value names, and procedures of the language binding to those of the language-independent specification. For instance, the mapping must preserve equality and other relations between each datatype's values. How should these properties be specified?

Possible conformance levels include:

- *direct conformance*
- *partial conformance*
- *conformance with extensions*

Section 8: Guidelines for Language-Independent Specifications

This chapter provides guidelines for developing a new abstract interface specification or deriving one from an existing language binding.

8.1 General Guidelines

8.1.1 Document Organization

In developing a language-independent standard from a language-specific standard, it may happen that the changes made in the translation process suggest improvements to the document organization of the language-independent standard. The benefits of such changes need to be balanced against the value for purposes of review of maintaining a close correspondence between the sections of the language-independent standard and those of the language-specific standard from which it was derived.

8.1.2 Terminology

The development of a language-independent standard from existing language-specific system service specifications involves the articulation of a new, abstract interface based on coherent data and procedure models. The entire specification must be reviewed carefully to determine which requirements are language-specific and which are language-independent, and to invent new points of articulation between them where necessary. Thus, the development process imposes the requirement and provides the opportunity for a greater rigor in the use of terminology.

It must be ensured that all normative terms and phrases are defined, and used consistently, and that any use of undefined terms does not lead to normative ambiguity.

Clear and consistent terminology should be used when referring to system objects and their attributes, to associations between objects, and to internal procedures that are invoked directly or indirectly.

8.1.3 Documentation Requirements

The translation to language-independent form introduces the new term "language-binding defined", analogous to "implementation defined", for those features and behaviors that are not specified in a language-independent standard but are required to be specified in each language binding.

The terms "unspecified" and "undefined" also shift in meaning, since any feature or behavior that is unspecified or undefined in a language-independent standard may be specified in a language binding, or specified to be implementation defined.

As a result of these changes in the framework of documentation requirements, all uses of these terms must be reconsidered in the translation process. Some features and behaviors that are specified in a language-specific standard become language-binding specified in a language-independent standard. Some features and behaviors that are implementation-defined in a language-specific standard become unspecified in a language-independent standard.

8.1.4 Language-Specific Features

Interfaces should be generalized, removing restrictions imposed by a particular language binding. Language-independent interface specifications should abstract from the implementation to the purpose of the interface.

- Features that are part of a specific language binding, but for which the primary specification is the language standard, should not be included in the language-independent specification.

- Features that depend on, or are made necessary by features of the specific language, should not be included in the language-independent specification.

- Features that are included in a specific language binding for historical reasons only, or are deprecated, should not be included in the language-independent specification.

Features that originate in a specific language binding but are supportable and useful to other language bindings should be included in the language-independent specification, though perhaps in a more general form.

8.1.5 Atomicity

In general, the language-independent specification should describe atomic features that will not be further subdivided by language bindings. Features should be combined only where it is desirable for all language bindings to group features the same way.

8.2 Identifiers

Since the language-independent service specification is intended for human readability, identifiers should be descriptive without being cumbersome. Identifiers should adhere to a consistent style, and use typographic and other conventions to distinguish syntactic categories, such as procedures, parameters, datatypes, and value names. Commonly used abbreviations, such as "id" for "identifier", are acceptable if used consistently.

It is useful for language-independent specifications to recommend for language bindings any natural groupings of identifiers that should be capable of being independently made visible by applications.

8.3 Datatypes

Select a datatype that is as abstract as possible, yet provides the required operations.

New datatypes should be specified by recursively specifying constructed datatypes. Eventually, the construction will rest on one or more simple datatypes. Each constructed datatype should be specified by its method of construction and the datatypes of its component values. Each simple datatype that is a member of a datatype family should be specified by its distinguishing attributes within that datatype family.

When specifying opaque datatypes, any names representing values of that datatype should be specified with it.

For each datatype, the standard should specify whether the language binding should fully specify the corresponding concrete datatype, or instead defer this to the implementation.

8.3.1 Opaque Datatypes

Use opaque datatypes to indicate that values have no visible semantics. Be sure to specify the required number of different values that must be supported by an opaque datatype.

Use opaque datatypes for attributes of, and handles for, externally defined objects.

8.3.2 Named Datatypes

For each state datatype used, the functional specification should specify whether the datatype is extensible.

8.3.3 Order Datatypes

Use order datatypes for sequencing and array indexing.

8.3.4 Numeric Datatypes

Use numeric datatypes only for counting and measuring. Be sure to specify the required range of values.

8.3.5 Special Datatypes

8.3.6 Derived Datatypes

8.3.6.1 Abstracted Datatypes

8.3.6.2 Restricted Datatypes

If a parameter is a restricted datatype, it may be treated as if it were a less restricted version of the base datatype, with an exception status if the actual parameter is outside of the declared datatype. This is permitted because in general, language bindings cannot be expected to perform range checking for restricted datatypes.

8.3.7 Constructed Datatypes

Constructed datatypes should be used only where a specific data representation is required, and where the datatype values have only those properties implied by the construction method. Aggregate datatypes may prove restrictive if access control mechanisms do not affect all fields equally.

For each union or record datatype used, the functional specification should specify whether the datatype is closed or extensible.

8.3.8 Handling 'flag words'

If a group of flags represents an object state, this should be specified as a record of boolean_type values. If a group of flags specifies command options, this should be specified as separate boolean_type values.

8.3.9 Handling Sets and Lists

Concentrate on defining the datatypes. Don't constrain bindings to one set of operations: get_next_element vs callback_for_each_element. A choice that is natural for one language binding may not be so for another.

8.4 Value Names

Most value names fall into one of the following categories:

- configuration parameters: boolean and integer values
- state datatype values
- opaque datatype values

Names that represent values of state datatypes or opaque datatypes should be specified as part of the definition of the datatype. Unlike these, the meaning of configuration parameters is not closely associated with the datatypes of their values, so these should be specified separately.

Where a functional specification uses an abstract datatype to represent a concrete datatype in a language binding, literal value names should be represented by corresponding symbolic value names.

Functional specifications should indicate when names are bound to their values.

An important use of value names is to announce configuration options. Guidelines should be provided for this.

8.5 Procedures

8.5.1 How Big are Atomic Procedures?

When a language-independent specification is developed to support two or more existing language bindings, identification of the appropriate atomic procedures may be difficult.

The procedures in language bindings must be specifiable in terms of combinations of procedures in the associated language-independent standard. Hence, the atomic procedures selected in a language-independent standard should be no larger than the functional intersections of the procedures of all language bindings that must be supported.

On the other hand, a language binding is incomplete to the extent that all possible combinations of the atomic procedures in the associated language-independent standard are not supported. Hence, the atomic procedures selected in a language-independent standard should be no smaller than the corresponding procedures of all language bindings that need to be supported.

In case of conflict between these rules, the former takes precedence, since it is more important for a language binding to be well-defined than for it to be complete.

8.5.2 Atomic Set-and-Return-Previous-Value Procedures

For procedures that get and set service or service object attributes, two approaches to defining atomic procedures are possible. One is to have a single procedure that sets a new value of the attribute and returns the previous value. The other is to have two separate procedures: one to set a new value of the attribute, and one to get the current value of the attribute. In the first aproach it is not possible to atomically get the current value of the attribute without temporarily modifying that value in a way that might be externally visible.. In the second approach it is not possible to atomically set a new value and get the current value. Each approach provides useful atomic procedures for a different program design. A compromise approach would define two procedures: one to get the current value, and one to both set a new value and return the current value.

8.5.3 Atomic Compound Procedures

A procedure might atomically perform some subset of a group of actions, with the subset depending on the value of some parameters. In this case, breaking the procedure up into elementary procedures to undo overloading would change the semantics, since atomic combined actions would no longer be supported.

The choice of atomic operations affects the object model. If the a group of attributes can only be set together, then they constitute a single attribute.

8.5.4 Avoid Overloaded Procedures

Break up parameters that encode multiple values. Break up procedures whose parameters use different value subranges with different meanings, or which use different values to indicate different functions. Care must be taken, however, to avoid the introduction of race conditions. Overloaded procedures that use different parameter values, or the presence or absence of a parameter value, to indicate alternate underlying procedures, may be safely broken up to expose the underlying procedures. However, procedures that use overloading to indicate a selection of underlying procedures that are to be performed atomically, cannot be broken up without introducing race conditions.

8.5.5 No Side Effects on Procedure Failure

Any exceptions to the general rule that procedures that fail shall have no side effects should be carefully documented.

The specification of side effects on failure, or valid output parameters on failure, should be avoided. Whenever an error condition would result in a determinate side effect or a determinate output parameter value, this condition should be

• Cardinality: A datatype may have a finite or infinite number of different values. Infinite datatypes are seldom used directly in abstract interface specifications, but are useful for the definition of other datatypes. The cardinality of finite datatypes may be fixed in a functional specification or deferred to the language binding or implementation.

5.3.2 Properties and Operations

Other characteristics of a datatype depend on relationships among members of the value space of the datatype. Completely opaque datatypes have no visible relationships among their values. The value spaces of non-opaque datatypes have a visible structure that is defined by functions associated with the datatype.

Since the functions associated with a datatype can be composed to yield new functions, the complete set of functions defined on the value space of a datatype may be quite large. It is usually sufficient (and preferable) to list only a small set of basic functions. Where alternate basic function sets are possible, the choice of a specific basic function set can have a significant effect on the programming style of applications and the efficiency of implementations.

Because the model includes "pure" values, and does not (currently) include objects whose value can be altered by assignment, all datatypes support operations to test for value identity, but need not support assignment of values.

• Order: The most common relation between values is order. A datatype whose values have only order relations is an order datatype. Order datatypes support an **in_order** function that takes two values of the datatype and returns a boolean value: TRUE if its arguments are in order and FALSE otherwise. If the order datatype is discrete, it will support functions that return the predecessor or successor of a value. If the order datatype has a least or greatest value, it will support functions that return them. The family of order datatypes alone is quite varied, including linear and cyclic orders. A datatype whose relations include order relations is called An ordered datatype, to distinguish it from the pure order datatypes.

• Numeric: Most order datatypes have numeric analogs that supplement the order functions with arithmetic functions. Non-standard order datatypes may require non-standard arithmetic functions. Numeric datatypes may be pure, non-dimensional quantities (scalars) that may be freely combined using arithmetic operations, or they may have one or more associated dimensions, such as length or time, that must be "conserved" by arithmetic operations. Among the numeric datatypes are integer datatypes, scaled datatypes, rational datatypes, real datatypes, complex datatypes, etc. Scaled datatypes (defined in [ISO2]) behave like integers multiplied by a constant integral or fractional scale factor. Of the numeric datatypes, only integer and scaled datatypes are likely to be used in TCOS-SS functional specifications.

• Special: This group of datatypes includes all those that support functions other than those supported by order datatypes or numeric datatypes. These functions may be defined in various ways, including axiomatically, or by enumeration of their values.

5.3.3 Base Datatypes and Datatype Derivation

Although datatypes may be defined "from scratch", by defining the value space and the operations on that value space, datatypes are more commonly defined by their relation to similar, already defined datatypes. For this purpose, a set of base datatypes is predefined in the model (see below).

Datatype derivation can be either abstract or concrete. Abstract datatype derivation is simply a specification convenience, which does not imply a similar relationship between the implementations of the base and the derived datatypes. Concrete datatype derivation, on the other hand, does imply an analogous relationship between the implementations of the base and derived datatypes, and implies support for "casting" operations to generate values of the derived datatype from those of the base datatype, and conversely.

The following fundamental datatype derivation methods are available:

- Restriction: A restricted datatype is one that is derived from another datatype by selecting a subset of its value space, but keeping the same associated functions (with suitably adjusted domain and range). Examples are subrange datatypes and constructed datatypes with invariants.

- Abstraction: An abstracted datatype is one that is derived from another datatype by keeping the same value space, but removing some of the supported functions on values. An example is the order datatype derived from the integer datatype by keeping the value space and value names but removing the arithmetic operations.

- Extension: An extended datatype is one that is derived from another datatype by the addition of new values to its value space.

- Enrichment: An enriched datatype is one that is derived from another datatype by defining additional functions on its value space.

Of course, the change of value space that occurs in the derivation of a restricted or extended datatype usually involves changes in the domain and range of the functions defined on that value space. These changes may introduce exception conditions. A function on a restricted datatype may yield a value that lies outside the value subset associated with that datatype. For example, the last value of a bounded order datatype has no successor.

How should such exception conditions be specified?

5.3.4 Constructed Datatypes

Constructed datatypes use special kinds of concrete datatype derivation to create a non-atomic value space from one or more base datatypes. Constructed datatypes support no operations except the casting operations, so their component values are separately manipulable. This makes them particularly useful as representations within a module that defines and exports new functions, and exports the datatype as an atomic datatype. 4

specified as a success condition. If necessary, additional output parameters can be defined to provide diagnostic information.

8.5.6 Procedures that 'Can't Fail'

Error conditions can be defined in three ways: by the language-independent standard, by the language binding, or by the implementation. The current distinction, between procedures that shall not fail and procedures that simply have no error conditions defined, refers to the implementation. In the former case, implementations shall not define error conditions; in the latter case implementations may define error conditions. However, language bindings may always define error conditions. Even a procedure that returns an integer might raise an exception in a language that supports dynamic typing if called with an argument of the wrong datatype. The language-independent standard should not distinguish between procedures where language-specific errors are likely to be defined and procedures where language-specific errors are not likely to be defined.

For now, we are assuming that error conditions defined by language bindings and implementations will share the same status type. This implies that Status Type should be specified for all procedures, whether or not the language-independent specification specifies any conditions in which the procedure will fail.

8.5.7 Boolean Procedures

In creating a language-independent specification, there is often a choice to be made between alternate versions of procedures whose purpose is to perform a test. In the procedure model, test procedures can be specified to return a boolean_type output parameter that depends on the result of the test, or they can be specified to succeed or fail, depending on the result of the test. In certain language bindings, these versions may require very different programming styles.

If the boolean_type version is used, it should be carefully specified which conditions result in success with an output value of FALSE and which conditions result in failure. Depending on the definitions of these conditions, the boolean_type version might require an order for the detection of error conditions that is not required by the other version.

8.5.8 Undefined vs Ignored Input Parameters

In the procedure model, all input parameters must have defined values on a procedure call. This model permits a parameter value to be ignored in certain cases, but this should be avoided in general as a form of overloading.

A different case is when the presence or absence of a value for an input parameter has a controlling effect on a procedure. This should be modelled by a boolean_type input parameter accompanying the optional input parameter. If the boolean_type parameter has the value TRUE, the associated input parameter will be used; if FALSE, the associated input parameter will be ignored. Language bindings that support optional procedure parameters, or provide a special NULL value for the

parameter datatype, can eliminate the boolean_type parameter.

8.5.9 Undefined Output Parameters

In the procedure model described in TCOS LIS, all output parameters have defined values after a successful procedure call. In a few cases in POSIX.1LIS, output parameters are undefined for successful procedures. In these cases, each such output parameter is accompanied by a new boolean parameter whose value indicates whether the accompanied parameter is defined or not.

8.5.10 Procedure Descriptions

In addition to the interface specifications captured by the model, further specifications are needed, including:

- Restrictions on the use of this interface due to access controls or required privileges.
- Restrictions on the sequential or concurrent order in which this interface may be called.
- Restrictions dependent on the internal state of the service implementation.
- The precise dependency of the values of the output parameters on the values of the input parameters and the internal state of the service implementation.
- The initial state of the service implementation, and for each possible state, the new state resulting from a successful call to the procedure.

The specification of conditions that will or may result in the unsuccessful completion of the procedure, and the status value that will be returned in each case, is part of the informal interface description, and not part of the interface definition.

Section 9: Guidelines for Language Bindings

This section provides guidelines for specifying programming language bindings to a language-independent specification.

9.1 General Guidelines

A language binding defines a mapping between a language standard and a functional specification. This mapping should be harmonious, so that correct and natural access to the service features is provided to programs that are not only correct, but have good style in the particular programming language[3)].

Features should not be duplicated. Service features should be bound to existing language features, where possible, rather than to newly invented features.

Language bindings should reference the associated functional specification and language standard whenever feasible rather than duplicating specifications contained in those standards.

It is not necessary that a language binding provide access to all features of a functional specification. In particular, features of a base standard may be omitted from a language binding to that base standard where these conflict with features of the associate language standard. The omission from a language binding of any feature of the associated functional specification should be documented in that language binding together with a rationale for the omission.

9.1.1 Identify Language-Specific Interfaces

All features of the language binding that have no equivalents in the language-independent specification must be identified as language-specific extensions. This includes both interfaces that are defined in the language standard, and interfaces that are unique to the language binding.

---

3) It is desirable that both language standards and functional specifications be accompanied by guidelines for language bindings. This will facilitate the development of new language bindings, and will encourage consistency among bindings to different functional specifications and to different languages, respectively. See [ISO1], Guideline 4, p. 18.

9.1.2 Atomicity

The language binding may combine groups of features specified in the functional specification into a single feature or interface. However, a language binding should not map multiple features onto a single feature of the functional specification. In particular, interfaces that change the state of the service implementation shall not be subdivided. It is permissible, though discouraged, to subdivide non-state-changing interfaces[4].

The mapping of abstract and opaque datatypes by datatypes supported by the language may require exposing representations, and therefore parts, of values intended to be treated as wholes. Care must be taken so that the interfaces provided in the language binding, including language-specific extensions, preserve the integrity of these values.

9.1.3 Provide Cross-References

It is preferable for language bindings to a functional specification to have the same structure and organization as the functional specification, to facilitate comparison and verification of conformance[5]. Wherever a language binding follows a different document organization, adequate indices and cross-references must be provided to allow correlation with the functional specification.

9.2 Identifiers

Each language binding must map the identifiers in the language-independent specification to identifiers in the programming language. These identifiers should conform to the conventions for identifier style in the language, as well as the formal requirements on character set, maximum length, etc.

The language binding must specify any effects of application constructs on the visibility of identifiers, and any requirements or options for the presence of such application constructs. The language binding must also specify which identifiers are reserved in each case, and the effect of application use of defined or reserved identifiers.

Since applications and language binding implementations may both introduce indentifiers into an applications identifier name-space, language bindings should provide mechanisms for identifier name-space management. Where the programming language provides the necessary facilities (and they should be encouraged to do so), a modular approach to name-space management should be taken, including support for the following features:

---

4) See [ISO1] Guideline 8, page 20.

5) See [ISO1] Guideline 48, page 37.

- Where language binding identifiers fall into functional clusters that might be separately useful to applications, mechanisms should be provided for applications to control the visibility of each cluster separately.

- Applications should be able to freely define identifiers, without interference from indentifiers imported from language bindings.

- In environments where language bindings for multiple system services are available to applications, it is desirable to provide mechanisms for applications to selectively make visible the identifiers exported by any set of language bindings, and to qualify identifiers in order to avoid conflicts between identifiers imported from more than one language binding, or between an imported and an application-defined identifier.

- In environments where some language binding implementations may export identifiers to other language binding implementations, it is desirable to provide mechanisms to ensure that identifiers defined by an application do not interfere with identifiers exported from one language binding implementation to another, and that the visibility of identifiers exported by a language binding does not entail the visibility to applications of identifiers imported by that language binding.

A less desirable approach is to share identifier name-spaces among the application and all language bindings in an environment. This entails allocating reserved name-spaces to each language binding, with the remainder available for use by applications. Aside from restricting the name-spaces available to applications (and to language binding implementations), this approach creates a coordination problem among language bindings.

9.3 Datatypes

For each datatype included in a language binding, all required operations must be supported. Additional operations may also be supported where these do not conflict with the required operations. Such additional operations must be documented as language-specific extensions. Where such extensions are of general utility, they should be incorporated in future revisions of the language-independent specification. In general, language-specific extensions should be avoided, to minimize the divergence of the different language bindings to a service.

- use language-supported datatypes where possible

- use explicit procedures for datatypes not supported by the language

- take advantage of datatype-checking mechanisms

Different instances of an abstract datatype family need not be bound to members of the same concrete datatype family. For instance, different list datatypes need not be implemented the same way.

9.4 Value Names

Value names in the abstract interface specification should be preserved in the language binding. That is, they should not be replaced by literal values, however the abstract datatype of the value name is represented in the language binding.

9.5 Interface Objects

Where a language-independent specification specifies all interfaces in terms of values, and keeps all system objects hidden below the interface, language bindings may expose certain objects at the interface. Examples in [POSIX.1] are the DIR and FILE structures.

In such cases, language bindings must include additional requirements on applications regarding the initialization of objects, the effect of using copies of objects, and restrictions on concurrent access to objects.

Language bindings that might be used in multi-threaded environments should avoid the use of static objects for parameter passing, and should specify which procedure calls will behave as if serialized. Language bindings may provide additional mechanisms (i.e., locking) for application control of serialization.

9.6 Procedures

For each procedure in the language-independent functional specification, several decisions need to be made:

- Choose the parameter passing method. Examples include: by value, by reference, by implicit reference using a global object.
- If the language supports functions, the use of the return value must be decided.
- Choose the mechanism for reporting status. Examples include: return value, global value, or exception mechanism.

Where possible, parameters should be passed by value. Many languages impose restrictions on the datatypes that can be passed by value. Nearly all languages allow only a single output parameter to be passed by value. For output parameters passed by reference, it must be decided whether the caller or the service will allocate memory to store the value. Language bindings may map a parameter passed by reference to a global object, although this is generally undesirable.

The actual parameter passing mechanism used in a language binding may introduce language-specific error conditions. For instance, if an out parameter is passed by reference, the procedure might fail if the reference supplied by the caller is invalid, or the target object is too small.

Where the interface elements present in the language binding map directly to those of the abstract service specification, it is not necessary to repeat the interface semantics. Instead, a cross-reference to the appropriate part of the abstract

services specification should be given. Where the mapping is not direct, such specification should be provided as is necessary to indicate the relation between the behavior of the interfaces in the language binding to that of the interfaces in the abstract service specification.

For each interface element in the language-independent specification, a type of mapping must be chosen.

- high-level: use existing language services or create new services at the same level as existing language services
- low-level: provide direct access to service interfaces
- no mapping: if service feature conflicts with language model

Wherever there is substantial overlap between the service features and the features of the language binding, the relation between them must be specified. If the mapping is direct, the correspondence between the identifiers used in the service specification and those used in the language binding must be given.

If the language binding does not provide a direct mapping to a service interface, additional information is needed.

- What is the relation between the states of the language binding features and the underlying service features? How are they synchronized?
- What are the restrictions on interoperability between the (high-level) language interfaces and the (low-level) service interfaces?
- How is control transferred between the high-level and the low-level interfaces?

A non-direct mapping is one that uses different datatypes or values, or depends on state information outside of the service implementation.

Annex A
(informative)
Examples

The following examples are taken from the initial draft of the language-independent version of [POSIX.1].

A.1 Directories

A.1.1 Directory Operations

A.1.1.1 Synopsis

Datatype: *directory_handle_type*
Definition: *opaque_type*[MAX_DIRECTORY_HANDLE]
Description: Directory stream. See Description.

Name: MAX_DIRECTORY_HANDLE
Datatype: *integer_type*[<*implementation defined*>]
Description: Number of possible values in *directory_handle_type*.

Procedure: *open_directory*
Status Type: *posix_status_type*
Input: *directory_name* :*pathname_type*
Output: *directory_handle* :*directory_handle_type*

Procedure: *read_a_directory_entry*
Status Type: *posix_status_type*
Input: *directory_handle* :*directory_handle_type*
Output: *directory_entry_name* :*filename_type*
end_directory_flag:*boolean_type*

Procedure: *rewind_directory*
Status Type: *posix_status_type*
Input: *directory_handle* :*directory_handle_type*

Procedure: *close_directory*
Status Type: *posix_status_type*
Input: *directory_handle* :*directory_handle_type*

A.1.1.2 Description

A value of type *directory_handle_type* represents a *directory stream*, which is a sequence of all the directory entries in a particular directory. A conforming application shall only attempt to access directory entries using a value of type *directory_handle_type* that has been returned from a successful call to the *read_a_directory_entry* procedure and shall not attempt to access directory entries after a successful call to the *close_directory* procedure on that directory stream.

Directory entries represent files; directory entries may be removed from a directory or added to a directory asynchronously to the operations described in this subclause (A.1.1). The *directory_handle_type* may be implemented using a file descriptor. In that case, applications can only open up to a total of {OPEN_MAX} files and directories; see

A successful call to any of the *overlay_process_image* procedures shall close any directory streams that are open in the calling process. The result of using a directory stream after one of the *overlay_process_image* family of procedures is undefined. After a call to the *fork_a_process* procedure either the parent or the child (but not both) can continue processing the directory stream using the *read_a_directory_entry* procedure or *rewind_directory* procedure or both. If both the

parent and child processes use these procedures, the result is undefined. Either or both processes can use the *close_directory* procedure.

The *open_directory* procedure shall open a directory stream corresponding to the directory named by *directory_name*. The directory stream shall be positioned at the first entry.

The *read_a_directory_entry* procedure shall return the directory entry at the current position in the directory stream to which *directory_handle* refers, and position the directory stream at the next entry. After the last entry in the directory has been returned, subsequent calls to *read_a_directory_entry* shall set *end_directory_flag* to **TRUE** and the value of *directory_entry_name* is undefined.

The *read_a_directory_entry* procedure shall not return an empty *directory_entry_name*. It is unspecified whether entries are returned for dot or dot-dot.

The *read_a_directory_entry* procedure may buffer several directory entries per actual read operation; the *read_a_directory_entry* procedure shall mark for update the *time_file_last_accessed* field of the directory each time the directory is actually read.

The *rewind_directory* procedure shall reset the position of the directory stream to which *directory_handle* refers to the beginning of the directory. It also shall cause the directory stream to refer to the current state of the corresponding directory, as a call to *open_directory* procedure would have done.

If a directory entry that has not been returned is removed from or added to the directory after the most recent call to the *open_directory* or *rewind_directory* procedures, whether a subsequent call to *read_a_directory_entry* procedure returns that entry is unspecified.

The *close_directory* procedure shall close the directory stream referred to by *directory_handle*. If a file descriptor is used to implement type *directory_handle_type*, that file descriptor shall be closed.

If the *directory_handle* value passed to any of these procedures does not refer to a currently open directory stream, the result is undefined.

A.1.1.3 Errors

If any of the following conditions occur, the *open_directory* procedure shall fail and post the corresponding status value:

error_access_denied
: Search permission is denied for a component of the path prefix of *directory_name*, or read permission is denied for the directory itself.

error_name_length_limit
: The length of the *directory_name* argument exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX} while {POSIX_NO_-

TRUNC) is in effect.

error_file_does_not_exist
The named directory does not exist or *directory_name* is empty.

error_is_not_a_directory
A component of *directory_name* is not a directory.

For each of the following conditions, when the condition is detected, the *open_directory* procedure shall fail and post the corresponding status value:

error_process_open_file_limit
Too many file descriptors are currently open for the process.

error_system_open_file_limit
Too many file descriptors are currently open in the system.

For each of the following conditions, when the condition is detected, the *read_a_directory_entry* procedure shall fail and post the corresponding status value:

error_invalid_file_descriptor
The *directory_handle* argument does not refer to an open directory stream.

For each of the following conditions, when the condition is detected, the *close_directory* procedure shall fail and post the the corresponding status value:

error_invalid_file_descriptor
The *directory_handle* argument does not refer to an open directory stream.

A.1.1.4 Cross-References

None.

106 ## A.2 Working Directory

107 ### A.2.1 Change Current Working Directory

108 #### A.2.1.1 Synopsis

109 **Procedure:** *change_current_working_directory*
110 Status Type: *posix_status_type*
111 Input: *target_directory_name* :*pathname_type*

112 #### A.2.1.2 Description

113 The *change_current_working_directory* procedure shall cause the named directory
114 to become the current working directory, that is, the starting point for resolutions
115 of pathnames not beginning with slash. The *target_directory_name* argument is
116 the name of the directory to change to.

117 If the *change_current_working_directory* procedure fails, the current working direc-
118 tory shall remain unchanged by this procedure.

119 #### A.2.1.3 Errors

120 If any of the following conditions occur, the *change_current_working_directory* pro-
121 cedure shall fail and post the corresponding status value:

122 *error_access_denied*
123 Search permission is denied for any component of the pathname.

124 *error_name_length_limit*
125 The *target_directory_name* argument exceeds {PATH_MAX} in length, or
126 a pathname component is longer than {NAME_MAX} while {POSIX_NO_-
127 TRUNC} is in effect.

128 *error_is_not_a_directory*
129 A component of the pathname is not a directory.

130 *error_file_does_not_exist*
131 The named directory does not exist or *target_directory_name* is empty.

132 #### A.2.1.4 Cross-References

133 *get_current_working_directory*, A.2.2.1.

A.2.2 Get Working Directory Pathname

A.2.2.1 Synopsis

Procedure: *get_current_working_directory*
Status Type: *posix_status_type*
Output: *directory_name*:*pathname_type*

A.2.2.2 Description

The *get_current_working_directory* procedure shall return an absolute pathname of the current working directory.

A.2.2.3 Errors

For each of the following conditions, if the condition is detected, the *get_current_working_directory* procedure shall fail and post the corresponding status value:

error_access_denied
Read or search permission was denied for a component of the pathname.

A.2.2.4 Cross-References

change_current_working_directory, A.2.1.1.

Identifier Index

Alphabetic Topical Index

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z