TECHNICAL REPORT

ISO/IEC TR 10182

First edition
1993-12-15

Information technology — Programming languages, their environments and system software interfaces — Guidelines for language bindings

Technologies de l'information — Langages de programmation, leurs environnements et interfaces logicielles des systèmes — Techniques d'interface pour les normes de langages de programmation

Reference number
ISO/IEC TR 10182:1993(E)

Contents

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 main task of technical committees is to prepare International Standards, but in exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/IEC TR 10182, which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Sub-Committee SC 22, *Programming languages, their environments and system software interfaces*.

This page intentionally left blank

Information technology — Programming languages, their environments and system software interfaces — Guidelines for language bindings

1 INTRODUCTION

1.1 Status of the Document

This document is a compilation of the experience and knowledge gained by the members of ISO/IEC JTC1/SC22/WG11 (Techniques for Bindings) from the generation of programmers' interfaces to FUNCTIONAL INTERFACE STANDARDS. Although current experience was derived from the fields of computer graphics and database management, the problems discussed are thought to be generally applicable for mappings of other functional interface standards to programming languages. This document is intended

a) to identify the problems and conflicts which must be resolved;
b) to suggest guidelines for future use;
c) to provide scope and direction to required additional work, such as common procedural calling mechanisms and data types; and
d) as a historical record of past experiences and decisions.

This document is incomplete; the authors have concentrated on those areas where experience and expertise was readily available. The ideas and issues brought forward here emerged from more than ten years of work, and are represented in international Standards.

Section 2 of this document contains the results of a survey of current methods used for language binding development. Characteristics of each method are given, followed by reasons for the selection of the method.

Application of the methods has suggested some guidelines that are presented in Section 3. Sections 2 and 3 contain documentation of the current state of language binding efforts; Section 4 addresses future directions for language bindings.

Circulation of this document is necessary at this stage, as input and discussion from representatives of ISO/IEC JTC1/SC21 (functional specification standards developers), ISO/IEC JTC1/SC24 (computer graphics standards developers), and ISO/IEC JTC1/SC22 (language standards developers) is urgently sought. The document in its current form may be useful for those about to embark on language binding developments.

1.2 Scope

This document is based on experience gained in the standardization of two major areas in information processing. One area covers programming languages. The other area is composed of the services necessary to an application program to achieve its goal. The services are divided into coherent groups, each referred to as a SYSTEM FACILITY, that are accessed through a FUNCTIONAL INTERFACE. The specification of a system facility, referred to as a FUNCTIONAL SPECIFICATION, defines a collection of SYSTEM FUNCTIONS, each of which carries out some well-defined service.

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 the 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.

However, if these two areas are standardized independently, it is not possible to guarantee that programs from one operating environment can be moved to another, even if the programs are written in a standard programming language and use only standard system facilities. A language binding of a system facility to a programming language provides language syntax that maps the system facility's functional interface. This allows a program written in the language to access the system functions constituting the system facility in a standard way. The purpose of a language binding is to achieve portability of a program that uses particular facilities in a particular language. Examples of system facilities that have had language bindings developed for them are GKS, NDL, and SQL (see Section 1.3, References). It is anticipated that further language binding development will be required. Some system facilities currently being standardized have no language bindings and additional system facilities will be standardized. There is a possibility of n x m language bindings, where n is the number of languages and m the number of system facilities.

The scope of this document is to classify language binding methods, reporting on particular instances in detail, and to produce suggested guidelines for future language binding standards.

Note that the language bindings and the abstract facility interfaces must have a compatible run time representation, but the abstract facility does not necessarily have to be written in the host language. For example, if the application program is using a Pascal language binding and the corresponding facility is written in FORTRAN, there must be a compatible run time representation in that operating environment. How this compatibility is achieved is outside the scope of these guidelines. This is generally a property of the operating environment defined by the implementor, and is reviewed briefly in this document.

1.3 References

ISO/IEC 1539:1991, *Information Technology - Programming languages - FORTRAN.*

ISO/IEC 1989:1985, *Programming Languages - COBOL.*

ISO/IEC 1989:1985/Amd.1:1992, *Programming Languages - COBOL - Amendment 1.*

ISO 6373:1984, *Data processing - Programming languages - Minimal BASIC.*

ISO 7185:1990, *Information technology - Programming languages - Pascal.*

ISO 7942:1985, *.Information processing systems - Computer graphics - Graphical Kernel System (GKS) functional description.*

ISO 8651-1:1988, *Information processing systems - Computer graphics - Graphical Kernel System (GKS) language bindings - Part 1: FORTRAN.*

ISO 8651-2:1988, *Information processing systems - Computer graphics - Graphical Kernel System (GKS) language bindings - Part 2: Pascal.*

ISO 8651-3:1988, *Information processing systems - Computer graphics - Graphical Kernel System (GKS) language bindings - Part 3: Ada.*

ISO 8651-4:1988, *Information technology - Computer graphics - Graphical Kernel System (GKS) language bindings - Part 4: C.*

ISO 8652:1987, *Programming Languages - Ada.*

ISO 8805:1988, *Information processing systems - Computer graphics - Graphical Kernel System for Three Dimensions (GKS-3D) functional description.*

ISO/IEC 8806-1:[1], *Information technology - Computer graphics - Graphical Kernel System for Three Dimensions (GKS-3D) language bindings - Part 1: FORTRAN.*

ISO 8907:1987, *Information processing systems - Database Languages - NDL.*

ISO/IEC 9075:1992, *Information technology - Database Languages - SQL.*

ISO/IEC 9592:1984, *Information processing systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS).*

ISO/IEC 9593-1:1990, *Information processing systems - Computer graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 1: FORTRAN.*

ISO/IEC 9593-2:[1], *Information technology - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 2: Extended Pascal.*

ISO/IEC 9593-3:1990, *Information technology - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 3: Ada.*

ISO/IEC 9593-4:1991, *Information technology - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 4: C.*

ISO/IEC 9899:1990, *Programming languages - 'C'.*

ISO/IEC 10206:1991, *Information Technology - Programming languages - Extended Pascal.*

C. Osland, *Case Study of GKS Development*, Eurographics Tutorials 1983, Springer.

J. R. Gallop, C. Osland, *Experience with Implementing GKS on a Perq and Other Computers*, Computer Graphics Forum 9(7), North-Holland 1985.

F. R. A. Hopgood, D. Duce,J. R. Gallop, D. Sutcliffe, *Introduction to the Graphical Kernel System (GKS)*, Academic Press, 1983.

M. Sparks, J. R. Gallop, *Language Bindings for Computer Graphics Standards*, IEEE Computer Graphics and Applications, Vol 6, Number 8, August 1986.

M. Sparks, *Graphics Language Bindings*, Computer Graphics Forum, Journal of the European Association for Computer Graphics, vol. 4(4) 1985.

M. Sparks, *Graphics Standards Bindings - What Are They and When Can We Use Them?*, Computer Graphics/November 1985.

1 To be published.

Language Binding Generic Issues (document within ISO/IEC JTC1 SC24/WG4 and ISO/IEC JTC1 SC22/WG11)

1.4 Terms and Abbreviations

ABSTRACT SERVICE INTERFACE: 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.

ALIEN SYNTAX: Syntax of a language other than the host language.

CGI: Computer Graphics Interface standard (ISO DP) - a functional specification of the computer graphics programming system facility.

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

EXTERNAL IDENTIFIER: An identifier that is visible outside of a program.

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

FUNCTIONAL SPECIFICATION: 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.

GKS: Graphical Kernel System standard (ISO/IEC 7942) - a functional specification of the computer graphics programming system facility.

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

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

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

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

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

LANGUAGE BINDING OF *f* TO *l* or *l* LANGUAGE BINDING OF *f*: A specification of the standard interface to facility *f* for programs written in language *l*.

LANGUAGE COMMITTEE: The ISO technical Subcommittee or Working Group responsible for the definition of a programming language standard.

MDL: A language for the specification of an interface to a generic system facility, the MDL (module definition language) is used to generate a module to support the specific system facility access needs of an application program.

NDL: Database Language NDL may be used to define the structure of a database using the network model of data.NDL is defined in IS 8907. (See Section 1.3, References). The standard also includes the data manipulation functions and their language bindings.

PHIGS: Programmers Hierarchical Interactive Graphics System standard (ISO/IEC 9593) - a functional specification of the 3-D computer graphics programming system facility.

PROCEDURAL BINDING (system facility standard procedural interface): The definition of the interface to a system facility available to users of a standard programming language through procedure calls.

PROCEDURAL INTERFACE DEFINITION LANGUAGE: A language for defining specific procedures for interfacing to a system facility as used, for example, in IS 8907 Database Language NDL.

PROCEDURE: 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 data types. Invoking a procedure transfers control to that procedure.

PROCESSOR: 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.

PROGRAMMING LANGUAGE EXTENSIONS WITH NATIVE SYNTAX or native syntax binding: 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.

SQL: Database Language SQL (Structured Query Language) defines the structure of a database using the Relational model of data. Database Language SQL is defined in IS 9075. (See Section 1.3, References). The standard also includes the data manipulation functions and their language bindings.

SYSTEM FACILITY: 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: The ISO technical subcommittee or Working Group responsible for the development of the functional specification of a system facility.

SYSTEM FUNCTION: 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.

2 OVERVIEW OF FUNCTIONAL BINDING METHODS

2.1 Introduction to Methods

This section discusses the binding development problem in general by documenting a number of different approaches to bindings. Each approach has its own characteristics from the points of view of the user, the implementor, and the specifiers of standards.

The first task in specifying a binding of a system facility is to determine the usability, stability, and implementation goals of both the binding and the system facility, and to use these to help select the best method.

The functional binding methods are:

Method 1. Provide a completely defined procedural interface (the System Facility Standard Procedural Interface)

Method 2. Provide a procedural interface definition language (User-Defined Procedural Interface)

Method 3. Provide extensions to the programming language, using native syntax

Method 4. Allow alien syntax to be embedded in the programming language

Method 5. Binding pre-existing language elements.

Before addressing the individual methods, a discussion of a general issue that affects programming language implementations is indicated. This issue is whether to increase the capability of a given compiler to encompass the system facility, or to provide a pre-processor. Though this is of no direct concern to language binding developers, they may wish to consider the feasibility of each option when choosing a method.

A pre-processor is necessary for Method 4 above, and optional for Method 3. Method 1 does not require a pre-processor but it may be useful to provide a utility that checks the syntax of all the procedure calls. The function of a pre-processor is to scan a program source text, to identify alien syntax or syntax associated with a given facility, and to replace this text by host language constructs (for example, calls to system functions) that can be compiled by the standard compiler.

The advantages of a pre-processor are:

- A pre-processor can often carry out semantic checking not provided by the language compilers.
- A pre-processor can be independent of the particular language compiler.
- A pre-processor approach avoids problems that result from tampering with an existing language standard or with certified compilers.
- If the system facility is enhanced, it is easier to modify a pre-processor than a full compiler.

The disadvantages are:

- A pre-processor requires an extra pass through the source.
- There may be a problem with multiple pre-processors for different system facilities existing in the same environment.
- A pre-processor may produce code unfamiliar to the programmer and make debugging more difficult - for example, it may change statement numbers.
- Depending on the language extensions, it may be necessary to analyze the syntax of most of the language to detect the code to be replaced.

In the following sections, each functional binding method is discussed, circumstances that suggest a method be used or avoided are given, and relevant advantages and disadvantages are defined.

There is often more than one way to implement a given method. In addition, it may be necessary to implement more than one method for any given facility.

2.2 System Facility Standard Procedural Interface (Method 1)

With functional binding Method 1, the system facility is designed to support a fixed number of procedures. Each procedure has formal parameters of defined data types and each procedure invocation passes actual parameter values which match the data types.

Method 1 is appropriate when the syntax of the interface provided for each system function is fairly simple and can be fully defined by a few parameters. The method can become unwieldy when the functions that can be invoked use a large number of data types whose structure may be unknown until the time of invocation, and require parameters or data types that are unknown in structure until the time of invocation.

It is often useful to define subsets of the facility to suit different modes of use. For example, where the functions are largely independent and a program only requires a few of them, it may be possible to reduce the size of the run-time system by omitting portions of the system facility. These subsets are reflected by levels of conformance to the functional interface standard.

Use of Method 1 requires that the procedural interface be redefined for each programming language, in terms of the syntax and data types of that language. Thus, separate language binding standards to the same functional interface standard are created.

Method 1 has been used by GKS and other graphics draft standards, where the syntax of the parameters is fairly simple.

It should be noted that, if languages used a common procedure calling mechanism and equivalent sets of data types (ISO/IEC JTC1 has assigned work items on these topics to SC22/WG11), then it would be possible to derive system facility standard procedural interfaces from the abstract definitions. It would also be possible to derive system facility standard procedural interfaces from abstract definitions under other conditions, particularly for languages of sufficient abstraction (like Pascal and Ada).

2.3 User-Defined Procedural Interfaces (Method 2)

With functional binding Method 2, the run-time procedural interface is defined by the user, and the system functions invoked by the procedures are defined in a language appropriate to the system facility.

This method is appropriate when the interfaces to the system functions provided by the system facility are too complex to be defined by a few parameters, and when they cannot be easily contained in an exhaustive list.

Method 2 allows the binding document to be easily adapted to different programming languages, since the binding only deals with data types. The naming of procedures and parameters is done by the user and not the binding specifiers. The procedural interface definitions are compiled and the resulting object module must be linked both to the application program and to the system facility.

Advantages of Method 2 are:

- It may provide early diagnosis of errors.
- It is processed once and may allow specific optimization (for example, optimization of query searches) leading to run-time economies.
- Modules may be shared among application programs, since they exist independently.
- The task of creating modules may be specialized and managed outside of the user's program.

Disadvantages of Method 2 are:

- The definition of modules is an extra design step and risks poor usability when the programmer has to define his own modules.
- The procedural interface definition language is another language to learn unless the procedural interface language is part of the host language already.
- There is generally an administrative overhead for managing modules to ensure that they get recompiled and relinked when necessary.
- Porting an application involves porting the program and all the referenced procedural interface definition language modules.
- An additional compiler has to be provided for the procedural interface language unless the procedural interface language is part of the host language already.

Database facilities use this method, where a Procedural Interface Definition Language (in the database standards this is referred to as a Module Definition Language), containing both declarations and procedural statements, is provided. A module may declare the data to be accessed as a view of the database (as it may reference a predefined view) and it defines both the form and the execution of database procedures.

2.4 Programming Language Extensions with Native Syntax (Method 3)

With functional binding Method 3, 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.

This method is viable only when the system facility is stable and when the application requirements are well understood, since the cost of changes to programming language standards is high.

The main advantage is usability. The users of the language have little extra to learn except the new facilities. It also allows the language developers, when defining new versions of the language, to choose a conforming subset of the facilities or to change the appearance of existing language facilities if they believe this is helpful to their users. Another advantage is that new data types appropriate to the system facility can be constructed.

The disadvantages are that Method 3 ties a compiler to a particular system facility definition. It also ties the language specification to that of the system facility, making it highly desirable to process the standardization of both specifications together if enhancements are needed. It may also be more difficult to use this method in a mixed-language environment, since the same facilities may have confusingly different appearances in different host languages.

Method 3 has been tried with the COBOL and FORTRAN database facilities (Codasyl and ANSI) and with the graphics chapter for Basic.

2.5 Programming Languages with Embedded Alien Syntax (Method 4)

With functional binding Method 4, the system facilities are considered to be 'driven' by statements written in a 'system facility language' rather than in the host programming language. The embedded alien syntax must be clearly distinguishable from the host language so that it can be processed by a pre-processor.

Method 4 is suitable when the system facilities are too complex to be invoked by simple procedures (as for Method 2, User-Defined Procedural Interfaces). The method could be implemented by having the pre-processor generate Module Definitions as in Method 2.

The advantage of Method 4 over Method 2 is that simple programs, particularly those that may have a short life, may be easier to create. The advantage of Method 4 over Method 3 is that the independence of host language specifications from system facility specifications is maintained, so development of each can progress more quickly.

The disadvantage of Method 4 over Method 2 is that this method substantially complicates the relationships between applications and system facilities. Although the alien syntax should be very similar for all host languages, the pre-processor will need to 'know about' the conventions of each host language to be able to generate the correct interfacing code.

The disadvantage of Method 4 compared with Method 3 is that the programmer has to know two languages and may be confused by the differences between them.

Method 4 is one of the options in the ISO database standards.

2.6 Binding Pre-Existing Language Elements (Method 5)

In some cases, the host language may contain language elements that can be directly identified with corresponding elements of the abstract system facility. For example, in a binding to a system facility that opened and closed files, the host language may already contain constructs for opening and closing files.

The advantage is that pre-existing constructs are used and no extra work in binding needs to be done. If that facility is already present in the language, then making use of that facility avoids unnecessary perturbations to the language.

Care should be taken that the language construct fully meets the requirements of the system facility.

2.7 Conclusions

The subsections above have described five different methods for developing functional bindings, and the circumstances in which they can be used. None of the methods is appropriate in all circumstances, or for all languages. In practice, a combination of methods may be appropriate. In some languages it is necessary to combine Method 4 with Method 5.

It is possible, and often desirable, for a system facility to provide more than one method of binding, to give the implementor and user a choice. However, if an implementor provides only one of the standard methods, the user has no choice, and, unless there is a recognized way of converting between methods, portability problems result.

The objective of a standard language binding is to enable a program to be portable when it is written in a standard programming language and accesses a standard system facility. Often the system facility is written in a different language from the application program and requires a certain compatibility between the implementations of the two source language compilers. Of course, similar compatibility is necessary for different compiler implementations of the same source language. In particular:

a) the procedure calling mechanisms should be compatible, and
b) corresponding data types should have compatible machine representations.

Often, but not always, the hardware and operating system will determine appropriate standards or conventions for the representation of primitive data types and inter-program calls. Where there are mismatches, it is necessary for the implementor to create a layer of software to perform conversions between alternative data type representations or procedure calling mechanisms. There are now ISO/IEC JTC1 work items addressing a) and b) above. These are: work item 22.16 - Specification for a Model for Common Language-Independent Procedure Calling Mechanisms, and work item 22.17 - Specification for a Set of Common Language-Independent Data Types.

The methods described have been used in current ISO standards for database and graphics. Some papers defining bindings for communications facilities have also been reviewed, but the strategy to be adopted for ISO OSI bindings is yet to be determined.

3 GUIDELINES

3.1 Organizational Guidelines for Preparation of Language Bindings

This section describes some organizational guidelines that should be followed in order to facilitate the generation of binding standards. A general statement of each guideline is given, followed by some discussion. The guidelines appear in no particular order.

GUIDELINE 1

Standard bindings of some form should be developed for all standard system facilities that may be accessed from a standard programming language.

Here, "standard" means that an ISO standard or draft standard exists for both the system facility and the language.

There are standards describing system facilities which do not have standard language bindings associated with them. Lack of a standard may lead to implementor-defined interfaces, causing loss of portability.

GUIDELINE 2

Either the language committee or the system facility committee should have primary responsibility for the language binding.

In this area, current practice is 'whichever committee perceives the need for a binding' or 'if the language has an external procedure call mechanism, then the system facility, otherwise the language committee'. Unfortunately, sometimes a binding is required, yet no-one takes the initiative to start binding work; some method for resolving this impasse is required.

It is expected that it is the primary responsibility of the system facility committee to establish a reference binding to an arbitrary language and a generic binding. Subsequent bindings should be the responsibility of the appropriate language committees. In practice it is expected that the system facility committee will seek support from the applicable language committee in the creation of an arbitrary binding. This would increase the likelihood of getting a full range of language bindings to make a system facility useable. The system facility committee is more likely to have an interest in making the facilities accessible; furthermore, language committees might not have the necessary expertise to develop bindings to specialized facilities. Part of the primary responsibility is to respond to public comments.

GUIDELINE 3

Whichever committee is responsible for a particular binding, the other committee needs o be consulted as early as possible. The two committees have complementary responsibilities and concerns.

A system facility committee is concerned with

- fidelity to the functional specification, including relevant level structures.
- similarity of program structure independent of the programming languages.
- suitability of the system facility for binding to the various languages.

A language committee is concerned with

- correctness of the binding definition, and adherence to good practice in the language, including the avoidance of obsolete and deprecated features.
- consistency in the binding of similar concepts in similar ways throughout different bindings (for example, 'is a 2x3 matrix bound as a (3,2) array or a (2,3) array or a (6) array?' is a question to be answered by a language committee).

- the needs of programmers accessing more than one system facility in one program.
- suitability of the language for various binding methods.

Both committees are concerned with ease of use and orthogonality of concepts.

GUIDELINE 4
Specific guidelines should be produced alongside standards for particular system facilities and particular programming languages.

- A set of guidelines for producing language bindings for a system facility should be associated with the standard functional specification of any system facility.
- A set of guidelines for producing language bindings of different system facilities to a language should be associated with the standard of any programming language.

Potentially, there is an n x m problem of agreeing and processing n language bindings to m system facilities. Ideally, the development of appropriate guidelines should reduce the problem to one of order n+m.

Binding guidelines could be published as appendices to standards, and in some cases a supplemental standard could be considered. Sample programs would also be helpful. Publication of such guidelines by language committees would help not only system facility committees, but also any producers of packages needing bindings to languages, and would thus help promote portability.

GUIDELINE 5
Draft proposals for bindings should not progress beyond the standardization stage of the system facility or the language.

In addition, no system facility should progress to DIS (Draft International Standard) or IS (International Standard) until there is at least one language binding at or above the level of DP (Draft Proposed International Standard) or DIS, respectively. Users cannot fully judge a semantic standard without seeing the specific syntax which is their only access to it. Also, there are some difficulties with abstract semantics which are only revealed by the production of a language binding. Reference ISSUE 1: What should be the criteria for determining if the functional interface standard should be bound to a particular programming language?

3.2 General Technical Guidelines

This section contains guidelines that are general over all binding methods.

GUIDELINE 6
Language bindings to the same system facility should be similar in those respects where the languages are similar.

A functional specification might have a system facility data type that is easily represented in some languages but not in others. For example, GKS uses points which can easily be represented as records in Pascal and Ada, but not in FORTRAN. It seems unreasonable for the Pascal and Ada bindings to be constrained by the FORTRAN binding. On the other hand, after language bindings for a few very dissimilar languages have been produced, bindings for additional languages might be produced by analogy.