LANGUAGE-INDEPENDENT STANDARDS

by Brian Meek, Paul Barnetson and Willem Wakker

A draft working paper for ISO/IEC JTC1/SC22/WG11 - Language Bindings

Version 2.1, September 1991

0. Introduction

This paper attempts to summarise the relationship between different language independent standards (either published or in draft). A broad classification is suggested, and although it is probably not of great importance, it may help in getting thoughts oriented, especially for those experience is mainly in particular directions, e.g. a specific kind of language-independent standard, and specific languages. Some discussion of potential further language-independent standards is included.

The motivation for writing this has been a combination of perceived demand, as expressed in frequent requests for explanations of the purpose of particular standards projects, and of perceived need, by observation of misunderstandings about the nature of particular projects. The paper represents the second draft of a WG11 document to be used in helping to explain our work to others. At its September 1990 meeting WG11 agreed that the first draft, by Brian Meek, should be updated and expanded, with contributions from other authors, and it has been suggested that it be circulated to SC22. It might eventually be published more widely to the programming language community, e.g. through Sigplan Notices. This version includes new material based on contributions from Paul Barnetson (section 2.4) and Willem Wakker (section 2.5).

In version 2.1 of this document (September 1991) an updated status overview of the projects is given in section 4.

Contact address:

Willem Wakker, Convenor SC22/WG11
ACE Associated Computer Experts bv
Van Eeghenstraat 100
1071 GL Amsterdam
The Netherlands
Phone: +31 20 6646416
Fax: +31 20 6750389
Email: willemw@ace.nl

1. A broad classification

The suggested classification is into base standards, generic standards, and functional standards. The division is rough; often standards can appear to straddle boundaries that themselves are not very well defined, and may be thought to be unnecessary, as already mentioned. Another way of looking at it, however, is as a classification of roles that language-independent standards can play. (This copes with the boundary straddling problem since such standards can then be regarded as playing more than one role in some cases.) People may think one view is better than the other, and/or that neither is needed at all.

1.1 Base standards

Base standards are language-independent standards which provide (or could provide) the raw materials from which language definition standards can be built. (Since the term "base

standard" is used in other contexts with somewhat different connotations, this may not be ideal terminology but I have not thought of anything better).

The raw materials provided, however, do not in themselves carry specifically programming language connotations (which would make them generic) and might well be used also for purposes unrelated to programming languages.

The archetypal base standards for languages are the character set standards, because they can and indeed do provide the elements from which the building blocks of languages - keywords, identifiers, literals etc - are made. This remains true even though language standards too often give too little attention to character set standards. Of course, character set standards can (or could) contribute at various levels, partly because typical character set standards operate at several different levels of abstraction but mostly because they are base standards that can be used for various purposes.

The classical example is the different roles of the character **I** in the keyword **IF**, the user-defined identifier **CHIMP**, the variable name in **I:=1** or **J:=I**, the character literal **'I'**, and the comment (suitably delimited) **BEGINNING OF SORT ROUTINE**.

Hence this view of character set standards as base standards for languages has nothing to do with character data types and character handling, which are generic language issues. It applies even if a language were to have no concept of characters as data.

Since character set standards are outside the remit of WG11 and SC22 they are not discussed further. (However, the use of these standards for building language elements is an SC22 matter - see the guidelines DTR10176.) The only WG11 standard (with its proposed offshoots) that could be regarded as a candidate base standard is the Language Compatible Arithmetic Standard (LCAS, CD10967); this is discussed further in section 2.2 below. Syntactic and semantic formal definition method standards like the BSI metalanguage standard BS6154 and VDM could also be regarded as base standards for language definitions. Both have other applications, of course, but that is typical of base standards.

1.2 Functional standards

Whereas base standards, as the term implies, address levels lower than the language level, functional standards address levels above the core language level, or at that level but outside the language domain. Typically, functional standards can extend the range of applicability of languages or provide for language users access to facilities outside the language domain. The archetypal examples are the graphics standards, GKS etc, but things like FIMS (Forms Interface Management System), also fall into this category.

One difficulty here is that languages vary considerably in the functional area that they attempt to cover, so that the same functional standards may be a pure extension in some cases but overlap in others. Again the classical example is GKS, in its bindings to languages like Fortran and Pascal on the one hand and to Basic on the other.

In such cases, and where languages have some embedded features which other languages leave to the operating systems environment (Basic, APL and Mumps are instances), it could be argued that all this means is that the language standard incorporates some features from a (possible) functional standard. A much greyer area is input-output as a whole, because it is clear that, when it is being executed, a program must have facilities to communicate with its environment. Furthermore (as the example of Algol 60 demonstrates), it is necessary for these facilities to be portable with the program, and therefore somehow to be embedded

in the language. Hence the suggested language independent standard for input-output discussed in X3T2 would appear to be generic rather than functional, despite its obviously functional characteristics. However, where does one stop? Graphics, after all, contains a substantial input-output element which nevertheless most languages and their standards do not address.

One rule of thumb might be that input-output of values of the language-defined datatypes is generic but everything else is functional. The remaining grey area would be types which the language permits users to construct in addition to those defined in the language. Another rule of thumb might therefore be that anything independent of implementation-defined input-output facilities and application-specific semantics is generic, the rest functional.

The implication, of course, is that the boundary between may not matter. Wherever the boundary is drawn, the functional standards need to be bound to the language standards that need them. Guidelines for producing such bindings form another WG11 project which is described in section 2.4.

1.3 Generic language-independent standards

The generic category covers standards which are both fundamental to programming languages and at the language level of abstraction. The archetypal examples are the Common Language Independent Datatypes (CLID) and Common Language Independent Procedure Calling (CLIP) standards currently in draft. A common language independent array handling standard, something which has been talked about as a possibility but for which no project has been started as far as I know, would also come into this category as would exception handling (though not "event handling" in general, since this would cover process control facilities and would qualify as "functional" for most languages).

2. Current projects and their relationships

2.1 Common Language Independent Datatypes (CLID)

This is a generic standard, the most fundamental of those under review. It provides a reference collection of datatypes which can be used as common ground by actual programming languages. CLID is an enabling standard, to aid the specification and development of tools and services which all languages can share. For languages to avail themselves of such CLID-based facilities, mapping standards will be needed to bind the native datatypes of each language to CLID, and vice versa. The CLID standard specifies conformity rules for such mapping standards and provides guidance in performing the association. In particular CLID specifies mappings of type to type, and of value-of-a-given-type to value-of-a-given-type, but no more. The CLID standard specifies "characterising operations" for values of each of its types but it is not a requirement that a language must support all of those operations for values of the native datatype associated with the relevant CLID standard datatype. Possibly the mapping standards themselves could add such requirements, if the community served by a particular actual language required it. Even if the mapping standards did not add such requirements, since they in turn are just enabling standards, then possibly applications standards or services might do so. Such matters are not the concern of the CLID standard.

It may then be asked, why does CLID mention operations at all? The answer is simply to help those attempting to identify mappings between actual languages and the CLID standard datatypes to recognise the most appropriate match for their purpose - the "best fit". In general, for each datatype the characterising operations listed are neither exhaustive

nor minimal. The aim has been to provide enough characterising operations to distinguish a given CLID standard datatype from others with a similar "value space", and leave users of the standard (envisaged after all as typical and experienced "language people") in no doubt as to the nature of the type intended.

It must be borne in mind throughout that the CLID standard provides a very general abstract model not aimed at any specific application. Despite some misconceptions it does not exist just to support parameter passing in common language-independent procedure calling (see below 2.3) though that is undoubtedly an important application. Similarly it does not exist just to support transmission of data between one language processor and another (though that too is important). The CLID standard very carefully and intentionally avoids any reference to representation of values of the types, or even definitions which depend on an underlying representational model. Some illustrations are given in **[1]**.

An implication of the CLID standard is that if language-independent facilities define their own datatypes then this multiplies the number of mappings that have to be defined. If an application needs more than one such facility then this number is multiplied again. Types so defined are likely to be limited to those required for the facility and may be further constrained because the facility requires a representational model to be added. This could lead to a situation where,in some applications or environments, one facility is restricted by the datatypes supported by the other.

Basing all such language-independent facilities on the CLID standard datatypes avoids the need for a multiplicity of mappings and reduces the danger of one facility being constrained by the limitations of another, at least in respect of datatypes. It is of course important to ensure that the CLID standard is sufficiently general to support all kinds of language-independent facility.

2.2 Language Compatible Arithmetic Standard (LCAS)

2.2.1 The basic LCAS

In some respects LCAS **[4, 5]** can be regarded as an extension of the CLID standard, though its aims are rather different. To get one point out of the way first, this standard should also really be designated the "Common Language Independent Arithmetic" standard; using the word "compatible" is the perpetuation of an historical accident.

The LCAS covers properties of the (CLID) numeric datatypes real and integer. The motivation is to specify properties of datatype real; integer datatype is there for completeness, so that the document is self-contained, since some of the properties of real values need to be specified in terms of the properties of integers.

The aims of LCAS differ from CLID in two main ways. One is that it addresses the properties not just of values but of operations on those values. The other is that it (necessarily) must take cognizance of representational issues, because the properties of real datatype values and operations depends on their representation. LCAS is based on a floating point representation, which is not in itself a severe restriction since that representation is so common.

The reason why real datatype requires special treatment is that in general values are represented only approximately rather than exactly, and hence are representation-dependent. For datatypes where the values are exact and are represented precisely and unambiguously - regardless of how they are represented - the mappings between CLID

standard datatypes and actual language datatypes are relatively straightforward, at least in principle. For datatype real (and, by extension, complex) the idea of having different representations of the "same" value disappears because one has different approximations to what may or may not be the "same" value. Two values may be mathematically the same but computationally the approximations may be different; similarly two approximations may be calculated in exactly the same way by exactly the same process but be different because their representations are different.

LCAS does not directly address the mapping question but supplements it by addressing the properties which an implementation of an actual real datatype in an actual processor should have. At the very least it can be used to ensure that implementors document how their real arithmetic behaves, so that programmers and designs of application packages can enjoy a greater degree of certainty about how portable their software may be to a different environment, greater predictability of what will happen and the extent of any variation in end results when a program is run.

Used normatively, it is found that some common hardware architectures cannot guarantee to deliver all of the desirable properties in all circumstances. Though this could be seen as potentially creating "political" problems of acceptance, the response should not be to achieve consensus by the all too common devices of inclusion of options or of weakening the requirements so that all architectures can conform. That would destroy the point of the exercise. Architectures, like languages, are designed with a variety of different aims with different priorities. The important thing is for users to know to what extent properties important for their application can be relied upon.

2.2.2 Relationship to IEEE floating point standard

These points can be illustrated by consideration of the IEEE floating point standard **[2]**. This is a standard for a particular kind of microprocessor architecture, and contains a number of options. Hence knowing that hardware merely "conforms to IEEE 754" or "is IEEE compatible" is not in itself enough to be able to predict with certainty how an application will behave. You need to know what options have been used. Naturally, this also adds to the difficulty of predicting whether an application will behave consistently if transferred from one IEEE-conforming processor to another. It turns out that IEEE processors conform to LCAS only for some combinations of options.

2.2.3 Proposed LCAS extension: mathematical procedures

LCAS in itself covers only the basic operations on real numbers in floating point representation, plus a small number of other operations, including square root, where maintenance of the desired properties can be ensured without difficulty. However, most practical applications use procedures involving chains and/or parallel combinations of operations, so that predictability depends not just on the basic operations but the algorithm used for the computation.

This applies to commonly used procedures usually built in to the language processor, such as the trigonometric functions, as well as to user-defined procedures. The proposed new standard can be seen as as incremental standard, to extend the coverage of LCAS into this area.

2.2.4 Proposed LCAS extension: complex

Some languages support datatype complex as well as datatype real. It is tempting to regard this as a trivial extension, by taking the traditional way of introducing complex numbers to

mathematics pupils as an ordered pair of real numbers representing the coordinates of a point in the Cartesian plane. The limitations of this approach are that the properties of approximate complex arithmetic implied by the Cartesian coordinate model are different from those implied by the (equally valid) polar coordinate model, quite apart from the properties implied by the representations of the relevant real values.

Hence the most appropriate starting point is not a simplistic use of a "pair of reals" model, however defined, but the more abstract concept of the complex domain.

This is enough to render the extension of LCAS to complex arithmetic non-trivial, which is why LCAS excludes complex values from its scope. The proposed new standard can again be regarded as an incremental standard to extend the coverage of LCAS, though of a significantly different style since its starting point is not LCAS itself but the principles upon which LCAS is built. The proposed scope covers both basic complex arithmetic and the mathematical procedures for datatype complex equivalent to those for datatype real in the other proposed LCAS extension discussed in the previous subsection.

2.3 Common Language Independent Procedure Calling (CLIP)

Again to get one issue out of the way first, the full registered title of this project is "Common Language Independent Procedure Calling Mechanism" In some papers it is therefore abbreviated to CLIPCM or CLIP-CM. It is now generally accepted among those working on the project that the word "mechanism" ought to be omitted, since it leads people into thinking it is a standard for implementing procedure calling. The word "mechanism" is not incorrect, since what it refers to is a language mechanism, which is an abstraction. Such a concept is understood (usually) in programming language circles but open to misinterpretation outside. Hence, recent discussions have tended to drop "mechanism" except in the official title, and to use CLIPC or CLIP as the abbreviation. CLIPC is more accurate but CLIP is shorter and is used here.

2.3.1 The CLIP project

The motivation of CLIP is that, whereas most if not all programming languages include the concepts of procedures and their invocation, they vary in the way that they view them, especially with respect to parameter passing - and this irrespective of any differences there may be about datatypes. In Appendix B (RPC tutorial) of the ECMA standard **[3]** for remote procedure calling (RPC) using OSI (Open Systems Interconnection), it is argued that "the idea of remote procedure calls is simple" and that procedure calls are "a well-known and well-understood mechanism...within a program running on a single computer". Both statements are true at a given level. Procedure calling is a simple concept, at the level of provision of functionality. It becomes less so at the language level (let alone the implementation and operational level) because of its interaction with both datatyping and program structure, as is testified by the numerous variations and (apparently) arbitrary restrictions on procedure definitions and calling in existing languages. It is also undoubtedly "well-understood" almost universally in the language community. The trouble is that this general understanding is not necessarily the same understanding.

This is to some extent acknowledged in section 3.4 of Appendix B of the cited document **[3]**. However, once these points are put together, along with the absence hitherto of CLID, the "simple and well understood" view of procedure calling becomes increasingly elusive; more tenuous, harder to sustain.

As with datatypes, then, the purpose of the CLIP standard is to provide a common reference point to which all languages can relate. Again, it is an enabling standard to aid

the development of language-independent tools and services; again, mappings will be needed; and for parameter passing it will need to use the CLID standard. It will aid the development of common procedure libraries. Operating systems very often contain such libraries, of things like the mathematical functions, which are shared by the various language processors running under them. However, they tend to be embedded in the environment and to be system- specific. The CLIP standard will enable such libraries to be specified in a standard way and built in a standardised language such as C or Fortran. It can be noted that numerical procedure libraries will be able to make good use of all three of CLIP, CLID and LCAS.

Another use of CLIP is to aid mixed language programming; in fact it was demands for this to be supported in a standardised way which led to the work item proposal. In mixed language applications, called procedures would run on language processors operating in "server" mode, and the procedures would be called from language processors operating in "client" mode. Note that the languages need not be different, and if the processors are the same the model collapses into conventional single processor programming. However, one can envisage some applications using a language processor with good diagnostics or a good human-machine interface to call procedures written in the same language running, say, under an optimising compiler or a vector processing server. (Hence the term should really be "mixed language processor computing" though that is rather clumsy.) In such a use of CLIP, the procedure calling mappings would be straightforward but not totally redundant; the CLID-based parameter passing would still usually be needed as a filter for incompatibilities caused by the use of optional features or implementation- dependent aspects allowed by the language standard. This could be dispensed with only if it were certain that the processors shared identical characteristics in every relevant respect.

The CLIP standard specifies conformity rules for language processors operating in client mode and in server mode. It is expected that many processors will conform in both modes.

It is important to note that CLIP does not address the question of how the procedure call initiated by the client mode processor is communicated to the server mode processor, or how results are returned. CLIP is concerned with mappings of calls at the conceptual language level, not at the communications protocol level. In the examples cited earlier, of generic procedure libraries and mixed language programming in the same host environment, such communications may be routine, even trivial. This was a deliberate design decision at the time that the scope of the project was defined: knowing that important application areas needing the standard existed in single environments containing no significant communications problems, it was felt that the CLIP standard should not run the risk of unduly restricting language-independent procedure calling because of constraints made necessary in contexts where communications limitations were greater. This brings us to the relationship between CLIP and RPC.

2.3.2 Relationship with Remote Procedure Calling using OSI (RPC)

The RPC project which exists in SC21 is based on the existing ECMA standard already cited. This standard defines a model of procedure calling and for remote procedure calling where the client's call of a procedure and the return of the results by the server are communicated through OSI protocols. In various respects the RPC model of procedure calling is more complete than the current draft of CLIP and as such it provides valuable source material in developing CLIP. However, it is apparent that the model is predicated on the client and server communicating by OSI protocols. In CLIP terms, it is constrained by limitations at the communication level which are independent of the language level which CLIP addresses. Another way of putting it is that RPC is driven by OSI rather than by procedure calling.

In principle CLIP and RPC are not in conflict; the task is to ensure that they can coexist without conflict. It is vital that CLIP is not defined in a way that would create problems for the RPC/OSI community, since RPC applications are potentially just as important as procedure library and mixed language applications - indeed can complement them by extending their range. However, as mentioned earlier it is equally vital that CLIP itself not be constrained by communications considerations.

The ideal solution would appear to be for CLIP to define a generic model of procedure calling but be careful not to stray beyond that area, and for RPC to define the gateways between CLIP client and CLIP server language processors including additional conformity requirements as needed in "remote" (OSI) contexts and the OSI-based communications between the two.

It is to be hoped that forthcoming meetings will resolve these matters and ensure that there will be no incompatibilities between CLIP and RPC, and no unnecessary duplication of coverage. Duplication, as well as duplicating effort, would carry the dangers of leading to confusion in the future for potential users of the standards, and of risking the development of incompatibilities e.g. if revisions of either standard were undertaken.

2.4 Guidelines for Language Bindings

2.4.1 Introduction

There is one further WG11 project, indeed historically the first, to produce guidelines for language bindings. These guidelines will appear not in a language independent standard but in an ISO Technical Report, TR 10182, which is in the final stages of editing and is likely to appear in 1991. This project is covered here because of its relevance to the implementation of functional standards in a programming language environment - see section 1.2 above.

This project was the outcome of the experience gained from the pioneer work on language bindings of functional standards, particularly for GKS. The aim of the report is to make the benefits of that experience available to future workers in the field. The report classifies language binding methods and suggests guidelines that should be found useful by those interested in various aspects of sharing functional resources between, and communicating between, different language environments.

2.4.2 Functional Overview

The first section is a functional overview of different approaches to language bindings, and investigates five separate methods, listing the advantages and disadvantages for all. It reaches the conclusion that no one method is appropriate in all circumstances or for all languages, so in practice a combination of methods is probably appropriate.

2.4.3 Guidelines

The report provides more than fifty guidelines that it recommends be followed by those concerned, to ease the specification and implementation of language bindings of functional standards. These vary from the procedural "Either the language committee or the system facility committee should have primary responsibility for the language binding ...", through general technical guidelines "The system facility should recover from errors wherever possible language if the dynamically specified length of an array is zero."

Each guideline is accompanied with various comments and descriptions of unresolved

issues.

2.4.4 Conclusion

The report finishes with some comments on future directions, followed by a series of annexes giving an example of language bindings, and also how the unresolved issues affect one specific language binding implementation.

2.5 The POSIX projects

While the projects described above are all taking place within or are directly related to work going on in WG11, the POSIX projects are going on in another SC22 working group, WG15, in collaboration with an IEEE working group, P1003, in which the work originally began and which is still primarily responsible for producing the successive drafts of the various sections of the emerging standard. Though mostly this work continues independently from that in WG11, there are areas of common interest, and liaison is maintained informally through overlap of membership of the two groups, with provision for more formal liaison, e.g. joint meetings, should the need arise.

This overview of the POSIX work is included in this paper since it was felt that this would help to "complete the picture" for the programming languages community as far as language-independent standards are concerned.

The POSIX definition, as currently defined by P1003 and WG15, is derived from the original AT&T specification. This specification contains a description of the functionality of the 'system calls' (entries into the UNIX supervisor), together with a specification (in the C language) of the interface to the system calls. Note that the manuals of the versions 6 and 7 of UNIX also contain an interface description in PDP11 assembler.

At the time that it was proposed that the standardization work on POSIX be started in ISO, the following problems with regard to the specification method (as used originally by AT&T, and later in IEEE) were identified:

1. The C definition on which the interface description was based came from the book by Kernighan and Ritchie, and was not in line with the work going on in ANSI X3J11 and SC22's WG14.
2. When the only specification of the POSIX functionality is dependent on a binding to one specific language (in this case C), this can result in problems being created for bindings to other languages.

These bindings could be obtained either by defining a mapping of the language specific interface onto the existing C interface, or by defining both the interface AND the functionality in a language dependent way in one, self-contained document (the latter is called a 'thick' binding, a 'thin' binding contains only a mapping of the interface). The problem with thin bindings to C is that too much C dependency must be introduced in the binding, thick bindings have the problem that the POSIX functionality is defined in more than one place (when the two are not identical, the question arises: which one is right? Also maintenance problems can be foreseen in keeping the two aligned).

In order to overcome these problems, ISO has asked IEEE (who produces the text for the ISO POSIX standards) to:

1. provide text for a language independent POSIX specification, and
2. provide text for the (thin) language specific bindings to this language independent POSIX specification; the languages that are requested at this moment are C, Ada and Fortran-77.

An IEEE group is now working on the production of a technical report on Programming Language Independent Specification methods. This technical report should be used for the production of the language independent POSIX specifications.

The affected parts are 1003.1 (System Services), 1003.4 (Real-Time extensions), 1003.6 (Security) and 1003.8 (Transparent File Access). In ISO terms, 1003.1 means ISO 9945-1:1990, the other parts will be amendments to ISO 9945-1:1990.

In the current draft of this technical report (Draft 2.0, October 1990) a set of datatypes is defined that very much resembles the datatypes from the CLID Draft 4. Procedures are modelled as abstract interfaces. This model allows input and output parameters of any type. All parameters are passed by value. Information passed between the caller and the service is contained wholly within the procedure parameters.

A preliminary investigation has shown that some of the well known C interfaces to system calls have to be rewritten extensively: too many C dependencies are crept in. Examples are those system calls, where the interface describes different behaviour (and different types of return values!) based on the value of one of the parameters. The system call *sysconf* (see ISO 9945-1, section 4.8) can return, depending on the value of the input parameter, values like "the maximum number of bytes supported for the name of a time zone" or "the number of files that one process can have open at one time". Languages like C (with almost no type checking) can handle this kind of construct (whether you like it or not), in other languages (Pascal, Ada) one will have a problem.

It is suggested to have in the language independent specification, a separate interface for all currently admissible parameter values, with a different name for each interface, and a correct return value specification. If needed and wanted, the C bindings document (the thin one) can map all these interfaces again on the current *sysconf* call.

Note that the document that is currently circulating in SC22 as the Ada POSIX binding (SC22/N843) is a thick binding, and is only sent to SC22 for review and comment, not for adoption as a working draft.

The thick Ada POSIX bindings document shows clearly one of the drawbacks of the 'formal' approach as outlined above: the language used in the document is completely targeted at the Ada user. In the formal approach, the user should use two documents (the language independent specification plus the language bindings document). The language independent document will, for instance, describe processes in terms that are likely to be completely different from the familiar Ada terminology. It is clear that this might cause confusion.

3. Concluding remarks

It is clear that we are still on a steep section of the learning curve for dealing with language independent facilities and standards, both for those working on the language independent projects and for those in the various language communities. Traditionally the language communities have been largely independent and disjoint, doing things their own way and talking only to their own kind - and those who move to the cross-language projects cannot

always cast off immediately all the inbuilt assumptions from their respective backgrounds, whether they be from Fortran, Cobol, Pascal, C or whatever. It is hoped that this paper will help to clarify some of the issues and help readers to climb a bit further up the learning curve.

4. Status Update on Projects, September 1991

4.1 Guidelines for Language Binding

The document is technically finished and is being prepared for publication as TR 10182. Target date: December 1991.

4.2 Common Language-Independent Datatypes

The first CD (CD 11404) was circulated in SC22 in May 1991. The second CD is expected by April 1992.

4.3 Common Language-Independent Procedure Calling Mechanism

The first CD is expected by April 1992.

4.4 Language-Independent Arithmetic

The first CD (CD 10967) was circulated in SC22 in March 1991.

In the September 1991 plenary of SC22 it was decided that this CD will be the first part of a multi-part standard on Language-Independent Arithmetic, with the following parts:

10967-1 Integer and Floating Point Arithmetic
Second CD 10967-1: April 1992

10967-2 Mathematical Procedure Standard
CD registration February 1993

10967-3 Complex Arithmetic and Procedure Standard
CD registration: October 1993

5. References

[1] Meek, B.L., Two-valued datatypes, Sigplan Notices of the ACM, Vol 25 No 8, pp 75-79, August 1990

[2] IEEE standard for binary floating point arithmetic, ANSI/IEEE Std 754-1985

[3] ECMA standard RPC (Remote procedure call using OSI), ECMA-127, final draft 2nd edition, January 1990

[4] Payne, M., Shaffert, C. and Wichmann, B., Proposal for a language compatible arithmetic standard, Sigplan Notices of the ACM, Vol 25 No 1, pp 59-86, January 1990

[5] Wichmann, B.A., Getting the correct answers, National Physical Laboratory Report DITC 167/90, June 1990