Too Soon, Too Late, Too Narrow, Too Wide, Too Shallow, Too Deep

by Brian Meek
KINGS COLLEGE, LONDON

■ What makes standards succeed or fail is the subject of much speculation, often during late-night chat. Speculation it always remains: firm conclusions are never reached, or do not bear the scrutiny of cold and sober dawn. Anecdotal evidence is not in short supply, and case studies can be done, but little can be translated into general principle. Standards, it seems, are sensitive plants; one will "take" and thrive, while another, to all appearances equally fit, will struggle to survive at all.

One thing is certain: technical excellence is not sufficient for success. More worryingly, it is demonstrably also not necessary. Some of the anti-standards brigade (and some cynics who claim to be basically pro-standard) even claim that a successful standard cannot be technically excellent, because it is bound to be a compromise between conflicting interests, or because it will be obsolete by the time it gets established, or both.

That proposition will not be discussed here, but will be taken instead as a starting point. It is assumed that, as IT professionals, we want to make a standard "succeed," both technically (to be adequate to the task) and practically (to become accepted and used). We know that many IT standards succeed in one sense or the other, but very few succeed in both.

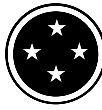
Note that the technical requirement is to be adequate to the task, not to be the best achievable, a point pursued elsewhere [Meek 1995]. Technical and practical objectives don't always lead in the same direction. We sometimes hear it argued that there's nothing worse than a bad standard (see e.g., von Sydow [1990])—but sometimes that a bad standard is better than none. Like so much in this area, the answer is not clear cut. It depends on what the aim of the standard is, its context, and the nature of its badness. And even if "adequate" versus "best achievable" is not in itself an issue, what constitutes "adequate" can be, though doubt can be reduced by having clear objectives and conformity rules [Meek 1995].

So, given that we cannot guarantee that a standard will succeed, what can we do to ensure that our standard will be of adequate or better technical quality without putting its chances at risk, so that, should it thrive, it will be seen to be a good standard, rather than "dreadful, but it's all there is," or worse?

To succeed in practice, our standard will have to meet a need: a need which is perceived; which is regarded as important; and which cannot be met in any other way than by a standard. Many factors will influence this process; many quite outside the control of the standards-makers. Two strong factors are timing and sizing, of which the second has two

Permission to make digital/hard copy of all or part of this material without fee is granted provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 1996 ACM 1067-9936/96/-0600-114 $3.50



dimensions (at least): breadth and depth. It is variously said of standards that they are too soon or too late, too narrow (in scope) or too wide. You seldom hear too shallow or too deep, so this article will mostly discuss depth, the aspect that seems to have attracted the least attention in the past.

Timing

Often, standards are castigated for coming too late, a charge frequently coupled with complaints about the slowness of the standards-making process. However, try to produce a standard before a technology has crystallised, and you will get complaints that it is "too soon" to decide a standard; either not enough (it is claimed) is known, or, if the anti-standards lobby cannot make that stick, "there may be unforeseen implementation difficulties." It is sometimes supposed that there is a "magic moment" between "too soon" and "too late"—but such a moment may not exist. Mike Sykes made this point years ago in a published discussion [Sykes 1980]:

> "You ask whether standards get produced too soon or too late. My answer is, 'often both'. It is almost impossible to know what the standard should be until there is some experience, by which time there is an investment in something that is going to be nonstandard. Even where the standard is seen to be needed, as in communications, different standards are adopted by (or thrust upon) some parts of the user population before being generally agreed—which may dismay us, but shouldn't surprise us."

Sykes was referring to programming language standards—though note the remark about communications—but the argument is a general one, and seems as valid today as it was in 1980.

Standards committees can probably not do much to get their timing "right," as in "most favorable"—"least unfavorable" is perhaps more accurate. Too much is outside their control. However, they can do something. They can set themselves specific, achievable targets, and stick to them [Meek 1995]. They can set deadlines, and meet them. They can resist the temptation, having achieved "the good enough," to spend further time seeking "the perfect" [Meek 1995]. They can watch out for, and marginalize, the occasional member who for some vested interest or personal agenda tries to prevent consensus rather than achieve it. A good convenor is key here, but all members can help.

These suggestions are aimed at preventing "too late" rather than "too soon," but most of the standards process tends to work that way. (Will fast-tracking of "publicly available specifications" go too far the other way?) Anyway, once a standards project has started, people start looking for results. I have often argued that standards are products, like hardware and software, and committees are well advised to heed the lesson of marketing: if you've announced a product, don't take too long deliver it.

Sizing: Breadth

An important aspect of sizing is breadth, or how much the standard covers. The essence of success here is to get the scope statement right, though conformity rules can play a part. Too narrow a scope will not deliver tangible benefit (in the absence of supporting standards and required conformity to all). Too broad a scope may make it too large for implementors to swallow—its demands will be too great, or appear so. Furthermore, the sheer size may lead to timing problems, of the "too late" variety; in extreme cases the standard may never be finished at all.

Larger numbers of people can help, if they are willing to subdivide and work in parallel, but they will need to trust one another and maintain coordination and good project management; but management overhead can itself become a problem, as we know from other walks of life.

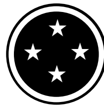
Many large standards are not just developed in parts but also published in parts. Designing those parts and the conformity rules for them, and determining the publication pattern (simultaneously, sequentially, or a mixture), are tasks that require great care and skill.

One form of "too wide" is potentially the worst of the lot: open-ended standards. These are always bad, and they can be disastrous. At best they adversely affect timing: if scope of the project is not agreed on before the start, committee time is wasted. And how can you ever be sure you have finished? That can be hard enough anyway, but an open-ended standard is an invitation for any member so inclined to propose something additional, at any time. As for breadth, even if the original scope statement is quite narrow, you cannot know until you finish (if you ever do) how broad the total scope will be.

Member bodies should never approve a new work item proposal (NP) with an open-ended scope; but they do. Excuses like "giving the committee freedom and flexibility to use their judgement" and the like should be treated with the scorn they deserve. The JTC1 "study period" idea may help here, but is sometimes needed for things that would not qualify as large new work areas. It would be nice if open-ended scope statements in NPs could be made "unconstitutional," allowing secretariats to reject them before they ever go out to vote.

Sizing: Depth

As mentioned earlier, the depth of standards sizing has not received a lot of direct attention—certainly not as much as timing and breadth. "Depth" means, partly, the thoroughness with which the standard covers its domain of applicability. Usually, any shortcomings manifest themselves as gaps: the standard is deficient in covering its scope, i.e., it is underspecified.



UNDERSPECIFICATION

Various deficiencies are found that lead to underspecification. Many standards have exclusion statements in them: “this standard does not specify . . .”, “. . . is implementation-dependent” and the like. These may be scattered throughout the text, or gathered together as a list. It is best to do both. The list warns a user what to look out for. Claims that a product is standard-conforming need always to be checked against that list. On the other hand, references to exclusions in relevant parts of the text are also valuable reminders.

The presence of an exclusion statement does not necessarily mean that the standard is underspecified, too shallow, or narrow. It may simply be there as a reminder of the scope, to those who might assume too much about what the standard delivers.

Ideally, the standard should distinguish between exclusions that do or do not relate to something within the scope, and give a rationale for those within. However, the committee may not wish to draw attention to or defend such exclusions, especially if the reasons for them are political rather than technical. Indeed, it is not uncommon for there to be omissions that are not acknowledged at all: the standard is silent on the matter, even though it is within the scope. This is worse, because it is harder to see something that isn’t there. Such unacknowledged omissions are not always deliberate: they may be oversights or the result of an assumption. They still constitute underspecification.

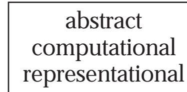
Underspecifying can occur in two further ways: by inadequate conformity specifications, or by the use of options, the two often coming together. Options at least have received some attention [Meek 1990, 1993]. They come in two forms: take-it-or-leave-it, and this-way-or-that. They look like a form of overspecification, but take-it-or-leave-it can mean underspecification if one must “take it” to achieve the aims of the standard. This-way-or-that can mean underspecification if the difference between the alternatives harms the aim of the standard and the standard fails to take a position on the matter. Many have drawn attention to the problems caused by options in OSI standards. These eventually led to a whole new class of standards, International Standard Profiles, or ISPs, together with a mechanism for getting them approved, simply to plaster over the deficiencies.

The preceding discussion assumes that the scope and aims of the standard are properly specified. This is not always the case. We have noted that exclusion statements may be no more than reminders of the scope, but the scope itself may be flawed. For example, exclusions may be enshrined in the scope statement that undermine the stated aim. Such inadequacies ought to be sorted out at the NP stage, but they may not be.

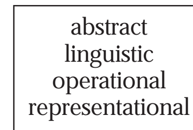
OVERSPECIFICATION

To discuss overspecification properly, we need to introduce the concept of levels of abstraction, first used by the author[1] informally in standards meetings in the early 1990s, before its later more general publication [Meek 1994]. The idea was originally applied to standards relating to programming languages, but seems capable of generalization, though it is not claimed that it can be applied universally.

In the original version, elements of programming languages were considered to exist at three levels of abstraction [Meek 1994]:



where the computational level could be divided into two sublevels:



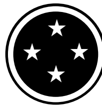
The idea was that the linguistic elements were instantiations at the computational level of the abstract concepts, while the operational level dealt with manipulation of the elements, which inevitably looked “downwards” to the realization of the elements in actual, processible entities at the representational level. It is important to note that the representational level is not regarded necessarily as the physical hardware level, or even the slightly more abstract bits and bytes; it depends on the linguistic element concerned. The idea that a complex number is, for computational purposes, an ordered pair of real numbers, is also a representational assumption, even without any assumption about how computational real numbers are represented either logically or physically.

The similarity between the concept of levels of abstraction and the idea of the seven layers in the OSI Reference Model is obvious, though one should not read too much into it. The one was not derived from the other. They are similar responses to a need for a systematic approach; there is no direct relationship.

The first of the above diagrams appears to generalize fairly well; there is nothing specific to programming languages there. A possible generalization of the second diagram which might be useful in a variety of contexts could be

1 Though the term has been used by other writers (see, e.g., below), the author is unaware of any that preceded his own; he would be interested to learn of any. In any case, the concept, in the sense explained here, was identified and developed independently, apart of course from any subconscious influences (see main text).

Supplement 2 *(Guidelines for JTC1 API Standardixation)* to the third edition of the ISO/IEC JTC1 Directives devotes clause 1.2 to the concept of “Level of Abstraction” and applies it to APIs (Application Program Intefaces). The clause describes the concept as “complex” and says that the use of the term “implies variation in the amount of functionality offered to the calling program by each invocation” but later says that another usage “reflects the degree to which the implementation method is visible/invisible to the users of the API specification calls”. The present author does indeed regard the view of that Supplement on levels of abstraction to be complex—“complicated” would perhaps be better—and hopes that readers will find that the exposition here makes the concept easier to understand and more useful.



abstract
static
dynamic
representational

though it is readily accepted that some IT entities may not conveniently be divided into static and dynamic levels, and that for some, ordering them as shown above may not be appropriate. They may have sublevels of their own.

In this general case, the abstract level is still concerned with concepts, independent of the technology, while the computational level is an expression of these concepts, taking into account the nature of the technology and the properties and limitations of computing systems, but without specifying anything about the way they are implemented. That belongs at the representational level.

In any particular instance, any or all of these levels can be divided into sublevels; for example, the representational level will almost certainly always be divided into a logical representational level and a physical representational level. It might even be helpful to subdivide sublevels and produce a hierarchy of levels.

Levels of Abstraction Applied to Standards

Applying this idea of levels of abstraction to the depth of standards, we get the following. (For simplicity, "levels" from now on will mean levels or sublevels, as appropriate, not just the three main ones.)

(1) A standard existing at the abstract level is not an IT standard, even if it specifies something intended for realization on IT systems.
(2) A standard should be confined to one level of abstraction, or to a set, preferably limited, of adjacent levels.
(3) A standard should cover its scope thoroughly at its level or levels of abstraction. If it does not, it is underspecified.
(4) A standard should not stray beyond its level or levels of abstraction. If it does, it is overspecified.

In connection with point (1), an example of a standard at the abstract level is an information standard, like ISBN for books. Note that apparently abstract (or non-computing) standards relating explicitly to an IT environment are not (in our sense) at the abstract level but are at the computational level (because they take the technology into account), perhaps at some conceptual sublevel. Examples might be standards for IT documentation, or safety procedures, or design methodologies.

One advantage of the approach advocated here is that it allows one to get a handle on the vexing question of "implementation-dependence." Innumerable arguments must have occurred on standards committees, for good technical reasons or for more dubious ones, as to whether something should be left implementation-dependent or not. If it is outside the levels of abstraction addressed, the answer is clearly "yes, it should be"; if it belongs at the standard's proper levels, the answer is "no, it should not be" (or, more precisely, "if it is, then the standard will be damaged, its technical quality will be reduced, and it will be underspecified"). Similar considerations apply if the argument is about options.

No doubt some arguments in the past have turned, implicitly, on the level of abstraction being addressed, albeit probably expressed in topic-specific terms, with every case be dealt with ad hoc. A clear committee view of at what level or levels the standard is provides a basis for assessment, and reduces the obfuscation caused by lumping all matters relating to implementation-dependence ("supplier freedom" etc.) into a single category. It will be clearer what the technical damage is likely to be, and so easier to decide if it is worth doing to resolve some conflict or to placate some vested interest. If you are lucky, it may even suggest a "fix", at the level addressed, which resolves the issue.

On this basis, I suspect that many IT standards are underspecified, some are overspecified, and some are both. Examples abound of each of these cases in the programming languages area where these ideas developed. Underspecification is rife, through leaving elements implementation-dependent, but overspecification occurs too, in the form of representational assumptions, explicit or implicit. You can even get the same things being both underspecified and overspecified, being partly defined in abstract terms but with representational undertones. Languages designed for systems implementation or certain kinds of process control application do need to address some things at the representational level, but many do not.

Levels of Abstraction Applied to Scope

Levels of abstraction can be and have been used to clarify issues surrounding the scope of standards. An instance occurred during the development and approval process of ISO/IEC 10967-1:1994 Language independent arithmetic, Part 1, Integer and floating point arithmetic. The issues concerned the relationship between the scope of that standard, and that of IEC 559:1989, better known as IEEE 754:1985, Standard for binary floating-point arithmetic.

The issues were several, interacting and somewhat overlapping, and generally confused during the discussions that occurred while ISO/IEC 10967-1 was progressing through its various stages. The questions were whether the two standards addressed the same area, whether they were in conflict and whether there should be two standards when one (i.e. IEEE 754) was enough.

An examination of the two standards, and their scope statements in particular, show that LIA-1 and IEEE 754 do indeed appear to "address the same area"—provided one thinks only of breadth, and ignores depth. Their respective scopes make it clear

that they may address the same area, but they do so at different levels. The specifications of ISO/IEC 10967-1 are at the computational level that the applications software user sees. Those of IEEE 754, in contrast, are at the representational level. More precisely, it belongs to a logical or "abstract architecture" sublevel of the representational level, between the computational level of ISO/IEC 10967-1, and a more specific system implementation sublevel.

Of course, it would matter if the two standards, in overlapping in the breadth dimension, were inconsistent in their specifications. This is not the case, however; ISO/IEC 10967-1 was not developed without taking into account the earlier standard.[2] IEEE 754 was recognized as a high-quality standard which, if used properly, did indeed meet the LIA specifications. There is no clash; indeed, the two standards support each other. IEEE 754 specifies an abstract architecture which makes meeting the ISO/IEC 10967-1 requirements a straightforward task; ISO/IEC 10967-1 supplements IEEE 754 by specifying how best to use it and providing a way of documenting what it does.

The confusion caused by lack of understanding of levels of abstraction led some to complain that LIA-1 "undermined" IEEE 754 by "allowing" other (less desirable) hardware architectures—the (supposed) move towards (hoped-for) universal adoption of IEEE 754 would be put at risk. Such arguments of course came from those who, for various reasons, felt strongly about particular architectures. A related argument, less dependent on value judgements and commitment to a cause, was "why produce another standard when IEEE 754 has already met the need?"

There are two parts to the answer to these arguments. Firstly IEEE 754 does not fully "meet the need." IEEE 754 is somewhat underspecified as a standard at the representational level. As an attempt at that level, to meet the computational level requirements of LIA-1, it is also underspecified: it enables them to be met; one might even say that it encourages their being met, but it does not guarantee it, which is what a standard ought to do. ISO/IEC 10967-1 does guarantee it. (Once again, it is the presence of elements like options in IEEE 754 that do the damage.)

Secondly, of course, if you try to advance IEEE 754 as a standard for specifying computational level requirements for arithmetic operations, it is also a gross overspecification—because of its representational nature. Like some aspects of some programming language standards, inherited from their original design, things are specified at the representational level which (for this purpose) ought not to be. Of course, there are legitimate reasons for having a standard which does specify representations at this level, and it is right that it reflects computational level requirements, but it should be regarded as one way of implementing those requirements, not the only way. In other words, the careful representational neutrality of ISO/IEC 10967-1 is right, and its integrity should be preserved.

That argument quite clearly holds independently of whether IEEE 754 is underspecified for this purpose: remove the weaknesses of IEEE 754, and the argument's truth would be undiminished. It is wrong for standards intended to address issues at the computational level to depend as standards on representational assumptions. That is not just a matter of principle, but one of simple practicality: get the abstract and computational levels of abstraction right, and you are free to seek better ways of implementing them at the representational level. If representational standards are needed, as well they might be, then these can be improved and upgraded, without dangerous or troublesome implications at the less concrete levels.[3] It is a way of avoiding at least some instances of that oft-voiced complaint that standards "inhibit progress." (Often, in fact, it is the representational standards, or representational assumptions in standards, that impede progress because implementors and suppliers get locked into particular representations.) In the case of IEEE 754, only the most partisan of its proponents could believe that it is the only possible good way of handling floating point at the representational level, incapable of being matched by something different, or of being improved upon.

Conclusion

The concept of levels of abstraction as presented here has demonstrated its value in clarifying issues and determining how well a standard matches up to the technical demands made on it. It will not, regrettably, tell us whether a standard will succeed or fail, but may help to indicate if a standard has succeeded, what its strengths and weaknesses are, how best to use it, and where to start looking for improvements, as and when the time comes for a revision. **SV**

References

IEEE 754 1985. IEC 559:1989. Standard for Binary Floating-Point Arithmetic.

ISO/IEC. 10967-1. 1994. Language Independent Arithmetic, Part 1: Integer and Floating Point Arithmetic.

MEEK, B.L. 1990. Problems of software standardisation, Comput. Standards Interfaces 10, 1, 39–43.

MEEK, B.L. 1993. There are too many standards, and there are too few. Comput. Standards Interfaces, 15, 1 (May), 35–41.

MEEK, B.L. 1994. Programming languages: Towards greater commonality; ACM SIGPLAN Not. 29, 4, (April), 49–57.

MEEK, B.L. 1995. The seven golden rules of language-independent standardisation, In Proceedings of the Second IEEE International Symposium on Software Engineering Standards (Montreal, Aug.), 250–256 (ISSES).

VON SYDOW, E. 1990. Beware of bad standards. In An Analysis of the Information Technology Standardization Process. J. Berg and H. Schumny, Eds., Elsevier-North Holland, 267–271.

2 Would that it were possible to say that throughout the IT standards world. It is far too often the case that standards appear to be developed in various groups in apparent disregard of what has been or is being done elsewhere.

3 This is certainly a way in which the concept of levels of abstraction shares characteristics with the principle of the OSI seven-layer model, if not always its realization.