FILE

SOME REQUIREMENTS FOR SPECIFICATION LANGUAGES

C.B.Jones

1976, February 27.

Abstract

There is evidence that a large proportion of errors in past computer systems have resulted from weaknesses at the specification and design stage. The complexity of the systems which are being considered today is such that precise specifications are mandatory.

Although practical experience with some particular specification languages is discussed, the approach adopted here is not to propose one language but rather to identify those concepts which appear important in writing specifications. It is argued that any putative specification language should be capable of expressing these concepts.

VAB

Note

This note contains basically the same material as a paper of the same title which is to be presented at an IBM symposium in Yorktown Heights Research Lab.

0. INTRODUCTION

A number of studies of the development process for large program systems have indicated that a significant proportion of their errors result from the very early stages of the process. This observation has not been a surprise to those who have had to base their own work on the specificaticns cf any large system either hardware or software. The, so-called, specifications fail in their task of delineating what should be the final system: for those who should build it, their job is not defined; for those who should build to employ it, their assumptions are not defined. Most seriously, the two (or more) views of the specification may make divergent assumptions as to what was intended.

Nor does the problem diminish when the development project moves beyond its overall specification to the early stages of design. The systems under consideration today are such that the early stages of design must, in turn, generate large specifications. For a design to be correct, the combination of components built according to the sub-specifications should fulfil the overall specification: the probability that this is true, given specifications of todays quality, is low.

The subject of this paper is "Specification Languages", and from here on no arguments will be offered for the need to improve on the techniques in current use.

It is not, however, the aim of this paper to argue that one particular specification language will completely resolve the problem. Rather, the intention is to list some properties which any putative specification language should embody. These properties are developed in sections 2 - 5 from an analysis, which is given in section 1, of what a specification should achieve.

The lack of a concrete proposal may disappoint the reader. There are two main reasons why the author is reluctant to go so far at this time. Firstly there are certain technical problems (see section 7) without whose resolution it would be premature to claim to have a final solution. Secondly, there are many matters of taste in designing any language and it is a mistake to cloud the issue of what is required in a specification language with what should be the secondary questicn, cf choosing its concrete syntax.

That a paper with the current objectives is required is indicated by the flagrant way in which the points of the last paragraph are ignored. Proponents of some new specification technique tend first to become dogmatic about concrete syntax and only later, if ever, attempt to show how any basic problems have been resolved.

Section 6 will discuss practical experience with a series of meta-languages which have been employed in the specifications of various systems and embody many of the properties defined. A summary of the main points of the paper is given in Section 7.

1. WHAT IS A SPECIFICATION?

A specification should specify what is an acceptable system. From the point of view of those responsible for implementing the system this means that when their work is complete the only objections which can be raised will contain a reference to a requirement in the specification which has not been met. (For the bulk of the current paper, only the function of the system is considered: Section 7 includes some comments on performance specification). The potential user of the services of the system being specified has another viewpoint: the specification should define those functions on which he can rely. If a user assumes other properties will hold, he does so at his own peril.

It is important to see where such a specification can be created in the development cycle. It is not the first thing that gets written about a system, because it is too difficult to simply begin writing and expect to produce a final "contract" between user and developer. The first sketches of various desirable properties, which the system under consideration should have, play an important part in the process of developing the specification. But they are not the specification itself, this is created out of the intuitive sketches (and since this process provides feedback, it is iterative).

Having said what the specification should define, it is worth considering what it should not constrain. Unless they are essential to the function which is to be performed, the specification should not dictate implementation techniques. The choice of how to achieve his stated objective is exactly the role of the designer (this does not preclude the possibility that the same people will be involved in writing both specification and design. It only argues that, for the benefit of the documents that must be produced, the roles should be kept separate.) There are several reasons why commitment to design decisions should be kept to a minimum in the specification. Firstly the specification may be used for more than one implementation, on different machines for example. Even if this is not the case the design decisions are likely to be more fluid than the specification. Perhaps most importantly, it is frequently possible to specify what is required much more succintly than to describe how it can be achieved. For this reason, and because of the extra checking possible, it is often profitable to document a specification even if the implementation is "known".

If a specification is to define exactly the properties the proposed system should possess, what properties do we expect of the specification itself?

A specification must clearly be precise. If it is ambiguous and allows the possibility that two readers will interpret a requirement differently, the contract between user and developer will fail to fulfil its purpose.

To be meaningful, a specification must also be non-contradictory. Although one can state with some precision the properties required of "the largest prime number", the precision will not facilitate its computation.

Furthermore, a specification must be complete in that the user must not be allowed to reject a final system which matches all of the documented requirements on the basis of one not recorded in the specification.

To be useful a specification must be reasonably short. If one were prepared to accept a definition a posteriori, the code listing of an operating system satisfies the properties given above: it would not, however, be considered as a reasonable document from which to understand the properties of the system.

The question of utility also comes up with regard to organisation. If a particular system concept is being considered it should be possible to locate all those places which contribute to its specification. Furthermore, the number of places should be kept to a minimum.

The language in which the specification is written must itself be comprehensible. A distinction will be made below between the ability to read or write the specification language. In both cases it is important to avoid unecessary barriers for the user.

The six properties required of a specification have not been presented in any order of importance. The last three are, in a sense, in a different plane from the first three. It is not clear that this is a consistent set of properties! It is hoped that the practical experience outlined in Section 6 will indicate that such a goal is obtainable.

The discussion below (Sections 2 - 5), identifies the properties required of a specification language and is not in one-one correspondence with the above list. In each section, an indication of which specification properties are being considered will be given.

2. ON FORMALITY

The meaning of a program written in a high level programming language is fixed in the sense that it will always compute the same algorithm. (How one can precisely define the semantics of a programming language leads directly back to the subject of specifications!) In the sense that their semantics are fixed, programming languages are formal. It is in this sense that the current author believes a specification language must be formal. Only if the semantics of the specification language are fixed can one write precise specifications. If, as is the case with natural languages, statements can be constructed which are ambiguous, it will not be possible to guarantee unambiguous specifications.

There is one key area where a precise description is particularly crucial and that is the description of objects (see below for discussion of "state"). The essential structure of all objects mentioned in a specification must be defined. Moreover, the description must be such that precise ways of referencing the objects and their components are defined.

An interesting example of a specification which attempts to use English as a definition language is the joint ECAM/ANSI proposed PL/I standard (ref /18/). In this definition the requirement to define precisely the class of (abstract) PL/I programs and the state of the defining machine led to the use of a formal abstract syntax notation. The attempts to use natural language in order to describe the state transitions has led to a very stilted style which is by no means easy to read. The level of formality of this standard is a considerable step forward over its predecessors, but it might well have been easier to make the complete step to a formal meta-language.

The development of formal languages is not an easy task and if one were to begin afresh for a specification language the burden on its designer and student would be unacceptable. There is then a very strong argument for basing a specification language on existing notation where appropriate. In some cases it will be advantageous to use notation known from mathematics (e.g. set theory, functions) or logic (e.g. propositional connectives). This use of an existing base should aid communication. Unfortunately, because of a misunderstanding, it actually hinders communication. The misunderstanding results from the concern that a potential user might have to be well versed in all of the areas of mathematics from which notation is borrowed. It is, however, only the notation which is normally adopted, not the deep results of the branch of mathematics.

In spite of the above praise of formalism, it is not intended to argue that the whole of a specification must be completely formal. Not only are there some items which are so much part of our common experience base that they can be adequately communicated by a comment (e.g. Sort a vector) but also there are times when it is convenient to specify by a comment knowing that the comment will be replaced (or supplemented) by a more formal description at a later point in time. The claim that a completely formal specification can be well organised in the sense of localising information is made plausible by experience with well-structured programs and supported by the evidence of experiments (see section 6). It is, perhaps, less obvious how a specification which contains some informal comments can be so organised as to provide extra information. Consider an incomplete program which in place of a block has a comment purporting to describe the effect of that block. Not only may the algorithm be incompletely specified, but also it is not a priori clear which known variables will be referenced or changed. If however the block is replaced by a call to a (external) procedure with a given argument list, it will be necessary to show (via name/value distinctions) more about the effect of the final block even though its function is defined by basically the same comment. With care, this method of incorporating informally defined functions into a specification will do little damage either to the precision or organisation of the whole. Moreover, it will always be clear which parts require more precision in the case of uncertainty.

Another argument in favour of using a formal specification language comes from the desire to provide mechanical aids to the writer of a specification. Such aids might range from a syntax checker and cross-reference program to a retrieval system which could, for example, locate all references to certain classes of objects. Such programs would require, at least, a description of the syntax of the specification language.

The arguments to this point for formality could all be satisfied by a programming language (although the next two sections will indicate directions in which this would be inadequate) there is a further step in the requirement of formality which would anyway preclude the unrestricted use of almost any current programming language. The subject of proofs of implementation correctness will not be covered in this paper (see ref. /12/). But if one desired to use a specification as a base for correctness arguments of implementations, it would not be enough that the semantics of the language were fixed: it would also be necessary that the semantics were defined in terms of mathematically tractable objects. The experience of attempting to prove compiling methods correct from the original PL/I definitions (definitions ref /15/, proof work ref /10/) showed that, although the meta-language had a carefully defined meaning, the proofs were unnecessarily cumbersome. The meta-language used in the more recent definition of PL/I (ref /1/) is defined more directly in terms of mathematical functions and is in consequence a more suitable base for correctness arguments.

It is thus claimed that choosing a formal, rather than a natural language, for the specification language will further the aims of precision, conciseness and organisation as well as simplifying a check for completeness of specifications written in that language.

VAB

3. ON IMPLICIT DEFINITIONS

In considering the input/output relation for a function, it is sometimes easier to define the properties required of a result rather than to provide an algorithm for its computation (e.g. square root, matrix inverse, etc.). This might then provide a way of shortening specifications. There are also, however, inherent dangers in such definitions and the utmost caution is required in order to avoid specifying either non-existent results or an unintentionally wide range thereof.

The danger of specifying a result which cannot be computed comes from the difficulty of checking, where many properties are stated, that they are non-contradictory. If the description is sufficiently complicated the contradiction may remain undetected until the attempt to construct an algorithm falters.

It is sometimes necessary to document only properties required of a permissable result in order to leave the implementation a degree of freedom (with which the user knows he can achieve his task). An incomplete list of specifications, on the other hand, can introduce unintended ambiguity as to what is required.

Evidence for the extreme difficulty of constructing axiomatic definitions of complex systems can be found in the attempts to provide axiomatic definitions of programming languages (e.g. ref /7/). In spite of the dangers a typical, informal, specification will mix required properties and constructive definitions in a rather rash manner.

The alternative to using implicit specification is to be constructive. That a constructive definition can be made far shorter and clearer than an implementation will be argued in the next section. However, the dangers of axiomatic methods should not force us to eschew their use entirely: where the saving is significant it is desirable to be able to define results implicitly. One of the best precautions, in so doing, is to limit the definitions to self-contained functions.

It is thus claimed that a specification language should be able to define results implicitly in order to achieve shorter descriptions. The organisation and care in use must be relied upon to avoid introducing contradictions or unintentional lack of precision.

4. ABSTRACTION

Abstraction is the most important aid the mind has for governing complexity. If we can identify what is crucial to a situation and leave aside details which are accidental, we will have reduced the amount of information we must hold or manipulate when considering that situation. The implicit definitions of section 3 are already an example of this in that the required input/output relation was separated from details of how to compute such a relation (Dijkstra refers to this as "Operational Abstraction").

The use of abstraction which is advocated in this section is the employment of "Abstract Objects" in specifications. In a sense the degree of abstraction is relative (e.g. an array in FORTRAN is an abstraction of the linear storage on which it is built). What should be sought in a specification is that the objects used are as close as possible to the objects inherent in the architecture of the system being specified. In other words, the objects of the specification should not possess any properties which are irrelevant to defining the function of the system. To give an example: the authors of the PL/I definition (ref /18/) tried to minimize the number of object types and avoided the use of sets. In representing sets as lists a number of problems were artifically created: testing for membership and element removal are rather tedious to define; much more crucially, the reader can only determine that the ordering is not used at some point in the definition by locating and checking all uses of such objects.

Finding the appropriate abstract objects on which to base a definition is essential to the provision of a short and clear definition: in this way an essentially algorithmic (i.e. constructive) specification can be written which avoids implementation details (see below for a more implicit view). Seeking such abstract objects is extremely difficult, it requires a full understanding of the key properties of the system under consideration. The search is almost invariably an iterative process in which the writer of the specification is forced to clarify his ideas.

The trivial example of using a set where no ordering property will be used in a collection of data has been mentioned; considering a compiler dictionary as a mapping from identifiers to data attributes is a more significant simplification; perhaps the most telling example is the use of "Abstract PL/I Programs". The PL/I definition (ref /1/) defines the meaning of an abstract tree form of PL/I programs. Such a tree is a normal form for many concrete PL/I programs whose differences are irrelevant to their meaning (e.g. placing of procedures, order of attributes, etc.) Defining the semantics in terms of such a normalisation avoids the proliferation of testing for the various forms. Furthermore the location of information is far simpler in this tree form than in a linear text. The relation of abstract programs to concrete is not difficult to document. But of more interest is the use for which the PL/I definition was intended: a mapping to machine code was to have been derived from the definition. This mapping would have also been based on abstract programs. The next stage of the plan was to use this mapping as a specification to develop the back-end of a compiler, where the front-end was to be taken from the PL/I Checker. Thus, for this task, abstract programs were being used as an abstraction of the "CTEXT" and dictionary of the Checker. This representation of PL/I programs was extremely complicated because of the desire to facilitate efficient access. To have specified the back-end mapping directly would have again proliferated this complexity through the entire document. Instead this specification can be deduced from the description of the relation from CTEXT to abstract programs (see ref /17/) and the documented mapping.

It has so far been argued that specifications can be simplified by the use of abstract objects. Clearly, then, a specification language must permit the use of such abstractions. Unfortunately, this is not the only requirement. The first point to be made is that the objects required will differ from one specification to another. So that, not only are the existing data types of for example PL/I or APL not rich enough, but also any attempt to provide a fixed list of objects with which an existing language should be extended will not be adequate for all purposes. It is necessary that a specification language should permit the definition of new objects (e.g. STG in ref /1/).

Some of the clearest examples of the use of abstract data are given in the work of Hoare (refs /8/, /6/). There is one way in which this author regrets the emphasis which comes from this work (and this leads to the second auxiliary requirement). Basically the presentation of new data objects is made via the Simula class concept. In the class definition one records the operations which are allowed for objects of this class, and one is then in a position to declare objects belonging to this class and to manipulate them with the given operations. The body of the class can then give realisations of the operations in terms of more basic objects and their operations. This gives a view of the process which is very like macro expansion (similar comments are valid for refs /14/, /3/).

But when an implicit specification for an operation states that the input/output variables should satisfy:

A(i,o) & B(i,o)

one would not expect to macro expand this into two programs which compute the sets of values satisfying A and B and then "expand" the "&" to an intersection operation! If we wish to combine the advantages of implicit definition of operations with those of abstract objects we must expect to find implicit specifications in terms of operations on an object, where the operations are never expanded! For example, in ref /11/ there is a stage of development where it is shown that if a set is computed such that:

LOWER $\subseteq$ x $\subseteq$ HIGHER

then x can be used to determine the final result of the program. However, the operation subset is never in any way coded in the final program.

(In ref /8/, this problem is recognised and overcome by the use of "recoding" cf. page 128).

This section has argued that in order to shorten specifications and to make them more comprehensible a specification language should contain a reasonable set of abstract objects (the notation should be close to that of mathematics to facilitate reasoning); have the ability to define new abstract objects; and encourage the implicit definition of operations using abstract objects.

There is one consequence of these conclusions that is considered by some people to be extremely unfortunate.

If objects, or indeed functions, exist which are implicitly defined, it is unlikely that an interpreter of the specification language can be constructed. Firstly even for the base set of objects the performance of the interpreter would be poor. For implicitly defined objects a model must be found. If such a model were found it would, of course, then give a particular interpretation which might be one of many. To this author, this is an indication that seeking interpretation at this stage is a mistake. The role of the specification is to define the range of possibilities. This range should be understood by the designer and it is precisely his task to find models (and to show their relationship to the specification). Of course, this is not an argument against having an interpreter for a subset of the specification language. The design process is then one of working towards efficient models of the specification which can be expressed in this subset. (For an extremely interesting approach to the problem of executing intermediate design stages see ref /5/.)

5. NOTATION

It is clear that a specification language must have sufficient expressive power to describe anything required in the specification (cf. Section 7).

VAB

The remainder of this section discusses choices which must be made in the design of a specification language in order to improve the usefulness of resulting specifications. One important distinction which should be kept in mind is the difference between reading and writing a specification. It has already been pointed out that seeking appropriate abstract objects for a specification can be extremely difficult. If the writer accomplishes his task the reader will have less difficulty in understanding the definition. (The difference also becomes apparent in the amount of training required to perform the respective tasks.)

It has been argued at several points above that existing (mathematical) notation should be used in a specification language where appropriate. There are, on the other hand, areas where concepts like those of programming languages are required (e.g. sequencing, iteration): in such cases it would seem appropriate for the intended audience to make the corresponding constructs look like those of programming languages. In order to satisfy the criteria of formality, it may be necessary to constrain their use.

A particular issue in relation to programming languages is whether a specification language should be purely functional. Experience suggests that, if this is so, either the argument lists become rather long or all possible objects are combined together into one component. In this latter case the distinctions as to what a function actually relies on or changes are lost. The most satisfactory solution appears to be to have a "State" into which are collected all of the system objects which are subject to change even at the lowest functional level. It is then a convention that this state can (like a global variable) be referenced by any operation. All other objects are explicitly handled as arguments. Even in this case it is worthwhile to distinguish those pure functions which do not depend on the state.

Apart from these points the usual rules of good language design (cf. ref /9/) should be followed. Care must also be given to the provision of supporting comments.

6. EXPERIENCE WITH SPECIFICATION LANGUAGES

Most of the ideas presented in this paper were first developed in the context of providing formal definitions for high level programming languages. Three versions of PL/I definitions were developed by the IBM Lab Vienna (for a good overview, and further references, see /15/). The notation used became known as "VDL" (Vienna Definition Language) and was applied to a number of other languages. Vienna, meanwhile, experimented with using the definitions as a basis for proofs of implementation correctness (e.g. ref /10/). Although this work was successful in that it showed such proofs to be possible, it became clear that certain problems existed which could only be resolved by changing the style in which the definitions were written (see ref /12/). The definition of PL/I provided in ref /1/ is an attempt to resolve these difficulties and subsequent work on using this style for implementation correctness is encouraging.

Turning now to systems other than languages, VDL itself was applied to experimental machine architectures. Other uses include /2/.

The description language used in /1/ has become known as "Meta-IV". Experiments were made on the application of Meta-IV to the description of machine and system architectures.

Another application of Meta-IV is the definition of a relational data base system (ref /4/): the work was based on the PRTV system of the IBM Scientific Centre in Peterlee, U.K.

7. DISCUSSION

This paper has not attempted to offer a complete answer to the need for a specification language (although it should be clear from the preceding section that this author believes "Meta-IV" is at least a basis for further work). Some of the technical problems which are known to exist should now be discussed.

Because of the danger that new application areas will require expression of new concepts, the completeness of expressive power is difficult to establish. One problem is known. One of the most difficult issues is the definition of systems which permit parallelism. In order to define programming languages like PL/I it was necessary to adopt a notation to indicate arbitrary merging. This part of Meta-IV is perhaps the least clearly defined and is the subject of ongoing work. But it is not likely that the rather primitive notation adopted will prove adequate for systems with richer merging and sequencing concepts.

A further specification language problem is the incorporation of other requirements into the model which defines the required function. It would, for example, be very attractive to use the same structure for simulation models which were used to specify/estimate the performance. This remains an open area.

There are a number of areas where the problems of making a specification language more attractive to users pose a technical challenge. The question of support systems has already been discussed; the role of other communication modes than linear strings of formulae has a bearing thereon. The enthusiasm shown by some people for HIPO charts suggests that providing a graphic way of presenting the same information content may make users more prepared to generate the required data. This leads to the concept of one system which contains a formal model with a number of different user interfaces.

The current paper began with a description of properties required of a specification (Precision, consistency, completeness, succinctness, organisation, comprehensibility) from this a number of requirements have been developed for Specification Languages (Formality, Implicit definitions, Abstract Objects, Clean Notation). It would be possible, perhaps, to go on and show how

Meta-IV is a realisation of these properties. It would however only be one such realisation (and one that is known to be imperfect).

REFERENCES

/1/ H.Bekic, D.Bjorner, W.Henhapl, C.B.Jones, P.Lucas: "A Formal Definition of a PL/I Subset", Parts I and II, IBM Lab Vienna, Techn.Rep.TR 25.139, Dec 1974.

/2/ A.Birman: "Correctness in Design: The S-Machine Experiment", Yorktown Heights Res.Lab, Techn.Rep RC-4193, Jan 1973.

/3/ A.Hansal: "Software Devices for Processing Graphs Using PL/I Compile-time Facilities", Information Processing Letters, 1974.

/4/ A.Hansal: "Formal Definition of a Relational Data Base System", Forthcoming IBM Techn.Report.

/5/ P.Henderson, R.A.Showdon: "A Tool for Structured Program Development", IFIP Proceedings Stockholm 1974.

/6/ C.A.R.Hoare: "Proof of Correctness of Data Representations", Acta Informatica, Vol.1, pp.271-281, 1972.

/7/ C.A.R.Hoare and N.Wirth: "An Axiomatic Definition of the Programming Language Pascal", Report of ETH Zurich, November 1972.

/8/ C.A.R.Hoare: "Notes on Data Structuring", in "Structured Programming", A.P.I.C. Studies in Data Processing, No.8, Academic Press, 1972.

/9/ C.A.R.Hoare: "Hints on Programming Language Design", SIGACT/SIGPLAN Conference on "Principles of Programming Languages", October 1973.

/10/ C.B.Jones and P.Lucas: "Proving Correctness of Implementation Techniques", in "Symposium on Semantics of Algorithmic Languages" (Ed.) E.Engeler, Springer-Verlag Lecture Notes in Mathematics No. 188, October 1970.

/11/ C.B.Jones: "Formal Development of Correct Algorithms: An Example Based on Earley's Recogniser", presented at ACM SIGPLAN Conference, SIGPLAN Notices Vol.7, No.1, January 1972 (also as TR 12.095).

/12/ C.B.Jones: "Formal Definition in Program Development", Springer Verlag Lecture Notes in Computer Science No.23, 1975, "Programming Methodology".

/13/ C.B.Jones: "Formal Definition in Compiler Development", IBM Lab Vienna, Techn. Rep., TR 25.145, Feb. 1976.

/14/ B.Liskov and S.Zilles: "Programming with Abstract Data Types", Project MAC report, CSG-99, Sept 1974.

/15/ P.Lucas and K.Walk: "On the Formal Description of PL/I", in "Annual Review in Automatic Programming", Vol.6, Part 3, Pergamon Press, 1969.

/16/ D.L.Parnas: "A Technique for Software Module Specification with Examples", Comm. ACM, May 1972.

/17/ F.Weissenböck: "A Formal Interface Specification", IBM Lab Vienna, Techn.Rep. TR 25.141, February 1975.

/18/ "PL/I BASIS/1-12", ECMA ANSI working document, July 1974.

VAB