BIT 22 (1982), 437 453

FORMALIZATION IN PROGRAM DEVELOPMENT

PETER NAUR

Datalogisk Institut, Sigurdsgade 41, DK-2200 Copenhagen N, Denmark

Abstract.

The concepts of specification and formalization, as relevant to the development of programs, are introduced and discussed. It is found that certain arguments given for using particular formal modes of expression in developing and proving programs correct are invalid. As illustration a formalized description of Algol 60 is discussed and found deficient. Emphasis on formalization is shown to have harmful effects on program development, such as neglect of informal precision and simple formalizations. A style of specifications using formalizations only to enhance intuitive understandability is recommended.

Keywords: programming, program development, program specifications, formalization.

1. Introduction.

In recent years several authors concerned with the methodology of program development have claimed that the use of formalized modes of expression in certain particular manners offers considerable benefits to the program development activity. The claims are related to a few key notions, most conspicuously formal specifications of programs and abstract data types. The present study is an analysis of some of the arguments given in support of these claims. It will be found that several of these arguments are invalid, and it will be maintained that the emphasis on formalization in program development may in fact have harmful effects on the results of the activity.

The present analysis is not an attempt at a treatment of the complete literature related to it. It will deal with only a few selected contributions, chosen so as to cover both the discussion of general principles and their illustrative application to specific instances of program development. General principles will be discussed primarily as they are expounded by Liskov and Zilles [11] and by Jones [9]. Applications will be taken from Henhapl and Jones [7] and Liskov [10].

2. Specifications, formalization, and mathematics.

In this essay specification will mean a description of a problem to be solved with some aid from a program running on a computer, the specification to

Received June 28, 1982.

be produced by a programmer in addition to the program. This definition of specification is explicitly designed to exclude descriptions of a form that is used to control the computer directly, by means of automatic translation or interpretation, the reason being that in the terminology employed here such a description is a program. The present definition of specification is an attempt to capture the notion given to this word by Jones [9] and by Liskov and Zilles [11], but identity of meaning cannot be assured owing to their addition of constraints to the notion of the word, such as the requirement that the description be formal, in some sense.

The arguments for the program development techniques to be considered here are based to a large extent on claims about the techniques employed in mathematics and about techniques characterized as "formal". Thus Liskov and Zilles [11] on page 6 say: "Formality. A specification method should be formal, that is, specifications should be written in a notation which is mathematically sound ... formal specification techniques can be studied mathematically ..." Jones [9] on page 8 writes: "This book shows how precise specifications for programs can be written for data-processing problems. To achieve this, formal definition methods are explained and then shown to provide a basis for precise, though not fully formal, specifications. A similar approach is taken with correctness arguments. For these to be sound (or rigorous) the writer needs to understand what would comprise a formal proof." Goguen [6] expresses himself similarly.

In spite of this stress on formalization, none of the authors quoted makes any attempt to clarify what it is to be formal. As an approach to clarifying this concept, here are the relevant explanations given in The Oxford Concise Dictionary:

Formal: concerned with the form, not the matter, of reasoning; valid in virtue of its form, explicit and definite, not merely tacit.

Formalize: give definite shape to; make precise, or rigid; imbue with formalism.

These definitions are too vague to be used as basis of a discussion of the formalization in program development referred to in the quotations above. In these contexts the concept appears to be understood in a more specific sense, essentially this:

Formal: expressed purely by means of symbols given a specialized meaning.

Here the restriction to specialized meaning is intended to exclude the symbolism used in natural language writing. Even with this clarification, however, the formality or otherwise of many common forms of expression, for example numerals expressed in decimal notation, is unclear according to the definition. This vagueness of the concept seems to be unavoidable, and accords with the main point of the following discussion, which asserts that the formal mode is merely an extension of the informal one, not a replacement of it.

On the other hand, the special notations used for example in arithmetic and in so-called formal systems of mathematical logic are clearly formal according to the definition.

It will here be argued that the uses of the word formal in the contexts formal specification and formal proof of the correctness of a program rest on a misunderstanding of the nature of formalization and therefore are misleading. In essence what Liskov and Zilles [11] and Jones [9] suggest is that the full meaning of an expression given in an informal mode can be conveyed by a formal expression. What will be argued here is that in reality the meaning of any expression in formal mode depends entirely on a context which can only be described informally, the meaning of the formal mode having been introduced by means of informal statements.

The dependence of formal expressions upon informal ones holds in mathematics as much as anywhere else. As illustration, consider the following passage from Gauss [5], appearing as part of a proof to which he himself attached extraordinary importance, his third proof of the law of quadratic reciprocity: "We can shorten the following discussion considerably by introducing certain convenient notations. Let the symbol (k, p) represent the number of products among

$$k, 2k, 3k, \ldots, \frac{p-1}{2}k$$

whose smallest positive residues modulo p exceed $p/2$. Further if x is a non-integral quantity we will express by the symbol $[x]$ the greatest integer less than x so that $x-[x]$ is always a positive quantity between 0 and 1. We can readily establish the following relations:

$$I.\ [x]+[-x] = -1."$$

This passage says clearly that the reason for introducing the formalizations is brevity and convenience. Here and elsewhere in the proof the formalizations are used in the first instance as convenient ways of stating certain facts. In some cases the derivation of a new fact from given ones can be accomplished by purely formal manipulation, but for the most part the derivations depend directly on the reader's intuitive understanding of the facts, whether these are expressed formally or informally.

The use of the formal mode, on the other hand, clearly has nothing to do with the validity of the various steps in the argument. Throughout the proof Gauss passes back and forth between the two modes with perfect ease. However, this ease cannot conceal the pre-eminence of the informal mode. As a verification of this point one might make the experiment of rewriting Gauss's proof using exclusively one or the other of the modes. This would show immediately that a purely informal formulation would be long and inconvenient, but perfectly possible, while a purely formal one could not conceivably make sense.

The pre-eminence of the informal mode of expression, with particular regard to program development, has already been asserted very strongly by Zemanek [16] on page 14: "No formalism makes any sense in itself; no formal structure has a meaning unless it is related to an informal environment [...] the beginning and the end of every task in the real world is informal." This holds as much in mathematics and automatic theorem proving as anywhere else. Even in the most formalized mathematical argument the justification of each argument step in terms of a rule of inference must finally rest with the author's intuitive, informal acceptance that the rule applies and has been used correctly. In an automatic theorem prover the validity finally rests with the program author's intuitive, informal acceptance that the actions of his theorem prover match the adopted rules of inference properly, or with the user's equally intuitive and informal trust that the author has realized the rules that the user wants to employ.

Instead of regarding the formal mode of expression as an alternative to the informal mode we must view it as a freely introduced part of the basic informal mode, having sometimes great advantages, mostly for the expression of highly specialized assertions, but having also great disadvantages, first and foremost in being limited to stating facts while being inapplicable to the many other shades of expression that can be reached in the informal mode. With this view of the formal mode of expression many of the turns of phrase employed by Liskov and Zilles [11] turn out to be logically improper and therefore misleading. Thus "formal specification" and "formal proof" suggest a contrast to "specification" and "proof" which does not exist. At most one might distinguish between specifications or proofs that make more or less use of formal modes of expressions. It follows that the explanation of the first criterion for evaluating specification methods in Liskov and Zilles's section 1.2, which says that "A specification method should be formal ... This criterion is mandatory if the specifications are to be used in conjuntion with proofs of program correctness", is meaningless. It also contradicts the explanation on the previous page which suggests that "code reading becomes an informal proof technique." This whole contradistinction between formal and informal techniques is a discussion of a pseudo-problem.

For similar reasons the whole argument in [11] that purports to justify the need for a separate specification that somehow makes the connection between what is called a concept and a corresponding program is void. Assertions may be proved true whether they are expressed informally or formally, and it is perfectly possible to prove that the result of a program satisfies a requirement stated informally. In fact, many of the proofs of the properties of programs given in such a treatise as Aho, Hopcroft, and Ullman [1], conform to this pattern, which again is quite similar to the usual mathematical style, as illustrated in the quotation from Gauss given above. Thus the claim of Liskov and Zilles that formal specifications, whatever they are, are necessary as basis for proofs of programs cannot be substantiated.

3. Correctness and abstraction in programming.

The dominating reason given for insisting on formalization in program development is that supposedly formalization is necessary for proofs of correctness, and correctness supposedly must have absolute priority over any other concerns. Having already dismissed the first part of this argument it might seem less important to consider the second part of it. However, as an illustration of the argumentation used in support of formalization the question of program correctness and its importance is in need of examination.

Liskov and Zilles [11] say on page 2: "Although we are coming to realize that correctness is not the only desirable property of reliable software, it is surely the most fundamental: If a program is not correct, then its other properties (e.g., efficiency, fault tolerance) have no meaning since we cannot depend on them." This is a remarkable statement since it is obviously proved false every day, in large scale, in probably most of the activities in which computers are used. Surely many programs that are used have errors in them, and are thus not correct, but even so provide useful service, and their efficiency and fault tolerance are issues of meaningful concern.

The claim that correctness of programs is an issue of absolute priority seems to be connected with the adoption of a scale of values that also attaches high importance to what is called abstraction, while it tends to dismiss as unimportant and trivial such issues as the format of data used in interfaces to users. In order to see some of the manifestations and consequences of such a scale of values, consider the example of a specification of a program, with particular illustration of the manner in which input and output will be described, as given by Jones [9], chapter 19 Input/Output Statements. Jones in his figure 79 gives a specification of a program for producing as an ordered output list the first n primes, where n must be given in the input to the program. The figure gives the specification first at the highest level, consisting of eight lines of formulae, and then decomposed into more detailed descriptions, consisting of 18 lines. Most of the substance of the description is concentrated in two lines of the description at the highest level:

$$\textbf{let } n = \textbf{hd } il$$

$$\textbf{elems } ol = \{i | 1 \leqq i \leqq n \wedge \textit{is-prime}(i)\} \wedge \textit{is-ordered}(ol)$$

These lines refer to an input list, *il*, and an output list, *ol*, and express that when the program has finished execution, if n denotes the head element of the input list then the output list is such that the set of its elements is the same as the set of the primes between 1 and n, and in addition the output list is ordered.

This specification reflects very clearly a scale of values that gives high priority to such abstract characteristics of the output as its inclusion of all primes, its exclusion of non-primes, and its being ordered, while such issues as the output medium, the line and page format, and the number representation, are ignored completely. Such a scale of values can only be maintained consistently if the responsibility of the program designer is defined so as to embrace only the strictly computer controlled part of the activity, while excluding the part of the data processing that includes interaction with humans. If a wider responsibility is assigned to the program designer this scale becomes unacceptable. As illustration, consider a solution of the production of primes that produces output in a smudged, printed form that makes it difficult to distinguish the digits 3, 5, 6, 8, and 9, or one that uses octal or hexadecimal number representation. Let us be concerned with the perception of this output by a human, and in particular the error rate of that perception. In this situation the correctness of the internal production in the computer might be less important than the characteristics of the physical form of the output.

This reflection points to a deep inconsistency in the arguments of those who in the interest of reliability advocate the use of so-called formal specifications and at the same time insist on concentrating on what is called the abstract aspects of the problems. These arguments must assume that reliability is highly dependent on the forms of expressions used in interfacing with those people who do the program development, while it is independent of the forms used in interfacing with the eventual users of the programs. Clearly such a perspective on program development leaves important reliability issues uncovered.

4. Notes on a description of Algol 60.

As one example of the results of following the methods based on so-called formal specifications, the description of Algol 60 given by Henhapl and Jones [7] (referred to as the HJ-report in what follows) will in this section be subject for some critical analysis. The HJ-report will be compared with the official description of the same language given in the Modified report on the algorithmic language ALGOL 60 [4], referred to as the Mod-report in what follows.

As the first point in the comparison of the two reports, the HJ-report presents itself not as an independent description of a programming language but as a demonstration of a description technique. Thus the introduction of the Mod-report, which places the language in the context of history and related techniques and gives an overview of its characteristics, is replaced in the HJ-report by one that accounts for special technical points in which the description deviates from earlier related descriptions. Because of this difference in aim of the two descriptions the following comparisons will be confined to those parts of the two reports that describe the same matters.

As the second point of the comparison, the Mod-report includes examples of uses of the language that are not found correspondingly in the HJ-report. A fair comparison of the sizes of the descriptions will therefore have to omit the examples from the Mod-report.

Third, the formal notation used for the description of the language is defined in the Mod-report as section 1.1 having 32 lines, while the corresponding basis of the HJ-report must be sought elsewhere, for example in Jones [8], a description of 60 pages. While the enormous difference in size of this background material must be kept in mind, both descriptions will be omitted from the size comparison below.

Fourth, with the omissions stated in the previous points, the sizes of the two descriptions of the language, counting lines of formal and informal mode statements equally, are as follows:

Mod-report	1310 lines
HJ-report	1110 „

Fifth, as the first substantial item in the comparison of the overall characteristics of the two language descriptions, the HJ-report does not define the language Algol 60, but a language, let us call it VDMgol, defined by a so-called abstract syntax. The relation between the language Algol 60 and VDMgol is not specified, indeed is only hinted at in section 1.2 of the HJ-report. This means that the number of lines of the HJ-report given above is entirely misleading, since they correspond to only a fragment of the description of Algol 60. There can be no doubt that a complete description of the language obtained by completing the HJ-report would be considerably longer than the Mod-report.

Sixth, the formulations of section 1.2 of the HJ-report, which are informal notes concerning the relation between Algol 60 and VDMgol, are remarkable for their unclarity and ambiguity. They rely heavily on the reader's thorough background knowledge of Algol 60 and on his willingness to guess an intended meaning behind obscure phrases. Thus they mostly leave it unclear whether a word such as expression, block, or body, refers to one or the other of the two languages, and they refer to the body of a for statement, a concept defined in neither of the languages.

The relation between the descriptions of language details in the HJ-report and the Mod-report will be illustrated here only by some examples taken from the part of the HJ-report reproduced in figure 1. This part of the HJ-report is the first one to present actual language rules. It describes in lines 1 to 9 the restrictions that must be satisfied by programs, and gives in lines 10 to 23 part of the logic that goes into the check of blocks. The following notes are brief indications of details of figure 1 that either in themselves or in a comparison with the Mod-report are incorrect or problematic. In this comparison many issues are uncertain, owing to the lack of a description of the relation between Algol 60 and VDMgol.

(1) Line 2 is incompatible with the Mod-report or is misplaced, depending on what is held in the *s-oid* fields of type and array declarations, a matter left

```
Line
 1    is-wf-program(mk-program(b)) =
 2        /* for all type-decl, array-decl's within b, their s-oid is unique */ &
 3        (let oads = {d|within (d, b) & is-array-decl(d) & s-oid(d) ≠ NIL}
 4        /* all expressions in s-bdl of elements of oads are integer constants */) &
 5        (let env = [n ↦ mk-type-proc(INT)|n ∈ Int-funct-names] ⋃
 6                   [n ↦ mk-type-proc(REAL)|n ∈ Real-funct-names] ⋃
 7                   [n ↦ PROC|n ∈ Proc-names]
 8        is-wf-block(b, env))
 9    type: Program → Bool
10    is-wf-block(mk-block(dcls, stl), env) =
11        let labl = /* list of all labels contained in stl without an intervening block */
12        is-uniquel(labl) &
13        is-disjoint(⟨elems labl, {s-id(d)|d ∈ dcls}⟩) &
14        (let renv = env\{s-id(d)|d ∈ dcls}
15        let lenv = [s-id(d) ↦ (cases d:
16                   mk-type-decl(,,tp)          → tp
17                   mk-array-decl(,,tp,)        → mk-type-array(tp)
18                   mk-switch-decl(,)           → SWITCH
19                   mk-proc-decl(,PROC,,,,)     → PROC
20                   mkproc-decl(,tp,,,,)         → mk-type-proc(tp))
21                             |d ∈ dcls] ⋃
22                      [lab ↦ LABEL|lab ∈ elems labl]
23        let nenv = renv ⋃ lenv
```

Fig. 1. Excerpt from Henhapl and Jones [7], section 2.1.

to be guessed by the reader. If the *s-oid* fields hold the original identifiers in the case of own declarations then the uniqueness is inconsistent with the Mod-report. If, on the other hand, the *s-oid* fields hold internal identifiers generated in the translation from Algol 60 to VDMgol then the rule has nothing to do with the language Algol 60, being merely a requirement on the translator.

(2) Lines 3 to 4, requiring the bound expressions of own array declarations to be integer constants, should be compared with the following sentence from the Mod-report, section 5.2.4.2: "The bounds of an array declared as *own* may only be of the syntactic form integer (see Section 2.5.1)." An examination of what is meant by integer constant in the HJ-report and integer in section 2.5.1 in the Mod-report shows that they cannot be identical, as demonstrated by some examples:

Bound expression	Accepted by Mod-report	Accepted by VDMgol
(34)	No	Yes
−45	Yes	No

(3) Lines 11–12 check that no identifier is used more than once as a label in one block, and line 13 checks that no label identifier is otherwise declared in the block. However, there is no check that an identifier is not declared explicitly more than once. For this reason lines 15 to 21 may form an inconsistent

mapping. What lines 11 to 13 of figure 1 thus fail to convey properly is expressed in the Mod-report, section 5, as follows: "No identifier may be declared either explicitly or implicitly (see Section 4.1.3) more than once in any block head."

(4) In line 14 the mapping *renv* is formed from the global environment by removing the items corresponding to identifiers that are redeclared in the block. However, items corresponding to identifiers used as local labels are not thus removed. For this reason the operation used in line 23 is incompatible with the specification language.

Other remarks on the formulations used in figure 1 are made in section 5 below.

The remarks above are concerned only with a small fraction of the 1110 lines of the HJ-report. The remaining part of the report has not been similarly analyzed. However, even the present limited enquiry provides examples that indicate the problematic nature of several of the claims made in support of the use of formalization. First, items (3) and (4) above both indicate flaws in the purely formal part of the HJ-report that, in addition to failing to describe Algol 60 properly, entail formal inconsistencies related solely to the rules of the description language employed. Thus, in spite of the insistence that formalization should be employed in order to make proofs possible, the consistency of the formulations of the HJ-report cannot have been proved by its authors. As to the arguments employed in the present discussion, they have been based on the intuitive understanding obtained by means of careful reading of the formulae and thus are quite independent of the fact that some of the matters involved have been expressed formally.

In connection with the statement (1) above it is relevant to compare directly with the corresponding formulation of the Mod-report, section 5.1.3: "A variable declared **own** behaves as if it had been declared [...] in the environmental block, except that it is accessible only within its own scope. Possible conflicts between identifiers, resulting from this process, are resolved by suitable systematic changes of the identifiers involved." In discussing this passage, which incidentally was added to the Revised Report on the Algorithmic Language ALGOL 60 [3] when forming the Mod-report, it must be noted that strictly speaking the last sentence is unclear since "this process" is undefined. The sentence seems to have been formed by thoughtless paraphrasing of section 4.7.3.3 of the same report. Presumably the intended meaning is that which is expressed if the first part of the passage is rewritten so as to define the meaning in terms of a process of program transformation. With this understanding of the passage, the meaning of **own** is explained by means of a virtual program transformation. Line 2 of figure 1 appears to be an attempt to express this notion formally, and the flaw noted in statement (1) indicates the failure of this attempt.

In summary, the description of Algol 60 given in Henhapl and Jones [7] fails to confirm the claims made for the advantages of using a formal specification language on all counts. Compared with the standard description of the language the more formal description is quite incomplete with roughly the same size; an examination of a small part of it has revealed numerous errors and inconsistencies, and the proof of its consistency appears to be impossible, or at least so impractical that it has not been done by its authors.

5. Neglect of informal precision and disdain of intuition.

While the development and use of formal modes of description in itself are at worst harmless, some of the current discussion and argumentation that tries to promote the use of formal modes most likely is harmful to effective program development. As the most prominent harmful effect, the claim that formal modes can be regarded as a superior alternative to informal ones inevitably will influence the authors arguing for formal modes into suggesting or implying that there are no such things as principles and practice of good informal expression, or, at least, that such matters are not worthy of attention.

Examples of unclear informal formulations can be found abundantly in the writings of the authors who make a special point of the alleged superiority of the formal mode. Thus much of the criticism of the argumentation in Liskov and Zilles [11] presented above can be viewed as directed against inconsistent use of words such as "formal", "proof", and "specification". Examples of unclear informal formulations that are directly involved in specifications of programs can be found in Liskov [10], section 3.1 Problem Specification, an excerpt of which is shown in figure 2. The section gives in one page an informal, high-level specification of a text formatter, supposedly as it might result from an analysis of the initial problem description. Quite apart from the fact that, as may be seen from sentences S6 and S12 of figure 2, this specification is heavily oriented towards sequential processing and thereby in direct contradiction to the idea of a specification of "what" and not "how" which it is supposed to illustrate, it gives rise to many questions of detail. Thus "space" is used ambiguously, in S4, S6, and elsewhere, to denote a character, in S16 to denote whatever is between words, and in the obscure sentence S5 to denote, one may guess, position. The order of items explained is confusing; the paragraph from

(S1) The output text is indented 10 spaces from the left margin, and is divided into pages of 50 text lines each. (S2) A header of 5 lines, including a line giving the page number, is output at the beginning of each page.

(S3) An input text line consists of a sequence of words and word-break characters. (S4) The word-break characters are space, tab, and newline; all other characters are constituents of words. (S5) Tab stops are considered to be every eight spaces.

(S6) The tabs and spaces are accumulated in the current output line along with the input words. (S7) Thus, if two spaces occur in the input between two words and those words appear on the same output line, then they should be separated by at least two spaces. (S8) If the two words appear on different output lines, the spaces between them need not be preserved.

(S9) The formatter has two basic modes of operation. (S10) In "nofill" mode, each input text line is output without modification. (S11) In "fill" mode, output lines of 60 characters are produced. (S12) Input is accepted until no more words can fit on the current output line. (S13) Newline characters are treated essentially as spaces. (S14) The line is then justified as described below. (S15) Initially, the formatter is in "fill" mode.

(S16) Justification is performed by enlarging spaces between words, as evenly as possible, until the last word has its last character in the rightmost position of the line.

Fig. 2. Excerpts from Liskov [10], section 3.1, with sentence numbers added.

S6 gives certain rules about treatment of tab and space characters in the input, while the following paragraph, from S9, tells that the formatter has two modes of operation, and that in one of these the previous paragraph does not apply (presumably, the matter is not clear since the meaning of "modification" in S10 is not given). The attentive reader will become frustrated from lack of clear answers to simple questions; for example, are the 10 spaces of indentation mentioned in S1 considered to be a modification as in S10, and thus not produced in "nofill" mode? Are these spaces included in the 60 characters produced as an output line in the "fill" mode according to S11? Are the 5 lines of the header mentioned in S2 included in the 50 text lines that form a page according to S1?

As other examples of negligent informal formulations made by authors who make a point of advocating the use of formal modes, consider the excerpt from Henhapl and Jones [7] given as figure 1. Line 11 is defined in terms of "all labels contained in stl without an intervening block". Here the word "label" is used inconsistently; the abstract syntax definition of the structure of "stl" contains an item called label only as the operand of go-to-statements, while what must be meant to make sense at all in the context are labels appearing before colons before statements, identified in the abstract syntax, one may venture to guess, by the selector s-lp. Again, the phrase "contained in stl [a statement list] without an intervening block" is likely to make sense only to readers with a good background knowledge of Algol 60.

Further examples of unclear informal formulations in Henhapl and Jones [7] are found abundantly in their section 1.2, as already pointed out above.

The weaknesses of informal formulations shown as illustration above are caused predominantly by violations of a very elementary principle of expression. In a nut's shell the principle is that one should choose one's designations carefully and stick to them. Spelled out in more detail, the principle says

(1) that one should choose each specialized word and symbol carefully so as to identify unmistakably a concept that is useful and relevant in the context,

(2) that one should introduce each concept and designating word or symbol in the documentation where they are used, and

(3) that throughout the documentation each concept introduced must be referred to invariably by the same designation.

Obviously this principle is the basis of the use of symbols in mathematics. However, it is clearly just as important to clarity in informal expression. It is neither profound nor difficult to apply. When one finds that it nevertheless is grossly neglected by authors who wish to propagate formal modes, then it is hard to escape the conclusion that this neglect is influenced by the desire to demonstrate the alleged superiority of the formal mode.

Hand in hand with the claim that formal modes of expression are superior to informal ones goes an insistence that informal modes depend on unreliable intuitition, without any mention that the same holds for formal modes. Liskov and Zilles [11] even talk of a superior "formal understanding", whatever that may be. In the view presented here, neglect of the quality of informal expression and disdain of intuition amount to intellectual suicide. As already stated above, any description of real matters of this world, and not just such partial aspects of them that have already been isolated so as to allow formal description, must fundamentally be based on an informal mode. If the informal basis is not clear and consistent then there is no hope for the relevance or usefulness of the description, irrespective of whether it makes use exclusively of an informal mode or also makes use of formalizations. And any argument, no matter how formally expressed, depends in the final analysis on intuitive insight. Intuitive insight comes in many degrees of certainty and doubt, and may actually be mistaken, but this does not imply that it has to be vague or ambiguous, nor that it is beyond discussion and refinement. In fact, an important part of any scientific activity is to express, discuss, criticize, and refine, intuitive notions. In this activity formalization may be very helpful, but it is, at best, just an aid to intuition, not a replacement of it.

6. Neglect of simple formalizations.

The misguided notion that a formal mode of expression can replace the informal mode entirely seems to be also the background of the special stipulation put on the use of formal modes by Liskov and Zilles [11] when they say that "specifications should be written in a notation which is mathematically sound" and that "the syntax and semantics of the language in which the specifications are written must be fully defined." With these demands formalization is made an aweinspiring, heavy tool that can only be employed by those who have been initiated into the appropriate mathematical methods.

Faced with these high demands one may first wonder whether the actual uses of formalization that one finds in the literature satisfy them. Does Gauss's introduction of the notation $[x]$quoted above qualify? Does the introduction of the formalism for syntactic description used by Backus [2] and subsequently taken over in the Algol 60 report qualify? It has 16 lines of informal prose, one example of a metasyntactic formula, and four examples of symbol sequences generated by it. Does the introduction of the Pidgin ALGOL employed by Aho, Hopcroft, and Ullman [1] throughout their book satisfy the demands? On page 34 they say explicitly that "No attempt is made to give a precise definition, as that would be far beyond the scope of the book."

In the view taken here Liskov and Zilles's demand for full definition of the notation used in specifications is an unjustified take-over from the area of programming languages, where the demand is justified because such languages are designed for general use. The demand not only goes far beyond much established practice, but may have a harmful effect by discouraging useful formalization of limited scope, for special purposes. If we regard formalization, not as an overall replacement of the informal mode, but as a result of a freely made extension of the informal mode, then it will be obvious that one and the same description may employ any number of different formal notations side by side without contradiction, and there will be no objection to introducing a formal notation for a special purpose and defined only so as to make the meaning of that particular use of it clear. This is the attitude adopted by Aho, Hopcroft, and Ullman, who, as the continuation of the quotation above, say: "It should be recognized that one can easily write programs whose meaning depends on details not covered here, but one should refrain from doing so."

The unfortunate consequences of insisting on using only formalizations that are supported by heavy mathematical machinery and that supposedly replace the informal mode entirely are clearly visible in the illustration given by Liskov [10]. In the concluding discussion of that report it is said that "we have experimented with writing formal specifications and have found them very error prone and difficult." The reason for this experience can only be that the kinds of formalizations that have been tried have been inadequte. In fact, as already pointed out above the informal descriptions given in the report are quite unclear and incomplete. They could undoubtedly be improved considerably by the addition of a few simple formalized descriptions, thus in section 3.1, Problem Specification, quoted in part in figure 2, the handling of the various character classes in the input might be stated precisely by means of a table corresponding to viewing the input analysis as a finite-state machine. Similarly in figure 3, Partial Specification of Streams, the description might be clarified by the addition of a table showing the reaction of a stream in dependence on the possible states of a stream along one dimension and the relevant operations along the other. Such tabular descriptions are the obvious means for helping to assure that in a certain situation all cases, or all combinations of cases, are considered and treated properly. Often they can be used directly as the basis for efficient and clear programming (see, e.g. Naur [12], p. 88–95, and Naur [13]). That all of these advantages are disregarded, with no compensating gain whatsover, is a telling comment on some of the argumentation and thinking going on around the use of formalization.

7. Psychology of formalization and specification.

With the view on formalization expressed above, the choice between informal and formal modes becomes, not a question of what can be expressed or the validity of what is expressed, but a matter of ease and effectiveness for author and reader, and thus to a large extent a psychological question. On the ease of the use of formalization no generally valid claim seems possible. Bertrand Russell, well known for his highly formalized studies of the foundations of mathematics, says in [15]: "... a good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher." On the other hand in [14], page 76, he says: "It is not easy for the lay mind to realize the importance of symbolism in discussing the foundations of mathematics, and the explanation may perhaps seem strangely paradoxical. The fact is that symbolism is useful because it makes things difficult." On this background there is reason for scepticism when Liskov and Zilles [11] claim that "because it is difficult to construct specifications using informal techniques, such as English, specifications are often omitted, or are given in a sketchy and incomplete manner. Formal specification techniques [...] provide a concise and well-understood specification or design language, which should reduce the difficulty of constructing specifications."

As an approach to clarifying the ease or difficulty of using formal modes, let us consider some concrete examples. If we have to multiply two hundred and seventy four by three hundred and forty one, most of us would immediately rewrite the problem formally as 274×341 and then proceed from there using some kind of formal manipulation. The obvious ease of the use of the formal mode in this case should not make us overlook, however, that this ease depends partly on the problem being of a highly specialized kind for which very effective formal tools have been developed, and partly on our having been trained in the use of these tools.

As a concrete example of formalization directly relevant to programming, consider the writing of programs in a programming language such as Fortran or Pascal. Probably most users would agree that for some purposes, such as describing certain arithmetic calculations, these languages make the required formalization easy, while for other purposes, such as describing the generation of elaborate reports as output, they impose difficulties. Thus it is not the particular form of formal description employed in programming languages that gives difficulties, but the requirement to express particular kinds of actions using particular given forms of expression. Clearly this consideration must have been a driving force in the development of the programming language Cobol during the years following its first introduction, characterized by the addition of built-in formalizations of actions such as sorting and report generation.

On the background of these examples one may suggest that for a particular kind of formalization to be convenient and helpful it must first have been designed specifically so as to cover a particular class of problems in a manner that is convenient to the people who have to use it, and, second, these people must have been trained in its use. It must further be inferred that in so far as established programming languages, such as Fortran and Pascal, are felt to be inconvenient to users who are well familiar with them, they must fail to cover certain classes of problems adequately.

This kind of failure of established programming languages seems to have arisen in at least two different manners: (1) the wealth and variety of the problem areas that are of significant practical interest in present-day programming are so large and change so rapidly that adequate coverage by a single programming language is incompatible with keeping the language small and economical; (2) the form of expression predominantly employed in the established programming languages, essentially strings generated by productions, excludes several useful forms of expression, such as tables laid out in two dimensions.

Faced with these difficulties in expressing required solutions using established programming languages, the advocates of so-called formal specifications offer what is claimed to be a help to the programmers. According to these ideas the programmer must first express the solution in a so-called formal specification language, second express the same solution in a programming language, and third prove that the two solutions are equivalent, in some sense. From the discussion above it should be clear that in the view presented here this approach offers no help to the programmer, but only adds to his or her burdens. Indeed, the two manners in which programming languages pose difficulties in covering problems may be expected to work in largely the same way with any given, pre-defined specification language. Thus with this approach the programmer has to produce not one but two formal descriptions of the solution, and in addition has to prove that they are equivalent.

From this discussion it follows that while formalization under certain circumstances is very helpful to the people who employ it, the formalization required when using any one given language for formal specifications cannot be expected to yield any particular advantage, but rather will be a burden on programmers.

8. Specifications for human understanding.

While the discussion above must conclude in an invalidation of many of the arguments given for using formalization in particular manners in developing programs, it is not an attempt to argue for abolishing specifications or for avoiding formal modes. What will be argued is that for effective program development the criteria for selecting the form of specifications and for the use of formalization should be quite different from those advocated by the authors discussed above, and that consequently the kinds of specifications of programs that should be advocated are also quite different. According to the views adopted here specifications are sometimes a necessary evil, to be used for documentation of such aspects of programs that are not satisfactorily documented by the programs themselves. When specifications cannot be avoided they should be designed to bridge the gap between the user's intuitive understanding of each aspect of his or her problem and its solution on the one hand, and the programmed solution on the other. In general a program will have to be supported by part-specifications of several different forms, corresponding to its various aspects. The ideal specification of an aspect of a program is a description of what that aspect does such that on the one hand it is intuitively obvious to the user that it corresponds to his or her requirements, and on the other hand it is equally obvious to the programmer that it is realized in the program itself. In either case the intuitive understanding may need to be supported by an argument having several steps, a proof, but clearly it is preferable if such is unnecessary.

From this view of specifications it follows that the criteria for choosing the style and mode of each part-specification must combine ready understandability with the power to express whatever problem aspects are relevant. In addition the criteria must include clauses that ensure the ready, effective realization of the solutions described in terms of the programming language to be used. It follows that the most suitable form of a part-specification must depend on a variety of factors, including personal and environmental ones, to such an extent that the forms must be considered an open range of possibilities, unrestricted by any closed system of notions and techniques. Each part-specification must be connected to the environment by informal explanations, but may employ formalizations of any suitable form, such as tables for enumerating cases, decision tables, tables or graphs corresponding to finite-state algorithms, program skeletons, formulae of any kind, etc. A part-specification may employ a well-established description technique, or it may employ description forms that have been introduced specifically for a particular purpose.

When this is said it should be clear that these criteria correspond closely to the style and technique used in practice for traditional technical documentation. This, then, is the conclusion of the present discussion, that programs should be supported and specified by documentation of any kind, the overriding concern in producing this documentation being clarity to the people who have to deal with it. For achieving clarity any formal mode of expression should be used, not as a goal in itself, but wherever it appears to be helpful to authors and readers alike.

REFERENCES

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. J. W. Backus, *The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference*, Proc. International Conf. on Information Processing, UNESCO, 1959, pp. 125–132.
3. J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur (ed.), A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden and M. Woodger, *Revised report on the algorithmic language ALGOL 60*, Comm. ACM 6, 1 (1963, January), 1–17; Computer Journal 5, 349–367; Num. Math. 2, 106–136.
4. R. M. de Morgan, I. D. Hill and B. A. Wichman, *Modified report on the algorithmic language ALGOL 60*, Computer Journal 19 (1976), 364–379.
5. C. F. Gauss, *Theorematis arithmetici – demonstratio nova, Commentationes Societatis Regiae Scientiarum Gottingensis*, Vol. 16, Göttingen, 1808; *Werke*, 1876, Bd. 2, pp. 1–8; *English translation in D. E. Smith, A Source Book in Mathematics*, Vol. 1, Dover, New York, 1959, pp. 112–118.
6. J. Goguen, *Thoughts on specification, design and verification, ACM SIGSOFT Software Engineering Notes*, Vol. 5 no. 3 (1980, July), 29–33.
7. W. Henhapl and C. B. Jones, *A formal definition of ALGOL 60 as described in the 1975 Modified Report*, in D. Bjørner and C. B. Jones (eds.), *The Vienna development method: the meta-language*, Springer, Lecture Notes in Computer Science 61, Berlin – Heidelberg – New York, 1978, pp. 305–336.
8. C. B. Jones, *The meta-language: a reference manual*, in D. Bjørner and C. B. Jones (eds.), *The Vienna development method: the meta-language*, Springer, Lecture Notes in Computer Science 61, Berlin – Heidelberg – New York, 1978, pp. 218–277.
9. C. B. Jones, *Software development: a rigorous approach*, Prentice-Hall, Englewood Cliffs,New Jersey, 1980.
10. B. Liskov, *Modular program construction using abstractions*, in D. Bjørner (ed.), *Abstract software specifications*, Springer, Lecture Notes in Computer Science 86, Berlin – Heidelberg – New York, 1980, pp. 354–389.
11. B. Liskov and S. Zilles, *An introduction to formal specifications of data abstractions*, in R. T. Yeh (ed.), *Current Trends in Programming Methodology*, Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1977, pp. 1–32.

12. P. Naur, *Concise survey of computer methods*, Studentlitteratuur, Lund, Sweden, 1974.
13. P. Naur, *Control-record-driven processing*, in R. T. Yeh (ed.), Current Trends in Programming Methodology, Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1977, pp. 220–232.
14. B. Russell, *Mysticism and logic*, Penguin, Harmondsworth, England, 1953.
15. B. Russell, *Introduction, in L. Wittgenstein, Tractatus logico-philosophicus*, Routledge and Kegan Paul, London, 1922.
16. H. Zemanek, *Abstract Architecture*, in D. Bjørner (ed.), Abstract software specifications, Springer, Lecture Notes in Computer Science 86, Berlin – Heidelberg – New York, 1980, pp. 1–42.