On Formalism in Specifications

Bertrand Meyer, University of California, Santa Barbara

A critique of a natural-language specification, followed by presentation of a mathematical alternative, demonstrates the weakness of natural language and the strength of formalism in requirements specifications.

Specification is the software life-cycle phase concerned with precise definition of the tasks to be performed by the system. Although software engineering textbooks emphasize its necessity, the specification phase is often overlooked in practice. Or, more precisely, it is confused with either the preceding phase, definition of system objectives, or the following phase, design. In the first case, considered here in particular, a natural-language *requirements document* is deemed sufficient to proceed to system design—without further specification activity.

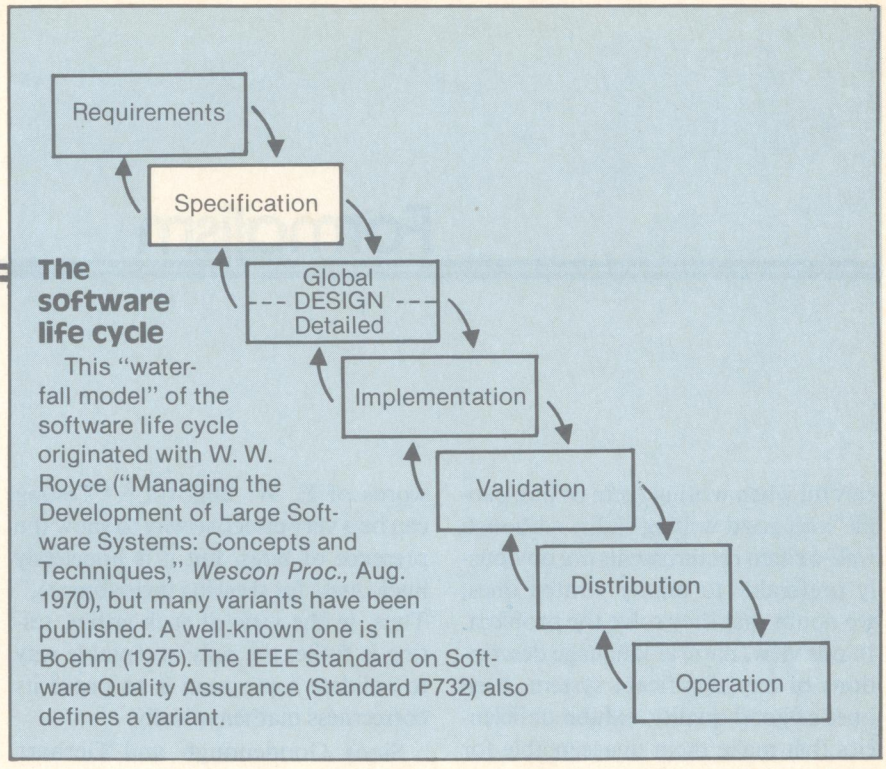
This article emphasizes the drawbacks of such an informal approach and shows the usefulness of formal specifications. To avoid possible misunderstanding, however, let's clarify one point at the outset: We in no way advocate formal specifications as a *replacement* for natural-language requirements; rather, we view them as a *complement* to natural-language descriptions and, as will be illustrated by an example, as an aid in *improving* the quality of natural-language specifications.

Readers already convinced of the benefits of formal specifications might find in this article some useful arguments to reinforce their viewpoint. Readers not sharing this view will, we hope, find some interesting ideas to ponder.

The seven sins of the specifier

The study of requirements documents, as they are routinely produced in industry, yields recurring patterns of

0740-7459/85/0001/0006$01.00 © 1985 IEEE



The software life cycle

This "waterfall model" of the software life cycle originated with W. W. Royce ("Managing the Development of Large Software Systems: Concepts and Techniques," *Wescon Proc.*, Aug. 1970), but many variants have been published. A well-known one is in Boehm (1975). The IEEE Standard on Software Quality Assurance (Standard P732) also defines a variant.

deficiencies. Table 1 lists seven classes of deficiencies that we have found to be both common and particularly damaging to the quality of requirements.

The classification is interesting for two reasons. First, by showing the pitfalls of natural-language requirements documents, it gives some weight to the thesis that formal specifications are needed as an intermediate step between requirements and design. Second, since natural-language requirements are necessary whether or not one accepts the thesis that they should be complemented with formal specifications, it provides writers of such requirements with a checklist of common mistakes. Writers of most kinds of software documentation (user manuals, programming language manuals, etc.) should find this list useful; we'll demonstrate its use through an example that exhibits all the defects except the last one.

A requirements document

The reader is invited to study, in light of the previous list, some of the software documentation available to him. We could do the same here and discuss actual requirements documents, taken from industrial software projects, as we did in a previous version of this article.[1] But such a discussion is not entirely satisfactory; the reader may feel that the examples chosen are not representative. Also, one sometimes hears the remark that nothing is inherently wrong with natural-language specifications. All one has to do, the argument continues, is to be

Table 1.
The seven sins of the specifier.

Noise:	The presence in the text of an element that does not carry information relevant to any feature of the problem. Variants: *redundancy; remorse.*
Silence:	The existence of a feature of the problem that is not covered by any element of the text.
Overspecification:	The presence in the text of an element that corresponds not to a feature of the problem but to features of a possible solution.
Contradiction:	The presence in the text of two or more elements that define a feature of the system in an incompatible way.
Ambiguity:	The presence in the text of an element that makes it possible to interpret a feature of the problem in at least two different ways.
Forward reference:	The presence in the text of an element that uses features of the problem not defined until later in the text.
Wishful thinking:	The presence in the text of an element that defines a feature of the problem in such a way that a candidate solution cannot realistically be validated with respect to this feature.

careful when writing them or hire people with good writing skills. Although well-written requirements are obviously preferable to poorly written ones, we doubt that they solve the problem. In our view, natural-language descriptions of any significant system, even ones of good quality, exhibit deficiencies that make them unacceptable for rigorous software development.

To support this view, we have chosen a single example, which, although openly academic in nature, is especially suitable because it was explicitly and carefully designed to be a "good" natural-language specification. This example is the specification of a well-known text-processing problem. The problem first appeared in a 1969 paper by Peter Naur where it was described as reproduced here in Figure 1.

Naur's paper was on a method for program construction and program proving; thus, the problem statement in Figure 1 was accompanied by a program and by a proof that the program indeed satisfied the requirements.

The problem appeared again in a paper by Goodenough and Gerhart, which had two successive versions. Both versions included a criticism of Naur's original specification.

Goodenough and Gerhart's work was on program testing. To explain why a paper on program testing included a criticism of Naur's text, it is necessary to review the methodological dispute surrounding the very concept of testing. Some researchers dismiss testing as a method for validating software because a test can cover only a fraction of significant cases. In the words of E. W. Dijkstra,[2] "Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." Thus, in the view of such critics, testing is futile; the only acceptable way to validate a program is to prove its correctness mathematically.

Since Goodenough and Gerhart were discussing test data selection methods, they felt compelled to refute this a priori objection to any research on testing. They dealt with it by showing significant errors in programs whose "proofs" had been published. Among the examples was Naur's program, in which they found seven errors—some minor, some serious.

Goodenough and Gerhart found seven errors—some minor, some serious—in Naur's program.

Our purpose here is not to enter the testing-versus-proving controversy. The Naur-Goodenough/Gerhart problem is interesting, however, because it exhibits in a particularly clear fashion some of the difficulties associated with natural-language specifications. Goodenough and Gerhart mention that the trouble with Naur's paper was partly due to inadequate specification; since their paper proposed a replacement for Naur's program, they gave a corrected specification. This specification was prepared with particular care and was changed as the paper was rewritten. Apparently somebody criticized the initial version, since the last version contains the following footnote:

> Making these specifications precise is difficult and is an excellent example of the specification task. The specifications here should be compared with those in our original paper.

Thus, when we examine the final specification, it is only fair to consider it not as an imperfect document written under the schedule constraints usually imposed on software projects in industry, but as the second version of a carefully thought-out text, describing what is really a toy problem, unplagued by any of the numerous special considerations that often obscure real-life problems. If a natural-language specification of a programming problem has ever been written with care, this is it. Yet, as we shall see, it is not without its own shadows.

Figure 2 (see p. 11) gives Goodenough and Gerhart's final specification, which should be read carefully at this point. For the remainder of this article, numbers in parentheses—for example, (21)—refer to lines of text as numbered in Figure 2.

Analysis of the specification

The first thing one notices in looking at Goodenough and Gerhart's specification is its length: about four times that of Naur's original by a simple character count. Clearly, the authors went to great pains to leave nothing out and to eliminate all ambiguity. As we shall see, this overzealous effort actually introduced problems. In any case, such length seems inappropriate

Rococo interior with fashionable pair dancing; engraving by Gravelot, 1770.
The Bettmann Archive

for specifying a problem that, after all, looks fairly simple to the unprejudiced observer.

Before embarking on a more detailed analysis of this text, we should emphasize that the aim of the game is not to criticize this particular paper; the official subject matter of Goodenough and Gerhart's work was testing, not specification, and the prescription period has expired anyway. We take the paper as an example because it provides a particularly compact basis for the study of common mistakes.

Noise. "Noise" elements are identified by solid underlines in Figure 2. Noise is not necessarily a bad thing in itself; in fact, it can play the same role as comments in programs. Often, however, noise elements actually obscure the text. When first encountering such an element, the reader thinks it brings new information, but upon closer examination, he realizes that the element only repeats known information in new terms. The reader must thus ask himself nonessential questions, which divert attention from the truly difficult aspects of the problem.

Here, a fraction of a second is needed to realize that a "nonempty sequence" of characters (8) is the same thing as "one or more" characters (9). These two expressions appear within a line of each other; the authors' aim was, presumably, to avoid a repetition. One is indeed taught in elementary writing courses that repetitions should be avoided, and no doubt this is a good rule as far as literary writing is con-

Given a text consisting of words separated by BLANKS or by NL (new line) characters, convert it to a line-by-line form in accordance with the following rules:

(1) line breaks must be made only where the given text has BLANK or NL;
(2) each line is filled as far as possible, as long as
(3) no line will contain more than MAXPOS characters.

Figure 1. Naur's original statement of a well-known text-processing problem.

References on the Naur-Goodenough/Gerhart problem

Original reference, Naur:

Peter Naur, "Programming by Action Clusters," *BIT*, Vol. 9, No. 3, 1969, pp. 250-258.

First version, Goodenough and Gerhart:

John B. Goodenough and Susan Gerhart, "Towards a Theory of Test Data Selection," *Proc. Third Int'l Conf. Reliable Software,* Los Angeles, 1975, pp. 493-510. Also published in *IEEE Trans. Software Engineering,* Vol. SE-1, No. 2, June 1975, pp. 156-173.

Revised version, Goodenough and Gerhart:

John B. Goodenough and Susan Gerhart, "Towards a Theory of Test: Data Selection Criteria," in *Current Trends in Programming Methodology, Vol. 2,* Raymond T. Yeh, ed., Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 44-79.

Another paper that uses the same problem as an example:

Glenford J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," *Comm. ACM,* Vol. 21, No. 9, Sept. 1978, pp. 760-768.

cerned. In a technical document, however, the rule to observe is exactly the opposite—namely, the same concept should always be denoted by the same words, lest the reader be confused.

An interesting variant of noise is **remorse**, a restriction to the description of a certain specification element made not where the element is *defined* but where it is *used*, as if the specifier suddenly regretted his initial definition. An example here is "the output text, if any" (20). Up to this point, the specification freely used the notion of output text (12,17); nowhere was there any hint that such a text might not exist. If the reader wondered about this problem, the specification did not provide an answer. Now, suddenly, when the discussion is focusing on something else, the reader is "reminded" that there might be no such thing as an output text, but no precise criterion is given as to when there is and when there isn't.

Another instance of remorse is the late definition of the "line" concept (24), to which we will return. We will meet again the tendency to say too much, which generates noise, as a source of contradiction and ambiguity.

Silence. In spite of all his efforts, the specifier often leaves, along with overdocumented elements, undefined features. Commonly, these features are fairly obvious to a community of application specialists, who are close to the initial customers, but they will be more obscure to those outside this circle. An example is the concept of "line," which is not really defined except in a parenthetical bit of remorse toward the end of the text (24), where it is described as a sequence of characters "between successive NL characters." (By the way, are those characters part of the line?)

An interesting point here is the cultural background necessary to understand this concept. In ASCII-oriented environments, "New Line" is a character; thus, people working on ASCII environments (DEC machines, for example) will probably understand easily the specification's basic hypothesis —namely, that NL is treated as an ordinary character upon input but triggers a carriage return upon output. These concepts are foreign, however, to somebody working in an EBCDIC environment, especially on IBM OS systems, on which files are made up of a sequence of "records" (corresponding, for example, to lines), each made up of a sequence of characters. A person coming from such an environment would not have written the above specification and will probably have trouble understanding it.

Besides, the late definition of line is plainly wrong. It applies only to lines that are neither at the very beginning nor at the very end of the text. In both these cases, a line is not "between successive NL characters" but between the beginning of the file and an NL, or between an NL and the end of the file—that is, between an NL and an ET. If we accept the authors' definition, the first and last lines of the output may be of arbitrary length; in fact, an output containing *no NL at all* is acceptable regardless of its length, since it does not have lines according to the definition given! This is obviously absurd and not what the authors had in mind, but the use of natural language leads naturally to such slips of the pen.

Another interesting silence concerns the variable Alarm. Line 16 specifies that this variable should be set to TRUE in case of an error, but nothing is said about what happens to it in other cases. The answer is obvious, of course; but the matter can only be brushed aside as minor by programmers who have never run into a bug due to an uninitialized variable. . .

It must be pointed out that Goodenough and Gerhart corrected a notable silence in Naur's original description. Naur's text does not explain what should be done with consecutive groups of more than one break character; this is one of the seven errors analyzed in Goodenough and Gerhart's paper. Their specification corrects it by requiring that such groups be reduced to a single break character in the output. Although something had to be done about the problem, note that this solution is, to some extent, obtained at the expense of simplicity. Eliminating redundant break characters and dividing a text into lines are two unrelated problems; merging them into a single specification complicates the whole affair.

It is probably better to deal with these two requirements separately, and this is what we do in the formal specification given below. Some of the current trends in programming methodology emphasize this approach—most notably under the influence of the Unix programming environment,

Ball at the home of a German baron; engraving circa 1750.
The Bettmann Archive

which, at least in principle, favors tools that are simple and composable rather than large and multipurpose.

Contradictions. There is another problem with the concept of line. Given a type *t*, one should distinguish between the types **seq**[*t*], whose elements are finite sequences of objects of type *t*, and **seq** [**seq** [*t*]], whose elements are sequences of sequences of objects of type *t*. Such a confusion can be found in Figure 2, where we are first told (1) that the input is a "stream," or sequence, of characters and later (10) that it "can be viewed" as a sequence of words and breaks. As any Lisp programmer knows, the sequences

$$<a\ b\ a\ c\ c\ a>$$

[sequence of objects]

and

$$<<a>\ <b\ a>\ <c\ c\ a>>$$

[sequence of sequences of objects]

are not the same. Note that the same problem with respect to the output is redeemed only by ambiguity; the type of the output is not clear:

- Is it **seq** [*CHAR*] as (21-22) seems to imply?
- Is it **seq** [*WORD*]—that is, **seq** [**seq** [*CHAR*]]—as (12-13) indicates?
- Or is it even **seq** [*LINE*] —that is, **seq**[**seq**[**seq**[*CHAR*]]]—if we consider a line as a sequence of words and breaks?

Thus, a sentence that at first appears to be only noise (9-11) yields a contradiction within a few lines (13-14): "The program's output should be the same sequence of words as in the in-

1 The program's input is a stream of characters whose end is
2 signaled with a special end-of-text character, ET. There is exactly
3 one ET character in each input stream. Characters are classified
4 as
5 • break characters—BL (blank) and NL (new line);
6 • nonbreak characters—all others except ET;
7 • the end-of-text indicator—ET.
8 A *word* is a nonempty sequence of nonbreak characters. A
9 *break* is a sequence of one or more break characters. Thus, the
10 input can be viewed as a sequence of words separated by breaks,
11 with possibly leading and trailing breaks, and ending with ET.
12 The program's output should be the same sequence of words
13 as in the input, with the exception that an oversize word (i.e., a
14 word containing more than MAXPOS characters, where MAXPOS
15 is a positive integer) should cause an error exit from the program
16 (i.e., a variable, Alarm, should have the value TRUE). Up to the
17 point of an error, the program's output should have the following
18 properties:
19 1. A new line should start only between words and at the be-
20 ginning of the output text, if any.
21 2. A break in the input is reduced to a single break character in
22 in the output.
23 3. As many words as possible should be placed on each line
24 (i.e., between successive NL characters).
25 4. No line may contain more than MAXPOS characters (words
26 and BLs).

Figure 2. Goodenough and Gerhart's final specification of the original problem statement in Figure 1. Analysis of this text, overprinted in blue, is according to the following key:

Noise **Ambiguity**
Remorse **Overspecification**
Contradiction **Forward reference**

put." This last comment is remarkable since *neither the input nor the output* is a sequence of words. Worse yet, even if we parse the input into a sequence of words, this sequence is not sufficient to determine the output—one also needs two binary informations: whether there is a leading and/or a trailing break.

The same sentence (9-11), in its overzealous effort to leave no stone unturned, ends up introducing another contradiction. An unbiased reader would be puzzled. How can the input "end with [the character] ET" (11) and at the same time have a "trailing break" (11)? "Trailing," precisely, means "at the end"! What's the last character if there is a "trailing" break: ET or a break character?

A more experienced reader, such as a programmer, will have no difficulty resolving this contradiction; his experience will tell him that "end" markers follow "trailing" characters. But this reliance on intuition and knowledge of the application domain can be particularly damaging when transposed to large requirements documents, which will be handed down to a group of system designers and implementors of diverse backgrounds and abilities.

Overspecification. Overspecification in requirements can be annoyingly close to silence. The reader is told too much about the *solution* while he is desperately trying to grasp the *problem* and figure out—by himself—features not covered by the text. Overspecification is typically, although certainly not exclusively, found in requirements documents written by programmers. Psychologically, this is understandable. An implementation-level concept is good, concrete, technical stuff, whereas true requirements deal with much less tangible material. To a computer specialist, a stack is easier to visualize than, say, the flow of information in a company or the needs of a radar operator. Thus, many specifiers have a natural tendency to cling to programming concepts. There is a price to pay for this: Implementation decisions taken too early may turn out to be wrong, and important problem features can be overlooked.

The example text contains an overspecification right from the first sentence: the notion of the end-of-text character ET. The only reason for the presence of this notion is Goodenough and Gerhart's desire to correct Naur's original program. Input-output facilities of the version of Algol 60 used by Naur (and, for fairness, by Goodenough and Gerhart) do not provide for end-of-file detection when reading, so one must assume the presence of a special character at the end of the file to make up for this deficiency. But ET is an implementation detail and should not be included in an abstract specification. Conceptually, the input is a finite sequence of characters; it should be transformed into an output that is a sequence of lines or, depending on the interpretation chosen, a sequence of characters. It is a programmer's vice to insist that finite sequences be specially marked at the end.

Why does the ET character receive such emphasis in Goodenough and Gerhart's specification? The reason is one of the errors in Naur's original program, which would go into an infinite loop unless the input was incorrect (that is, contained an oversize word). Upon closer examination, however, a case can be made for Naur's solution (without the other errors, of course). It is not so unrealistic to consider the required program as a potentially infinite process, which takes characters as input and produces lines as output, working somewhat like a device handler (for instance one that drives a printer) in an operating system. Such an interpretation should, of course, be clearly described in the specification, which was not the case with Naur's text. That decision would be less arbitrary than the one taken by Goodenough and Gerhart: their inclusion of ET changes the data structure at the specification level to accommodate the programming language used at the implementation stage.

The unacceptability of the change is further evidenced by the fact that the output does not satisfy the requirement on the input. Is it realistic to expect an existing file to be terminated by an explicit marker? If it is, the output produced by the program should satisfy that condition; however, examination of the specification, which is not completely clear on this matter, and, as a final criterion, of the proposed program, shows that ET will *not* be passed on to the output file. Assume that we want to write another program, for, say, right-justifying the text, that will take Goodenough and Gerhart's output (in "pipe" mode à la

Dancing the minuet in the open air; copper engraving by Charles Eisen. The Bettmann Archive

Unix). In designing that program, we will not be able to make the same assumption on its input. Thus, the overspecification has opened the way to serious inconsistencies.

Another overspecification in the text is the concept of "error exit" (16), which causes a "variable," Alarm, to have the value TRUE. Clearly, the notion of a variable belongs to the world of programs, not specifications. This piece of overspecification would have been less shocking if the problem had been defined as the task of writing a *procedure,* with Alarm as one of its parameters, or as one of the "exceptions" (in the sense of Clu or Ada) it might raise. A variable is internal to the program unit to which it belongs, whereas the specification of a parameter or an exception can be given relative to the environment of that unit.

The problem of the Alarm variable is less innocuous than it seems. One reason for shock at meeting the reference to this variable in a sequential reading of the text is that the definition of the error case (the one in which there is an oversize word) looks like overspecification until one sees the *last* sentence (25-26), 10 lines down, which gives the basic line-size constraint, MAXPOS. The world is really standing upside down here. Clearly, the constraint on word size is a consequence of the constraint on line size, and the definition of the error case cannot be understood until the latter constraint has been introduced.

We see here one of the major deficiencies plaguing requirements documents of more significant size: early inclusion of detailed descriptions of error handling, interwoven with descriptions of normal cases, which are usually much simpler. Here the matter is even worse; error processing is described before the reader has had a chance to recognize the problem—that is, before gaining an understanding of normal processing. Failure to clearly separate normal cases from erroneous ones makes the document much harder to understand.

	1	2	3	4	5	6	7	8	9	10
1	U	N	I	X		I	S		A	
2	T	R	A	D	E	M	A	R	K	
3	O	F		B	E	L	L			
4	L	A	B	O	R	A	T	O	R	I E S
	1	2	3	4	5	6	7	8	9	10

Figure 3. Output requirement (MAXPOS = 10).

Mathematically, a program that performs an input-to-output transformation often corresponds to the implementation of a partial function, which is not defined for some arguments of the input domain. Error processing then consists in "completing" the function with alternate results, such as error messages, for those arguments. This completion should not be confused with the definition of the function in its normal cases. Here, as we'll see later in a formal specification, failure to accommodate words larger than MAXPOS is a consequence of the requirements for normal processing, which can be *proved,* as a theorem, from the definition of the function.

Ambiguities. Error processing raises an ambiguity in the example text (Figure 3). The requirement that the output text satisfy properties 1 to 4 "up to

the point of an error" is susceptible to at least two interpretations.

The text says that up to (and presumably including) the point of the error, the program's output should correspond to the input. But where is the "point of the error" in Figure 3? Is it [line 4, column 10], last acceptable letter, or [3, 7], end of the last acceptable word? Nothing in the text allows the reader to decide between these two interpretations.

Another interesting ambiguity is connected with the basic constraint on acceptable solutions (23): "As many words as possible should be placed on each line." If we have, say, MAXPOS = 10 and the input text

WHO WHAT WHEN

there are two equally correct two-line solutions (WHAT may be on either the first or second line). This ambiguity may be acceptable since neither solution appears superior to the other; the specification as such is nondeterministic. We suspect (perhaps wrongly) that this nondeterminism was not intentional and that there was an implicit overspecification in the authors' minds: they considered it obvious that the input would be processed sequentially, so any ambiguity, as in the example above, would be solved by placing as many words as possible on the earlier line (giving line WHO WHAT followed by line WHEN). In this interpretation, property 3 (23-24) actually means, "As many words as possible should be placed on each line as *it is encountered in the sequential construction of the output.*" If this is the case, the specification should state it precisely.

Another potential source of ambiguity is the use of imprecise or poorly defined terms—for example, the use of "stream" (1) rather than the more standard "sequence." The expression "error exit" (15), stemming from the overspecification seen above, is ambiguous, and the reader is not comforted by the explanation that follows it ("i.e., a variable, Alarm, should have the value TRUE"); the notion of assigning a value to a variable does not by itself imply the idea of an "exit," which also means that the program stops in some fashion. We have seen that the concept of "line" is not well defined (24). Also note that the expression "new line" is to be parsed as a single entity (the *new line* character) in its first appearance (5) and as separate words ("a new *line* should start. . .") in its second (19).

Forward references. In a requirements document, not all forward references are bad. Some, corresponding to a top-down presentation of the concepts ("the notion of . . . will be studied in detail in section . . ."), might even be considered good practice, provided there are not too many. But *implicit* forward references (that is, uses of a concept that come before the proper definition of the concept, without particular warning to the reader) can present much more of a problem. They make a document extremely hard to read, especially in the absence of the technical apparatus (index, glossary, etc.) that should be a part of all requirements specifications and other software documents.

Here, of course, the text is very short, so the annoyance caused by forward references is nowhere near what it can be with full-size documents. Note, however, that ET is used three times (2, 3, 6) before it is defined (7), that the notion of line, defined not quite satisfactorily (24), has been used earlier (19-20), and that MAXPOS is used just before its definition (14).

So what? In dissecting Goodenough and Gerhart's specification, we identified a significant number of problems in a text that may seem innocuous to a superficial observer. Not all the problems were equally serious, and the reader may have felt that we were a bit pedantic at times. We submit, however, that one must be pedantic in dealing with such matters. Inconsistencies, ambiguities, and the like may not warrant the gallows when the problem is to split up a sequence of characters into lines. But keep in mind how the above defects transpose to more serious matters—a nuclear reactor control system, a missile guidance system, or even just a payroll program. The computer that executes the code resulting from a faulty specification is more pedantic than any human referee could ever be.

Thus, we should consider Goodenough and Gerhart's specification not only as an object of study in itself but also, and more importantly, as a microcosm for conveniently observing deficiencies typical of more meaningful requirements documents. Al-

Two people doing the minuet; copper engraving by Nilsson.
The Bettmann Archive

though the text was written with great care, we have witnessed how the authors, who started out to improve upon Naur's terse but simple text, sentence after sentence became a little more entangled in their own rosary of caveats. This says a lot about why interminable manuals occupy so much shelf space in programmers' offices and computer rooms.

In our opinion, the situation can be significantly improved by a reasoned use of more formal specifications. But again, let's emphasize that such specifications are a complement to natural language documents, not a replacement. In fact, we'll show how a detour through formal specification may eventually lead to a better English description. This and other benefits of formal approaches more than compensate for the effort needed to write and understand methematical notations.

We will now introduce such notations, which will allow us to give a formal specification of the Naur-Goodenough/Gerhart problem.

Elements for a formal specification

Many formal specification languages have been designed in recent years (see box). Choosing one of these languages would force the reader to learn its particular notation and would obscure the essential fact—namely, that their underlying concepts are, for the most part, well-known mathematical notions like sets, functions, relations, and sequences. We thus prefer to use a more-or-less standard mathematical notation. The style of exposition will be similar to that found in mathematical texts; translation to a specific formal specification language should not be hard, provided the language supports the relevant concepts.

Overview. Perhaps the only difficult part of the Naur-Goodenough/Gerhart problem is that the processing to be performed on the text involves three aspects: reducing breaks to a single break character, making sure no line has more than MAXPOS characters, and filling lines as much as possible. If these three requirements are separated, things become much simpler. Consequently, we will define the problem formally by considering two simple binary relations, called *short_breaks* and *limited_length,* and a function called *FEWEST_LINES.* (Throughout the discussion of the formal specification, the reader may wish to refer to Figure 4 for a picture of the overall structure of the relations and functions involved.)

Relation *short_breaks* holds between two sequences of characters *a* and *b* if and only if *b* is identical to *a,* except that breaks in *a* (i.e., successive break characters) have been reduced to single break characters in *b*.

Relation *limited_length* holds between two sequences of characters *b* and *c* if and only if *c* is a "limited length version" of *b*: that is, no line in *c* has length greater than MAXPOS, and *c* is identical to *b* except that some blanks may have been replaced with

References on formal specification

Many formal specification languages have been designed in recent years. A few are listed here, without any claim to exhaustivity.

Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs,* R. McNaughten and R.C. McKeag, eds., Cambridge University Press, 1980.

Rod M. Burstall and Joe A. Goguen, "Putting Theories Together to Make Specifications," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence,* Cambridge, Mass, 1977, pp. 1045-1058.

Cliff B., Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, N.J., 1980

R. Locasso, John Scheid, Val Schorre, and Paul R. Eggert, "The Ina Jo Specification Language Reference Manual," Technical Report TM-(L)-/6021/001/00, System Development Corporation, Santa Monica, Calif., June 1980.

David R. Musser, "Abstract Data Type Specification in the AFFIRM System," *IEEE Trans. Software Engineering,* Vol. SE-6, No. 1, Jan. 1980, pp. 24-32.

L. Robinson and Olivier Roubine, *Special Reference Manual,* Stanford Research Institute, 1980.

new lines and/or some new lines with blanks.

By applying these two relations successively, we associate with any sequence of characters *a* all sequences of characters that are "made of the same words," separated only by single breaks, and fit on lines no longer than MAXPOS. Given such a set of sequences, say, *SSC,* then *FEWEST_LINES* (*SSC*) is the subset of *SSC* containing those sequences that consist of a minimum number of lines and thus are acceptable outputs for the program.

We'll now define these notions formally, but a few simple conventions are needed first.

Basic form of the specification. As a general convention, we use uppercase for sets and for functions whose results are sets and lowercase for other functions, elements of sets (except for MAXPOS, which we write in uppercase as in the original specification), sequences, and relations.

The program to be written is the implementation of a function

$$sol: INPUT \rightarrow OUTPUT$$

where *INPUT* and *OUTPUT* are the sets of possible inputs and outputs, which we will describe below as sets of sequences. Function *sol* must satisfy certain constraints, which it is the role of the specification to express.

As noted above, there may be more than one correct output for a given input; in other words, a truly general specification of the problem should be nondeterministic. We will represent this fact by defining a binary relation between sets *INPUT* and *OUTPUT.* We call *goal* this binary relation; then a function *sol* will be a correct solution if and only if the following two conditions are satisfied (readers who are not so sure about functions and relations are referred to the refresher in the adjacent box):

- function *sol* is defined wherever relation *goal* is defined—that is, *sol* (*i*) exists for any *i* in the domain of *goal*;
- for any *i* for which *goal* is defined, then *sol* (*i*) yields a "solution" to *goal*—that is, *goal* (*i, sol* (*i*)) holds.

This definition is expressed in mathematical notation by writing that *sol* is an acceptable function if and only if

$$\forall i \in \textbf{dom}\ (goal),\quad i \in \textbf{dom}\,(sol) \text{ and } goal\ (i, sol\ (i))$$

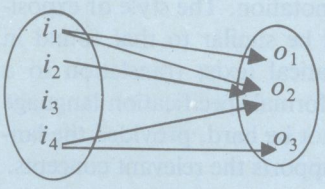
where **dom** (*sol*) is the domain of function *sol.* Note that there may be some inputs for which there is no acceptable solution (those not in the domain of *goal*), so *sol* may be a partial function. Also, in more concise notation, the above property can simply be

A reminder on functions and relations

Consider two sets—for example, *INPUT* and *OUTPUT.* A **binary relation** between these two sets is a set of pairs

$$\{<i_1, o_1>, <i_2, o_2>, \ldots\}$$

where each i_k belongs to set *INPUT* and each o_k belongs to set *OUTPUT.* Such a relation is represented pictorially at right. If *goal* is a relation, then we write *goal* (*i, o*) to express that the pair $<i, o>$ belongs to the relation.



A relation.

The **domain** of such a relation, written **dom** (*goal*), is the subset of *INPUT* containing only those elements *i* such that *goal* (*i, o*) holds for **at least** one element *o* in *OUTPUT.* Thus, in the example pictured, i_1, i_2, and i_4, but not i_3, belong to the domain of the relation.

A **function** is a relation *f* such that for any *i* there is **at most** one *o* for which *f* (*i, o*) holds; if *o* exists, then one may write $o = f(i)$. The relation pictured above is not a function, since i_1, for instance, has two buddies o_1 and o_2. Note that the domain of a function is made of those elements of *INPUT* for which there is **exactly** one corresponding element in *OUTPUT.*

Le Bal Paré: Typical Louis XVI court scene of the 18th century.
The Bettmann Archive

expressed by writing that the domain of *sol* is at least as large as the domain of *goal,* and that *sol* is **included** in *goal* (both being defined as sets of pairs):

dom (*goal*) ⊂ **dom** (*sol*)
and *sol* ⊂ *goal*

This way of presenting a specification is of very general applicability for programs performing input-to-output transformations. Such a program may be viewed as the implementation of a certain function (*sol*) which must ensure that a certain relation (*goal*) is satisfied between its argument and its result; in mathematical terms, the function is included in (is a subset of) the relation. To specify the problem is to define the relation; to construct the program is to find an implementable function *sol* satisfying the above conditions.[3]

Characters and sequences. The principal set of interest in our problem is the set of characters, which we denote by *CHAR*. The only property of *CHAR* that matters here is that *CHAR* contains two elements of particular interest, *blank* and *new_line*. We call *BREAK_CHAR* the subset of *CHAR* consisting of these two elements:

BREAK_CHAR ≡ {*blank, new_line*}

The basic concept in this problem is that of sequence. If *X* is a set, we denote by **seq** [*X*] the set whose elements are finite sequences of elements of *X*. Such a sequence is written, for example, as

<*a, b, a, c, c, d*>

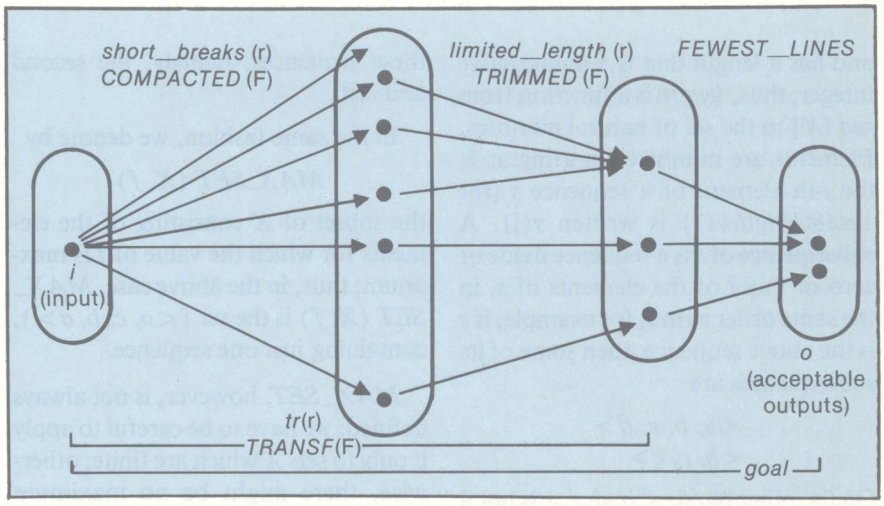


Figure 4. Overall structure of the specification: (r) indicates a relation, (F) a function.

Basic set and logic notations

The definitions marked (*) introduce predicates, that is, expressions which may have value "true" or "false."

{*a, b, c, ...*}: the set made up of elements *a, b, c, . . .*

$x \in A$: *x* is an element of *A* (*).

$x \notin A$: *x* is not an element of *A* (*).

$A \subset B$: *A* is a subset of *B* (all elements of *A* are elements of *B*) (*).

$\{x \in A \mid P(x)\}$: The (possibly empty) subset of *A* made up of those elements *x* which satisfy property *P*.

$\forall x \in A, P(x)$: All elements *x* of *A*, if any, satisfy property *P* (or: no element of *A* violates *P*); holds in particular whenever *A* is empty (*).

$\exists x \in A, P(x)$: There is at least one element *x* in *A* which satisfies property *P*; may only hold if *A* is nonempty (*).

$a \Rightarrow b$: *a* implies *b*.

a..b: the integer interval containing all the integers *i* such that $a \leq i \leq b$; empty if $a > b$. This notation is borrowed from Pascal.

The symbol ≡ means "is defined as."

and has a length that is a nonnegative integer; thus, *length* is a function from **seq** [*X*] to the set of natural numbers. Elements are numbered starting at 1; the *i*-th element of a sequence *s* (for $1 \le i \le length(s)$) is written $s(i)$. A **subsequence** of *s* is a sequence made of zero or more of the elements of *s*, in the same order as in *s*; for example, if *s* is the above sequence, then some of its subsequences are

$$<a, b, c, d>$$
$$<b, c, c>$$

On the other hand, $<b, d, c>$ is not a subsequence of *s* because the original order of its elements in *s* is not preserved.

The set of subsequences of *s* will be written *SUBSEQUENCES* (*s*).

The concept of sequences is well known, and we rely on the reader's understanding here. A formal definition of sequences and of the above notions is given in the box on the adjacent page.

Minima and maxima. If *X* is a set, and *f* is a function from *X* to the set of natural numbers,

$$MIN\_SET\ (X, f)$$

denotes the subset of *X* consisting of the elements for which the value of *f* is minimum. For example, if *X* is the following set, containing four sequences

$$X \equiv \{<a, c, b, a>, <a, b>, <b, a, b>, <c, c>\}$$

and *f* is the *length* function on sequences, then *MIN_SET* (*X, f*) will be the set consisting of the shortest of these sequences, namely, the second and last.

In the same fashion, we denote by

$$MAX\_SET\ (X, f)$$

the subset of *X* consisting of the elements for which the value of *f* is maximum; thus, in the above case, *MAX_SET* (*X, f*) is the set $\{<a, c, b, a>\}$, containing just one sequence.

MAX_SET, however, is not always defined; we have to be careful to apply it only to sets *X* which are finite; otherwise, there might be no maximum value for *f*. Note that the results of *MIN_SET* and *MAX_SET* are a subset of *X* rather than a single element, since there may be more than one element with minimum or maximum *f* value. These subsets are nonempty if and only if *X* is nonempty.

We will also need a way to denote the minimum and maximum elements of a set of natural numbers *SN*. They will be written, in the usual fashion, *min* (*SN*) and *max* (*SN*). Thus, if *SN* is the set

$$SN \equiv \{341, 7, 3, 654\}$$

then *min* (*SN*) is 3 and *max* (*SN*) is 654. Note that *min* and *max*, contrary to *MIN_SET* and *MAX_SET*, yield a natural number, not a set. Also in constrast to *MIN_SET* and *MAX_SET*, which are defined for empty sets (they yield an empty result), both *min* and *max* are defined only if the set *SN* is not empty; *max* further requires that *SN* be finite. It is essential to check for these conditions whenever using these functions.

Input and output sets. In the problem at hand, the input is a sequence of characters; we choose to describe the output as a sequence of characters as well. Thus, we define the two sets:

$$INPUT \equiv \textbf{seq}\ [CHAR]$$
$$OUTPUT \equiv \textbf{seq}\ [CHAR]$$

Note that, as mentioned above, another interpretation could have defined the set of possible outputs as **seq** [*LINE*], with *LINE* itself being defined as **seq** [*CHAR*] (or possibly **seq** [*WORD*] with *WORD* ≡ **seq** [*CHAR*], plus information on leading and trailing breaks).

We will now define the relations *short_breaks* and *limited_length* and the function *FEWEST_LINES*.

The formal specification

Short breaks. Let *a* be a sequence of characters. We define *SINGLE_BREAKS* (*a*) as the set of subsequences of *a* such that no two consecutive characters are break characters:

$$SINGLE\_BREAKS\ (a) \equiv \{s \in SUBSEQUENCE\ (a) \mid \forall i \in 2..length\ (s),\ s(i-1) \in BREAK\_CHAR \Rightarrow s(i) \notin BREAK\_CHAR\}$$

Note that we use the Pascal notation, *a. .b*, to denote the (possibly empty) set of integers *i* such that $a \le i \le b$.

Next, we define *COMPACTED* (*a*) as the subset of *SINGLE_BREAKS* (*a*) containing those sequences of maximum length:

$$COMPACTED\ (a) \equiv MAX\_SET\ (SINGLE\_BREAKS\ (a), length)$$

As stated above, *MAX_SET* (X, f) may be be undefined if X is an infinite set. This cannot occur here, however, since *SINGLE_BREAKS* (a) is a subset of *SUBSEQUENCES* (a) which, for any sequence of characters a, is finite.

Note that any sequence b in *COMPACTED* (a) must have retained from a all nonbreak characters (if such a character had been omitted, it could be inserted into b and yield a longer element of *SINGLE_BREAKS* (a)), and has a single break character where a had one or more consecutive break characters.

Thus, the relation *short_breaks* (a, b), which holds between a and b if and only if a and b are made of the same sequences of words and breaks but the breaks in b consist of a single break character, can be expressed simply by

short_breaks $(a, b) \equiv$
$b \in$ *COMPACTED* (a)

Limited length. The relation *limited_length* (b, c) holds between sequences b and c if and only if

- c is the same sequence as b, except that it may have a *new_line* wherever b has a *blank*, or conversely; and
- the maximum line length of c, defined as the maximum number of consecutive characters none of which is a *new_line*, is less than or equal to MAXPOS.

This is expressed more precisely as follows:

limited_length $(b, c) \equiv$
$c \in$ *TRIMMED* (b)

A definition of sequences

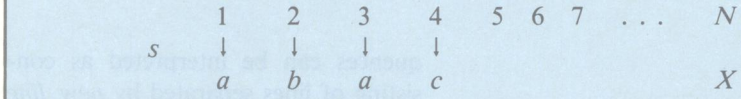
The following presentation is based on the formal specification of sequences given in the Z reference manual.[11]

N will denote the set of natural numbers.

Definition:

seq $[X]$, the set of finite sequences of elements of X, is defined as the set of partial functions from N to X whose domains are intervals of the form $1..n$ for some natural number n.

So a sequence is defined as a partial function; for example, the sequence $s = <a, b, a, c>$ is the function defined for arguments 1, 2, 3, and 4 only, and whose value is a for 1 and 3, b for 2, and c for 4. The following is a pictorial representation of s:



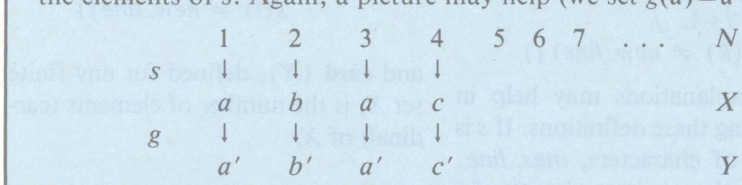
Note that the above definition allows $n=0$ (empty interval, thus empty function – that is, empty sequence) and that it justifies the notation $s(i)$ for the ith element of sequence s (which is the result of applying function s to element i).

The **length** of a sequence is defined as the largest integer for which the associated partial function is defined (i.e., n in the above definition).

Now let s be a sequence of elements of X and g be a (total) function from X to some set Y. The composition

$$g \bullet s$$

is a partial function from the set of natural numbers to Y, which has the same domain as s; thus, it is a sequence of elements of Y, with the same length as s. This sequence is obtained from s by applying g to all the elements of s. Again, a picture may help (we set $g(a)=a'$, etc.):



Now take for X the set N of natural numbers. A **sorted sequence** of natural numbers is an element s of **seq** $[N]$ such that

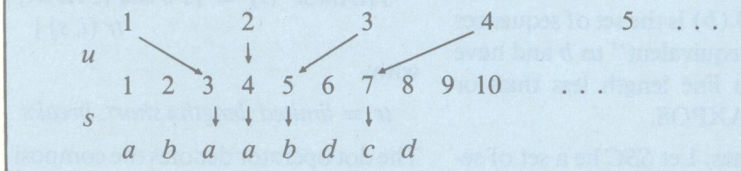
$$\forall i \in 2\,..\,length\ (s),\ s(i-1) \leq s(i)$$

With this definition, it becomes easy to formally define the notion of **subsequence** used in the text.

Definition:

Let s be an element of **seq** $[X]$ for some set X. A subsequence of s is a sequence of the form $s \bullet u$ where u is a sorted sequence of natural numbers.

The following picture shows how $<a\ a\ b\ c>$ is obtained as a subsequence of $<a\ b\ a\ a\ b\ d\ c\ d>$ using the above definition. The sorted sequence u of natural numbers used here is $<3\ 4\ 5\ 7>$; $<1\ 3\ 5\ 7>$ or $<1\ 4\ 5\ 7>$ would also work.



where

TRIMMED (*b*) ≡
  {*s* ∈ *EQUIVALENT* (*b*) |
 max_line_length (*s*) ≤ MAXPOS}

EQUIVALENT (*b*) ≡
  {*s* ∈ **seq**[*CHAR*] |
    length (*s*) = *length* (*b*) **and**
    (∀ *i* ∈ 1..*length* (*b*),
     s(*i*) ≠ *b*(*i*) ⇒
       s(*i*) ∈ *BREAK_CHAR* **and**
       b(*i*) ∈ *BREAK_CHAR*)}

max_line_length (*s*) ≡
  max ({*j*−*i* |
    0≤*i*≤*j*≤*length* (*s*) **and**
    (∀ *k* ∈ *i*+1. .*j*,
       s(*k*) ≠ *new_line*)})

A few explanations may help in understanding these definitions. If *s* is a sequence of characters, *max_line_length* (*s*) is the maximum length of a line in *s*, expressed as the maximum number of consecutive characters, none of which is a new line. In other words, it is the maximum value of *j*−*i* such that *s*(*k*) is not a new line for any *k* in the interval *i*+1..*j*. (We will have more to say about this definition below.) *EQUIVALENT* (*b*) is the set of sequences that are "equivalent" to sequence *b* in the sense of being identical to *b*, except that *new_line* characters may be substituted for *blank* characters or vice versa. Finally, *TRIMMED* (*b*) is the set of sequences which are "equivalent" to *b* and have a maximum line length less than or equal to MAXPOS.

Fewest lines. Let *SSC* be a set of sequences of characters. These sequences can be interpreted as consisting of lines separated by *new_line* characters. We define the set *FEWEST_LINES* (*SSC*) as the subset of *SSC* consisting of those sequences that have as few lines as possible:

FEWEST_LINES (*SSC*) ≡
  MIN_SET (*SSC*,
      number_of_new_lines)

where the function *number_of_new_lines* is defined by:

number_of_new_lines (*s*) ≡
  card ({*i* ∈ 1..*length* (*s*) |
      s(*i*) = *new_line*})

and **card** (*X*), defined for any finite set *X*, is the number of elements (cardinal) of *X*.

The basic relation. The above definitions allow us to define the basic relation of the problem, relation *goal*, precisely. Relation *goal* (*i*,*o*) holds between input *i* and output *o*, both of which are sequences of characters, if and only if

o ∈ *FEWEST_LINES* (*TRANSF* (*i*))

TRANSF (*i*) is the set of sequences related to *i* by the composition of the two relations *short_breaks* and *limited_length:*

TRANSF (*i*) ≡ {*s* ∈ **seq** [*CHAR*] |
    tr (*i*, *s*)}

with

tr ≡ *limited_length*•*short_breaks*

The dot operator denotes the composition of relations (see box). A look at Figure 4 may help explain the role of the various functions and relations in the above specification.

Existence of solutions. Once we have a formal specification, what can we do with it? Relying on the specification as a basis for the next stages of the software life cycle—program design and implementation (e.g., translating ∀s into loops) is the most obvious use. However, we'd like to emphasize two others. One use, studied in the next section, is as a starting point for better natural-language requirements. The other, to which we now turn, is querying the specification to learn as much as possible about properties of the problem and valid solutions.

What can the given specification teach us about the Naur-Goodenough /Gerhart problem and its solution? First, let's determine when solutions do exist. It is trivial to prove that, given a sequence of characters *a*, there is always at least one sequence *b* such that relation *short_breaks* (*a*, *b*) holds. Given *b*, however, the necessary and sufficient condition for the existence of at least one sequence *c* such that *limited_length* (*b*, *c*) holds is that *b* contains no word (i.e., contiguous subsequence of non-break characters) of length greater than MAXPOS. This follows from the definitions of *TRIMMED* and *max_line_length* used in the definition of *limited_length*. Thus, the domain of definition of the relation *tr*, which is also the domain of the function *TRANSF* and thus of the relation *goal*, is the set of input texts containing no word longer than MAX-

"The Compleat Figure of the Minuet," an engraving from George Bickhams's *An Easy Introduction to Dancing*, shows the basic spatial shapes used in the minuet; 1738.

From the library of Christena L. Schlundt, University of California, Riverside

POS. This can be formulated as a theorem:

dom (*goal*) =
  {*s* ∈ **seq** [*CHAR*] |
    ∀*i* ∈ 1..*length*(*s*) − MAXPOS,
    ∃*j* ∈ *i*. .*i* + MAXPOS,
      s(*j*) ∈ *BREAK_CHAR*}

The property expressed by this theorem is that the domain of relation *goal* consists of sequences such that, if a character *c* is followed by MAXPOS other characters, at least one character among *c* and the other characters must be a break.

An important problem, not addressed here, is how the specification deals with erroneous cases—that is, with inputs not in the domain of the *goal* relation—like sequences with oversize words. Clearly, a robust and complete specification should include (along with *goal*) another relation, say, *exceptional_goal*, whose domain is *INPUT* − **dom** (*goal*) (set difference); this relation would complement *goal* by defining alternative results (usually some kind of error message) for erroneous inputs. Formal specification of erroneous cases falls beyond the scope of this article, but a discussion of the problem and precise definitions of terms such as "error," "failure," and "exception" can be found in a paper by Cristian.[4]

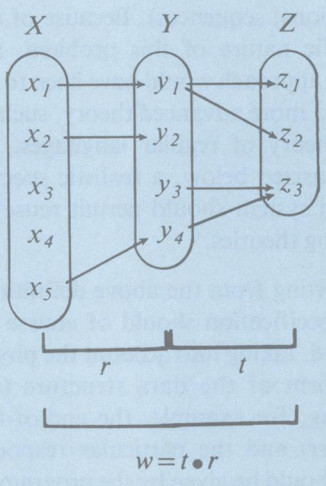
Discussion. What we have obtained is an abstract specification—this is, a mathematical description of the problem. It would be difficult to criticize this specification as being oriented toward a particular implementation: if

Composition of relations

Let *r* and *t* be two relations; *r* is from *X* to *Y* and *t* is from *Y* to *Z* (see figure).

The composition of these two relations, written $t \bullet r$ (note the order), is the relation *w* between sets *X* and *Z* such that *w* (*x*, *z*) holds if and only if there is (at least) one element *y* in *Y* such that both *r* (*x*, *y*) and *t* (*x*, *y*) hold.

Thus, in the example illustrated, *w* holds for the pairs $<x_1, z_1>$, $<x_1, z_2>$, and $<x_5, z_3>$ (and for these pairs only).



followed to the letter, the specification would lead to a program that (as illustrated in Figure 4) would first generate all possible distributions of the input over lines of length less than or equal to MAXPOS and then search the resulting list for solutions with minimum number of *new_line* characters—not a very efficient implementation!

An element that does seem to point toward a particular implementation technique is the composition of relations *short_breaks* and *limited_length,* which seems to imply a two-step process (first remove break characters, then cut into lines). A first design could indeed use a two-step solution. The steps could then be merged using coroutine-like concepts, such as the Unix notion of pipe or the "program inversion" idea of Jackson's program design method.[5]

We chose to model the problem's object and operations with very simple mathematical notions (sets, relations, functions, sequences). Because of the specific nature of this problem, another approach would have been to rely on a more advanced theory, such as the theory of regular languages. As emphasized below, a realistic specification system should permit reuse of existing theories.[6]

Starting from the above definition, the specification should of course be refined, taking into account the physical form of the data structure (including, for example, the end-of-file marker) and the particular response that should be given by the program in case of erroneous input.

Conclusion

Although natural language is the ideal notation for most aspects of human communication, from love letters to introductory programming language manuals, there are cases[7] where it is not appropriate. Software specifications, for example, require more rigorous formalism.

The use of formal notation does not, however, preclude that of natural language. In fact, mathematical specification of a problem usually leads to a better natural-language description. This is because formal notations naturally lead the specifier to raise some questions that might have remained unasked, and thus unanswered, in an informal approach.

Mathematical definition. Formal specifications help expose ambiguities and contradictions because they force the specifier to describe features of the problem precisely and rigorously. The problem studied in this article contains many examples of this. For example, let us try to redefine the function *max_line_length* using the definition of "line" taken from Goodenough and Gerhart's specification (line 24: "between successive NL characters"). Writing this definition mathematically, we obtain something like

$$\begin{aligned} &\mathit{max\_line\_length}\ (s) \equiv \\ &\quad \mathit{max}\ (\{\mathit{line\_length}\ (s, i) \mid \\ &\qquad 1 \le i \le \mathit{length}\ (s)\ \textbf{and} \\ &\qquad s(i) = \mathit{new\_line}\}) \end{aligned}$$

where *line_length* (s, i), the length of the line beginning after the *new_line* at position i in sequence s, may be defined as a minimum:

$$\begin{aligned} &\mathit{line\_length}\ (s, i) \equiv \\ &\quad \mathit{min}(\{k \mid \\ &\qquad 0 \le k < \mathit{length}\ (s-i)\ \textbf{and} \\ &\qquad s(i+k+1) = \mathit{new\_line}\}) \end{aligned}$$

However, as mentioned above, the maximum or minimum of a set of natural numbers is defined if and only if this set is nonempty and, in the maximum case, finite; so using mathematical notation prompts us to check for these conditions. Finiteness presents no problem, but we see immediately that the set whose maximum is sought in the definition of *max_line_length* will be empty if the sequence s does not contain any *new_line* character. Even if it contains one, *line_length* (s, i), itself a minimum, will not be defined if there is no other *new_line* further in the sequence. This prompts us to look for a better definition.

A fairly natural reaction at this point is to see that we really don't need to define the concept of "line," only that of **maximum line length.** Once we have noticed this, it's easy to come up with a correct definition: *the maximum number of consecutive characters, none of which is a new line.* This is the definition that was given above:

$$\begin{aligned} &\mathit{max\_line\_length}\ (s) \equiv \\ &\quad \mathit{max}\ (\{j-i \mid \\ &\qquad 0 \le i \le j \le \mathit{length}(s)\ \textbf{and} \\ &\qquad (\forall\, k \in i+1..\ j, \\ &\qquad\quad s(k) \neq \mathit{new\_line})\}) \end{aligned}$$

Note that we have been careful to apply *max* to a set that always contains at least one value (zero, obtained for

$i = j = 0$), even if s is an empty sequence (see box).

Natural language definition. Once such a mathematical definition has been produced, it may in return influence the natural language definition. In this example, the formal definition suggests that we should refrain from trying to define the concept of "a line in the text" which, although intuitively clear, is slightly tricky when one attempts to specify it precisely, as Goodenough and Gerhart's text shows. Instead, we should focus on the notion of "maximum line length," which is always defined, even for a text consisting of *new_line* characters only. Once we have obtained the specification of *max_line_length*, we can build on it and include it in the English problem definition a sentence such as

> The maximum number of consecutive characters, none of which is a *new_line*, should not exceed MAXPOS.

This sentence, a direct translation from the formal definition, is not, admittedly, of the most gracious sytle; but it is easy to remove the double negation, yielding

> Any consecutive MAXPOS + 1 characters should include a *new_line*.

The main advantage of natural language texts is their understandability. One should concentrate on this asset rather than trying to use natural language for precision and rigor, qualities for which it is hopelessly inadequate. Understandability is seri-

The reasoning behind formal specifications: the example of max_line_length

How does one obtain a formal expression such as the one defining *max_line_length*? Let's analyze the different steps involved.

We want to express the fact that *max_line_length* (s) is the maximum length of a line in s. A definition that avoids the pitfalls mentioned in the analysis of Goodenough and Gerhart's text is, informally, "the maximum number of consecutive characters, none of which is a new line."

To translate this definition into a formal description, we have to express the notion of a contiguous subsequence of s that does not contain a *new_line*. A contiguous subsequence can be given by its end indices, say, i and j. The sequence comprising the elements between indices i and j will have length $j-i+1$; if it is to yield a line length, then $s(k)$ should be a character other than *new_line* for any k between i and j, inclusive. Thus, a first try might yield

$$max\_line\_length\ (s) \equiv max\ (LINE\_LENGTHS)$$

where the set *LINE_LENGTHS* is defined as

$$LINE\_LENGTHS \equiv \{j-i+1 \mid 1 \le i \le j \le length\ (s)\ \textbf{and}\ (\forall k \in i..j,\ s(k) \neq new\_line)\}$$

But beware! One should only apply *max* to nonempty sets. With the above convention, we can end up with *LINE_LENGTHS* being empty if s is an empty sequence or all its characters are *new_line*; in either case, no i, j pair satisfies the condition. Now, if we write a program for the Naur-Goodenough/Gerhart problem and put in into a library, sooner or later someone will apply it to a sequence that is empty or entirely made of *new_line* characters, so we had better deal with these cases in a clean fashion.

The culprit is the condition $i \le j$, which prevents us from finding a satisfactory i and j in the borderline cases mentioned. The problem disappears, however, if we replace this condition by $i-1 \le j$. Then, for a sequence having only *new_line* characters or no character at all, the set *LINE_LENGTHS* will contain one element, 0, obtained for $i=1$ and $j=0$. For these values, the interval $i..j$ is empty; thus, the $\forall\ldots$ clause is true. (Remember that a property of the form $\forall x \in E,\ P\ (x)$ is always true when the set E is empty, regardless of what property P is.) Thus, we obtain the following replacement:

$$LINE\_LENGTHS \equiv \{j-i+1 \mid 0 \le i-1 \le j \le length\ (s)\ \textbf{and}\ (\forall k \in i..j,\ s(k) \neq new\_line)\}$$

(The first condition has been written $0 \le i-1$ instead of $1 \le i$.)

We have chosen to simplify slightly the writing of this condition by a change of variable (use i for $i-1$, thus eliminating $+1$ and -1 terms):

$$LINE\_LENGTHS \equiv \{j-i \mid 0 \le i \le j \le length\ (s)\ \textbf{and}\ (\forall k \in i+1..j,\ s(k) \neq new\_line)\}$$

This new version is defined in all cases.

It should be noted that this kind of analysis, which at first sight might seem quite remote from programmers' concerns, is in fact closely connected to typical patterns of reasoning about programs. Anyone who has tried to debug a loop that sometimes goes one iteration too few or too many, or works improperly for empty inputs or other borderline cases, will recognize the line followed in the above discussion. It is our contention, however, that such analysis is better performed at the specification level, dealing with simple and well-defined mathematical concepts, than at program debugging time, when the issues are obscured by many irrelevant details, implementation-dependent features, and idiosyncrasies of programming languages.

ously hindered when natural language requirements become ridiculously long in a vain attempt to chase away silence, ambiguity, contradiction, etc. Such attempts, as shown by the text studied here, only make matters worse. The length of many requirements documents found in actual industrial practice, often extending over hundreds or even thousands of pages, is due to such misuse of natural language. Natural language descriptions should remain reasonably short; the exact description of fine points, special cases, precise details, etc., should be left to a formal specification.

The advantages of brevity cannot be overemphasized. It could even be argued that Naur's specification, once the problems of termination and consecutive break characters are tackled properly, is preferable to Goodenough and Gerhart's because it is shorter and doesn't fuss unnecessarily.

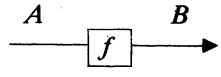
New specification. It would be fair game for the reader at this point to ask what natural-language specification we have to offer in lieu of both Naur's and Goodenough and Gerhart's texts. To answer such as request, we'd try to capitalize on the lessons gained from writing the mathematical definition. We'd propose something like the text in Figure 5, which is directly deduced from that definition (see in particular its relation to Figure 4).

No doubt this text deserves some criticism of its own. In particular, it still needs to be refined. For example, the implementor must know how to "report the error" before embarking upon detailed design and coding; he must know what the allowable characters are apart from *blank* and *new_line,* etc. Also note that this text avoids defining specific concepts (e.g., line length, word) explicitly; rather, it substitutes the definition for the concept when needed. Although this device can lead to interesting literary experiments,[8] it is certainly not recommended for large requirements documents where one must repeatedly refer to the same basic concepts.

It seems to us, however, that the above statement of the requirements embodies the essential elements of the problem and achieves a reasonable tradeoff between the imprecision of Naur's and the verbosity of Goodenough and Gerhart's specifications. (Its length is in fact slightly more than double the former's and half the latter's.) Its most important feature is that it draws heavily from the lessons gained in writing the formal specification, while retaining (we hope) clarity and simplicity.

End-users. An objection that is often voiced against formal specifications relates to the needs of end-users, who request easily understandable documents. Such an objection, we think, is based on an incorrect assessment of what specification is about. There is a need for requirements documents that must be read, checked, and discussed by noncomputer scientists, but there is also a need for technical documents used by computer professionals. The difference is the same as that between user requirements and engineering specifications in other engineering disciplines. Of course, there must be a way to communicate back the contents of technical specifications (for example, in the case of changes). As we have seen, the existence of a good mathematical specification is a great asset for improving a natural-language description.

Other ways can be found for translating formal elements into forms that are more easily understood. Many people like graphical descriptions, which play a basic role in such (nonformal) specification methods as SADT[9] or SREM.[10] A picture may be worth a thousand words at times, but it can also be dangerously misleading. On the other hand, a pictorial explanation of a well-defined concept certainly does no harm. If the picture

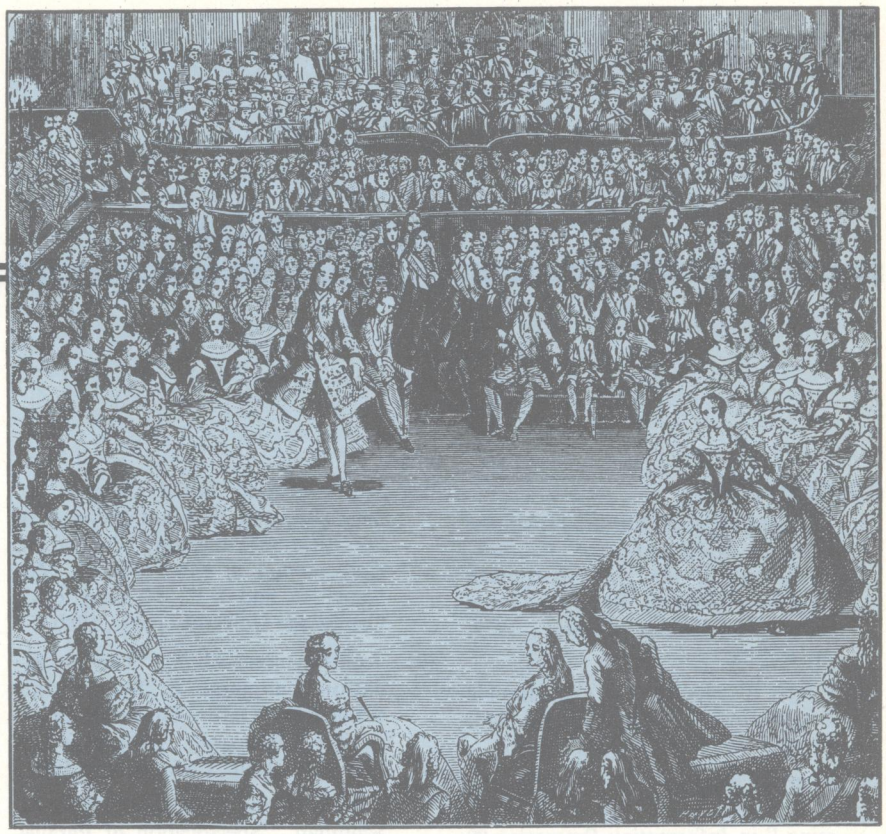


is considered more understandable than the function definition

$$f: A \rightarrow B$$

then why not have graphics tools generate the picture from the formula for the benefit of those who want it? There is certainly a great need for software tools of this kind in specification systems.

Techniques. The last point we want to emphasize is that formal specification *is not necessarily difficult.* The reader who is familiar with specification techniques will have noted that the



Minuet at dress ball given by Louis XV, February 24, 1745, in the armory of the Royal Stables at Versailles; from an engraving by C-N Cochin, "L'ancienne France," in book by the Duke of La Valliere, Louis César de la Baume-le Blanc (1708-1780), *Ballets, Opera, and Other Musical Works*, Ch. J. Baptiste Bauche, Paris, 1760.

example did not rely (at least explicitly) on such notions as abstract data types, finite-state machines, and attribute grammars. In fact, it used only very simple notions from elementary set theory and logic. These notions are no more difficult than the basic core of college calculus, even if most of today's university students are regrettably less at ease dealing with such concepts as sets, relations, partial functions, composition, and predicate calculus than with other mathematical objects and operations that are better established in the traditional curriculum.

Of course, the example studied here is a small problem. Experience with the Z language[11,12] and subsequent work prompted by this experience[13-15] shows, however, that the same basic concepts can be carried through to the description of much more complex systems. The main limitation of the problem studied here is that it is defined by a simple input-to-output relation, whereas most significant programs can be characterized, in our view, as *systems* that offer various services in response to possible user requests. We are currently working on methods, notations, and tools for the modular specification of such systems.[16]

Given are a nonnegative integer MAXPOS and a character set including two "break characters" *blank* and *new_line.*

The program shall accept as input a finite sequence of characters and produce as output a sequence of characters satisfying the following conditions:

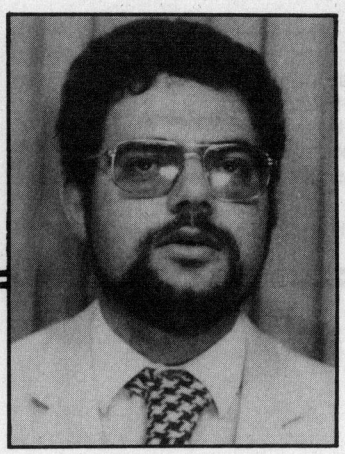
- it only differs from the input by having a single break character wherever the input has one or more break characters;
- any MAXPOS + 1 consecutive characters include a *new_line*;
- the number of *new_line* characters is minimal.

If (any only if) an input sequence contains a group of MAXPOS + 1 consecutive nonbreak characters, there exists no such output. In this case, the program shall produce the output associated with the initial part of the sequence, up to and including the MAXPOS-th character of the first such group, and report the error.

Figure 5. Yet another statement of the requirements.

Reuse. An essential requirement of a good specification formalism is that it should favor reuse of previously written elements of specifications. For example, the notion of sequence and the associated operations should be available as predefined specification elements. Languages Z and Affirm, among others, provide for such libraries of basic specifications. More work is needed to share and reuse the work of formal specifiers. Along with the availability of simple and efficient software tools, this is one of the conditions that must be met before formal specifications become for software engineers what, say, differential equations are for engineers in other fields. □

Formalism



References

1. Bertrand Meyer, "Sur le Formalisme dans les Spécifications," *Globule, Newsletter of the AFCET* (French Computer Society) *Working Group on Software Engineering,* No. 1, 1979, pp. 81-122.
2. Edsger W. Dijkstra, "The Humble Programmer," *Comm. ACM,* Vol. 15, No. 10, Oct. 1972, pp. 859-866.
3. Bertrand Meyer, "A Basis for the Constructive Approach to Programming," in *Information Processing 80 (Proc. IFIP World Computer Congress,* Tokyo, Japan, Oct. 6-9, 1980), S. H. Lavington, ed., North-Holland, Amsterdam, 1980, pp. 293-298.
4. Flaviu Cristian, "On Exceptions, Failures and Errors," *Technology and Science of Informatics,* Vol. 4, No. 1, Jan. 1985.
5. Michael O. Jackson, *Principles of Program Design,* Academic Press, London, 1975.
6. Rod M. Burstall and Joe A. Goguen, "Putting Theories Together to Make Specifications," *Proc. Fifth Int'l Joint Conf. Artificial Intelligence,* Cambridge, Mass., 1977, pp. 1045-1058.
7. I. D. Hill, "Wouldn't It Be Nice If We Could Write Computer Programs In Ordinary English—Or Would It?" *BCS Computer Bulletin,* Vol. 16, No. 6, June 1972, pp. 306-312.
8. Oulipo, *Ouvroir de Littérature Potentielle,* Gallimard, Paris, 1967.
9. Douglas T. Ross and Kenneth E. Schoman, Jr., "Structured Analysis for Requirements Definitions," *IEEE Trans. Software Engineering,* Vol. SE-3, No. 1, Jan. 1977, pp. 6-15.
10. Mack W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Trans. Software Engineering,* Vol. SE-3, No. 1, Jan. 1977, pp. 60-69.
11. Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer, "A Specification Language," in *On the Construction of Programs,* R. McNaughten and R.C. McKeag, eds., Cambridge University Press, 1980.
12. Jean-Raymond Abrial, "The Specification Language Z: Syntax and Semantics," Oxford University Computing Laboratory, Programming Research Group, Oxford, Apr. 1980.
13. Jean-Raymond Abrial and Stephen A. Schuman, "Specification of Parallel Processes," in *Semantics of Concurrent Computation* (Proc. Int'l Symp., Evian, France, July 2-4, 1979), Gilles Kahn, ed., Springer-Verlag, Berlin-New York, 1979.
14. Carroll Morgan and Bernard Sufrin, "Specification of the Unix File System," *IEEE Trans. Software Engineering,* Vol. SE-10, No. 2, Mar. 1984, pp. 128-142.
15. Bernard Sufrin, "Formal Specification of a Display-Oriented Text Editor," *Science of Computer Programming,* Vol. 1, No. 2, May 1982.
16. Bertrand Meyer, "A System Description Method," in *Int'l Workshop on Models and Languages for Software Specification and Design,* Robert G. Babb II and Ali Mili, eds., Orlando, Fla., Mar. 1984, pp. 42-46.

Acknowledgments

I learned most of what I know about specification from Jean-Raymond Abrial. Much of the material was contained in an earlier article, written in French and published in 1979 in a newsletter.[1] I am grateful to Axel van Lamsweerde for reminding me of the existence of that article and suggesting that it might be of interest to a wider audience (and to him and Jean-Pierre Finance for some heated discussions on specification). I also thank Flaviu Cristian for important comments on a previous version. The referees' comments were also useful.

This article also benefited from the involuntary contributions made by the authors of all the system requirements and other software documentation I have had to struggle with over a number of years.

Bertrand Meyer is a visiting associate professor at the University of California, Santa Barbara, on leave from Electricité de France, where he was division head in the research and development branch. He is interested both in doing research and in making the results of research useful to practitioners. He has published papers on programming methodology, software tools and environments, specification, interactive systems and user interfaces, algorithms, programming languages, and supercomputer programming, as well as a compendium on programming methodology and techniques, *Méthodes de Programmation* (Eyrolles, Paris, 1978, with Claude Baudoin). He is the editor-in-chief of the French computer science journal *Technology and Science of Informatics,* a member of ACM, AFCET and the IEEE, and a former member of the AFCET council.

Meyer's address is Department of Computer Science, University of California, Santa Barbara, CA 93106.