NORTH-HOLLAND

# Specifications Are Necessarily Informal or: Some More Myths of Formal Methods*

Baudouin Le Charlier
*Institut d'Informatique, University of Namur, B-5000 Namur, Belgium*

Pierre Flener
*Department of Computer Engineering and Information Science, Bilkent University, 06533 Bilkent, Ankara, Turkey*

We reconsider the concept of specification in order to bring new insights into the debate of formal versus non-formal methods in computer science. In our view, the correctness of a useful program corresponds to an objective fact, which must have a simple, precise, and understandable formulation. As a consequence, a specification can (and must) only make precise the link existing between the program (formality) and its purpose (informality). Moreover, program correctness can be argued only by means of non-formal reasonings, which should be as explicit as possible. This allows us to explain why specifications cannot be written in a strictly formal language. Our view of specifications does not imply a rejection of all ideas put forward in the literature on formal methods. On the contrary, we agree with the proponents of formal methods on most of their arguments, except on those following from the assumption that specifications could (or should) be formal. Finally, we examine why the role and nature of specifications are so often misunderstood. © 1998 Elsevier Science Inc.

## 1. INTRODUCTION

*Gist specifications were nearly as hard to read as those in other formal specification languages. We soon realized that the problem was not particular to Gist, but extant across the entire class of formal specification languages. In their effort to be formal, all these languages have scrubbed out the mechanisms which make informal languages understandable, such as summaries and overviews, alternative points of view, diagrams, and examples.*
—*R. Balzer [1985].*

Recently, there have been numerous papers advocating the use of formal methods in software development (e.g., [Bowen and Hinchey, 1995a, 1995b; Craigen et al., 1995; Fraser et al., 1994; Gerhart et al., 1994; Gibbs, 1994; Hall, 1990; Larsen et al., 1996], plus some of the opinions in Saiedian [1996]). Similar opinions were sporadically published before (e.g., [Fraser et al., 1991; Guttag et al., 1982; Hoare, 1987; Meyer, 1985; Wing, 1990]), plus some of the opinions in Denning (1989). In these papers, members of academe and industry describe formal methods as a key contribution to overcoming the chronic software crisis. Indeed, formal specification languages force specifiers to be absolutely precise about their intentions, since (internal) inconsistency and incompleteness can be mechanically detected. Moreover, formal specifications can be used during validation by the customer through animation or prototyping, and can guide the actual development of the software, or at least be used in the formal verification of the developed software. All this is proposed at various degrees of formality, from fully formal to "formal light."

However, fallacies in some assumptions underlying formal methods have been exposed, such as by

*Address correspondence to Dr. Pierre Flener, Bilkent University, Department of Computer Engineering, 06533 Bilkent, Ankara, Turkey. E-mail: pf@cs.bilkent.edu.tr*

**Contrary to previous papers of the "myths series" [4, 19], we do not discuss industry-level myths on the uselessness of formal methods, but rather analyze some academic myths on their usefulness.*

0164-1212/98/$19.00
PII S0164-1212(98)00172-6

pointing out essential differences between engineering and mathematics in general, and between computing and mathematics in particular (see other opinions in [Denning, 1989; Saiedian, 1996]), or by shedding some light onto the real nature of requirements and specifications, so as to identify minimum standards for languages allowing their representation (Jackson, 1995; Zave and Jackson, 1997). Some authors have been begging for caution about formal methods, by mentioning fundamental theoretical and practical problems, e.g., DeMillo, Lipton, and Perlis (1979), Fetzer (1989), Karp (in [Denning, 1989]), Parnas ([1994], and in [Saiedian, 1996]), and Winograd (in [Denning, 1989]).

A similar debate is going on about the teaching of computer science (Denning, 1989) should the curriculum include formal methods or not? To what extent?

Simultaneously, there is a debate on whether formal specification languages ought to be executable or not (Fuchs, 1992; Gravell and Henderson, 1996; Hayes and Jones, 1989). However, some researchers challenge the contention that specifications ought to be (fully) formal in the first place, e.g., Balzer et al. (Balzer, 1985; Balzer et al., 1986), Karp (in [Denning, 1989]), and Parnas ([Parnas, 1994; Parnas and Madey, 1995], and in [Saiedian, 1996]).

Our objective is to shed some further light onto these debates. We propose to go back to the very reasons that make the running of a program useful, i.e., the fact that its results can be straightforwardly interpreted as a statement about the real world. Starting from this simple observation, we draw the conclusion that the specification of a program only consists of (the statement of) the link relating the program (formality) and its purpose (informality). Since, as we will argue, the purpose of a program must be something directly understandable, specifications also are the essential tool for constructing, in practice, correct real-world programs through explicit but non-formal reasonings. Additionally, our discussion of specifications allows us to explain why formal specifications (i.e., specifications written in a formal specification language) are not really specifications, since this would be a contradiction in terms. Several researchers in formal methods have recently reported insights related to ours, namely that informal "comments" are inevitable adjuncts to formal specifications (Hoare, 1996; Jackson, 1995; Wordsworth, 1992; Zave and Jackson, 1997), or the fact that the knowledge of the environment in which the program will be embedded is essential to the understanding and the writing of specifications (Johnson, 1988). But our reflection goes, in a sense, beyond their conclusions, since we claim that specifications are, or ought to be, informal by their very role.

Our view of specifications does not imply a rejection of all ideas put forward in the literature on formal methods. On the contrary, we agree with the proponents of formal methods on most of their arguments, except on the fact that specifications had better be written in a formal, i.e., completely predefined and syntactically checkable, language. And, inevitably, we also disagree with other arguments that are a consequence of this assumption that formal specification languages are desirable.

Formal methods are in general introduced as being the use of mathematics in the process of constructing computer software (including the elaboration of specifications). We agree that mathematics are extremely useful in this context, but we disagree on reducing the concept of mathematics for computer science to the restricted framework of any formal specification language.

Program verification is advocated by most distinguished computer scientists as the only way to improve the quality of software. We agree that program verification or, better, systematic program construction is the only way to build satisfactory computer software, but we disagree on the fact that program proofs must be automated, since, as we try to demonstrate, this would imply a vicious circle.

Requirements engineering is viewed by most authors as the most crucial stage in the development of a large software system. We agree on this viewpoint and especially on the importance of the elicitation process, but we disagree with the opinion that writing formal specifications is the best basis for the elicitation process: such a process is best achieved in a language as expressive as possible, i.e., a natural language enhanced with any desired notational conventions.

Finally, it is generally accepted that formal methods should be supported by corresponding software tools. We argue that formal descriptions of any kind (programs, finite-state automata, "declarative" descriptions, and the like) can be useful only because they can be the input of an automated process whose output provides directly understandable information that could not be realistically discovered by manual calculation. Nevertheless, the elaboration of any formal description (of whatever nature) requires a careful construction process that cannot be formalized in any way since this would entail a *regressum ad infinitum*. Note that we do *not* say that such tools are useless, but only that the crafting of their inputs already is a programming activity whose mastering definitely requires explicit informal reasoning.

We conclude this introduction by summarizing the articulation of our argumentation along the three main sections of the paper.

**Section 2.** Our main thesis, i.e., the fact that explicit informal reasoning is the essential pivot of any well-conducted programming activity, is the subject of Section 2. Such reasonings are best based on clear specifications of all sub-problems that are identified during the program construction process (including the requirements engineering phase).

Most of Section 2 (i.e., Sections 2.1 to 2.4) is devoted to demonstrating that such specifications should (and in fact can) be made extremely simple by clearly separating the statement of the purpose of the program (which should boil down to citing a well-known concept) and a set of representation conventions (whose role in informal reasonings is subordinate yet necessary since the concepts of the programming language are totally alien to the problem that the program must [help to] solve). Section 2.1 motivates our notion of specification by showing through some example problems that the results of a program are meaningless by themselves and should be interpreted in some way to allow the resolution of the problem that the program helps to solve. We also show that this interpretation necessarily takes place at an intuitive (problem-related) level. Finally, if the program is really convenient to use, it is necessary that the interpretation of the results be extremely simple. Section 2.2 draws an important conclusion from these observations, i.e., that a specification should only 1) state the purpose of the program (in a straightforwardly understandable way) and 2) state the representation conventions that one needs to know to use it properly. Section 2.3 explains why such specifications are essential to articulate the programming activity, while Section 2.4 argues that it is actually possible to craft such specifications even for "real-world" problems.

The intuitive knowledge necessary either to properly use a program or to construct it is generally not available at the beginning. In order to write good specifications of the program and of all its parts, one thus needs to build a "theory of the problem" that provides this necessary knowledge. Section 2.5 is devoted to this topic: we revisit a few classical problems in order to show that the main role of this theory is to identify useful properties of the actual, real-world problem, not of a more or less arbitrary and unreasoned redefinition of it; since the objective is to understand the problem as it is, we also dispute the idea that it is necessarily better to define concepts as abstract data types or in nonexecutable style.

Section 2.6 summarizes our ideas by discussing the "general form" of specifications, while Section 2.7 draws a parallel between requirements specifications and our notion of "theory of the problem."

**Section 3** applies the ideas of Section 2 to a critique of the concept of formal specification. Since the concepts of a formal specification language are totally alien to those of any practical "real-world" problem, specifications in our sense cannot be written in such formal languages. Moreover, the correct construction of (what is usually called) formal specifications requires the use of (informal) specifications in our sense. In fact, all our argumentation of Section 2 applies as well to formal specification languages and to programming languages. This thesis is developed in Section 3.1. It allows us to discuss seven frequently asked questions about formal specifications, in Section 3.2.

**Section 4.** Finally, we try to explain why our view of specifications has not been largely accepted by computer scientists. The belief that all practical[1] mathematics can be embodied in a single formal system is—we guess—a main reason of the importance given to formal specifications. Another important reason is the desire to find methods to measure the value of a program and the programmer's productivity. In our opinion, such a goal is largely unreasonable.

This paper is based on the Ph.D. dissertation of the first author (Le Charlier, 1985) (and includes translations of tracts of this thesis). The first author has successfully used these ideas in several medium-sized projects (Le Charlier and Flener, 1997). The second author has used them for debunking some of the myths on deduction-based and induction-based approaches to the (semi-)automatic synthesis of (logic) programs (Flener and Popelínský, 1994).

## 2. THE ROLE AND NATURE OF SPECIFICATIONS

In this section, we more closely examine specifications of programs. Such specifications are the essen-

[1]From a theoretical standpoint, this belief has been ruined by Gödel's incompleteness theorem, but formalist mathematicians argue that the limitations pointed out by Gödel have no impact on mathematicians' practice.

tial pivot of the whole programming activity: without good specifications, it is impossible to understand what the correctness of a program means and hence to reason *rigorously* while constructing it or constructing another program using it. In the software engineering literature, the word "specification" is used to designate many different kinds of things (such as requirements specifications, for an entire software, and detailed-design specifications, for its modules), and yet there is something in common to all of them. For the moment, we deliberately do not make precise the kind of specification that we consider, but we will come back to this issue in Section 2.7.

## 2.1 Why and How can a Program be Useful?

Despite all the doubts one might have about the purpose of computers for the resolution of real problems such as the creation of a more just and harmonious society, if one writes and uses programs then it is because one believes they are useful. This fact is so evident that one never wonders why and how a program can be useful. However, it is the answer to that question that leads to an understanding of what programming is and why specifications play a fundamental role in it.

If a program is useful, it is not because its execution results in displaying certain strings on the screen or in changing the contents of the computer memory in a certain way. It is because this execution yields useful information or provides substantial help in the realization of a task. But, to take advantage of the program, other things than its text and the format of its data need to be known. Even observing its behavior for some time does not suffice. It must be possible to interpret the produced results, but the knowledge necessary for this cannot be part of the program text nor of its results. It is relative to concepts totally alien to the objects manipulated by the program, and to the conventions according to which these objects represent these concepts.

**Example: The Belgian National Lottery.** Suppose all we know about a certain program is how to launch it on a certain computer and that its execution only results in displaying the string:

5, 11, 15, 22, 29, 46

No information can be drawn from this; our lives are unaffected by the knowledge that the execution of a certain program gives exactly this result. Now suppose, to the contrary, that we know from an informed source that the execution results in displaying the next draw of the Belgian national lottery. This changes everything: everybody now sees how such a program can be used advantageously....

This single example shows why a program is "not useful" by itself, but only in conjunction with some knowledge that is totally outside of it, of which neither its text nor its results can give the slightest clue. Some will now object that it is easy to change that program so that it exhibits its own purpose, say by displaying the following string instead:

5, 11, 15, 22, 29, 46
is the next draw of the Belgian national lottery.

But this objection is flawed for two reasons. First, it is not the simple observation of the result that allows us to understand it. The act of "seeing" the string above cannot possibly give the knowledge necessary to the understanding of the sentence it represents. This knowledge must be available before or must be acquired by other means. Second, it is not enough to be able to interpret the result of a program by an assertion in order to deduce from it whether it is true. To do so, there should be other good reasons to believe that an execution of a program can only produce outputs that represent true assertions.

Finally, if a program can be useful, even though its manipulated objects have by themselves no meaning, it is because it is possible to use these objects to represent useful information so as to be able, first, to write the program so that it computes the representations in a correct way (according to chosen conventions), and, second, to "easily finish the job" by interpreting the results.

**Example: A payroll program.** Let us now consider the payroll program of a company. It is useful to the extent that it is easier to (correctly) solve the payroll problem with it than without it. In any case, it is not the running of the program that solves the problem. The problem is solved if and only if the whole personnel gets their due salary at the deadline. This happens or does not happen independently of the existence of a payroll program and its results. The responsibility of the solving of the payroll problem belongs to the corresponding accountant. The program can only help her as an intermediary and is only really useful if it noticeably reduces the amount of work the accountant has to do to solve the problem. The accountant's task is, on the one hand, to prepare the inputs to the program, and on the other hand, to exploit its results so that all employees get their salary. So she must know how to use the program. This also means that she must be able to make a *reasoning* by which, knowing the inputs, knowing the usage she made of the outputs, and knowing "sufficiently many things" about the program itself, she can conclude that everybody's exact salary is paid at the deadline. Nowadays, the accountant may have almost nothing to do to complete her task, but some verification (of whether the program performs its task) has to be done nevertheless.

**Example: A search sub-program.** Let us finally consider a sub-program that locates a value in an array. It is useful because one can use it as a primitive for writing a larger program, and this without worrying about how the search is done. However, to use it properly, some supplementary information must be available: how to call the sub-program and how the results are represented. One might think this example is fundamentally different from the first one. In this case, some will say, to understand the purpose of the program it suffices to know the programming language and to have the text of the program. Indeed, the latter would be so simple that one will "immediately see" what the program does. The text would define the purpose of the program. This opinion is incorrect: to understand the purpose of the program, the concept of membership in an array must be known in advance, but it is not a concept of the programming language because otherwise it would not have been necessary to write a sub-program representing it. The opinion above stems from the fact that one might recognize quite easily an array search in the program text provided one has already done some programming beforehand, hence one already knows what an array search is, for what it can be used, and what form one generally gives to programs performing it. But this does not mean that this knowledge can be derived from the program text.

This example has been chosen on purpose among the most simple and "classical" ones. It is clear, however, that in general one does not write programs solving known problems. Therefore, the knowledge of some programming concepts and methods is totally insufficient for understanding not only the purpose of a "large" program but also the one of most of its components. To understand the use of a program computing *sin*($x$) according to given representation conventions and a given precision, trigonometry and analysis notions must be known. Pretending that the program defines the corresponding approximation is only a pleasant joke, because it is not the scrutiny of this text that can give the slightest idea about trigonometry to somebody who does not already have it.

Finally, it often happens that the concepts necessary to understanding the purpose of a (sub-)program cannot be found in our "preliminary knowledge" but must be invented ad hoc. It is well-known that the resolution of a simple problem may necessitate the introduction of completely new ideas. Such invention is done via definitions. But there would be a vicious circle to try and explain the purpose of a program by referring to concepts only known by their definitions: this would almost amount to saying that this purpose can be understood by examining another program. To leave this vicious circle, it is necessary to give these newly defined concepts an intuitive and objective "substance," by shaping them into a theory allowing their understanding without any definitions. These ideas will be further developed in Section 2.5.

Note that there is an important difference between our notion of specification and the notion of requirements specification, which consists of a description of the problem to be solved. In our view, this notion should essentially coincide with what we call the "theory of the problem." Again, we refer to Section 2.7 for more details on this issue.

## 2.2 What is a Specification?

**"Definition."** A program specification is a statement whose role is to say (1) what purpose the program serves *and* (2) how the program can be correctly used.

This "definition" is not a mathematical one, but the previous discussion will help us to understand it in detail. The definition means that the specification of a program is the necessary link between what the program computes and the information that we can deduce from its results. This link is exactly what we need to use the program or to construct it.

**A specification must be simple and directly understandable.** The objective of a specification is to transmit information. So there is a parallel between the notions of specification and program output. The output is meaningless by itself: it must be interpreted in order to extract the information it carries. This does not mean the particular form of the outputs is irrelevant as long as the representation conventions are known. For instance, if the task of a teller machine in Belgium is to display the balance of a bank account, then not all representations are equivalent: a decimal representation of the amount expressed in Belgian Francs is acceptable, but a binary representation of the square root of the amount expressed in Turkish Lira is not. The good representation is the one that minimizes the work that remains to be done to transform the output into the desired information. In the example above, the first representation is the only acceptable one because the customer immediately knows how much money can be withdrawn from the account, whereas a long and tedious computation would be necessary from the second representation. Similarly, the *"good" specification of a program is the text that can be transformed as directly as possible into a correct understanding of the purpose of the program and of the way of using it.*

Besides this analogy, there also is a fundamental difference between a specification and the results of a program. The principal role of the specification precisely is to state how to interpret the results, but there is no need for a text explaining how to interpret the specification, as otherwise one would need a specification of the specification, and a specification of the specification of the specification, ad infinitum. *Therefore, unless one completely denies the pertinence of this notion, one has to admit that a specification is a text that must be comprehensible by itself. Hence it must be written in the only language adapted to this end: natural language. We do not say that specifications ought to be written in pure natural language. It can be a technical language including problem-specific concepts and notations. But it cannot be a formal language, in the strict sense of the word (i.e., whose syntax and semantics are defined a priori).* Indeed, statements in a formal language are incomprehensible by themselves (also see Section 3.1), because the problem concepts are always totally alien to those of the formal language. Hence, formal statements always need to be accompanied by explicit representation conventions, i.e., informal specifications. To the contrary, informal (natural language) statements are comprehensible by themselves because they directly refer to the problem concepts.

**A specification need *not* be correct, but only correctly understandable.** Since the role of a specification is to communicate the purpose of a program, the only correct means of judging the quality of a specification is to ask whether it allows every potential reader to understand conveniently and in the most direct possible way the purpose of the program.

The notion of "correctness" of a specification is thus less important than the one of "being correctly understandable." A specification can perfectly play its role, even if it lacks style, or has unorthodox phrases, if not even mistakes and contradictions.[2] A reader may well have understood it even though she estimates it to be "incorrect" or poorly written, because it does not follow her own stylistic criteria or contains some obvious mistakes. But how is it possible to correctly understand a specification while judging it incorrect? The answer lies in the observation that the role of a specification is not to define everything that ought to be known to understand the purpose of the program, but only to *state* this purpose. Where is the difference? According to the first viewpoint, one would suppose that the knowledge necessary to use the program is entirely inside the specification (i.e., would be derivable from the specification). It would, then, be evident that an incorrect, specification cannot be satisfactorily understood by itself because it would be the only reference. According to the second viewpoint, one supposes that the reader already knows almost everything on what makes the program interesting, the role of the specification being somehow to say "this is the program that you needed." In this case, the presence of some errors or quirks in the specification would not really be an insurmountable obstacle to its understanding, because the enormous quantity of things already known allows the reader to fill the gaps.

All this does not imply that specifications can be written carelessly, but only that the quality of specifications cannot be judged according to hypothetical correctness criteria. The key issue is that they communicate "the message" in the most direct way. This entire argument holds of course for *all* consumers of specifications, be they end-users, or programmers, or whoever. Correctness is relative to an external truth criterion, and the objective is to make a software correct with respect to a fact, but not with respect to a statement in a formal theory.

## 2.3 Why are Adequate Specifications Necessary?

The specification of a program is an indispensable aid for remembering details. After close consideration, it is even *only* such an aid, as it only has to

[2]In our view, whenever a program addresses a meaningful problem, there is a model in the real world for any "correct" theory of the problem. If we fail to build this correct theory, this does not mean in any way that the model does not exist, since it is preexisting (unless we deny that the world exists). That is why we dispute the importance of self-contradiction in a theory. A theory can be self-contradictory because of a single fortuitous mistake and yet one can be able to "see" the intended model underlying it. Self-contradiction can be problematic for technical reasons in formal theories, but of course we also dispute the idea that the theory of the problem must be a formal one.

state the purpose of the program but not all the knowledge necessary to understand its meaning. The customer must thus already know, before reading the specification of a program for the first time, everything that makes the program useful to her. She will then know that a program with this purpose exists and how it can be used. Later, she can occasionally re-read the specification, not because she has forgotten its purpose, but because she does not recall with certainty some representation details that are too arbitrary to be possible (or useful) to remember.

Specifications are not only absolutely necessary for documentation of already existing programs, but also before and during construction of programs, for three reasons.

First, one can only construct small programs at a time. The difficulty observed in the rigorous construction (à la Dijkstra, Gries, etc.) of small programs is inherent to programming (and there is no way such techniques can ever be scaled up to constructing "real" programs), so small programs exactly represent the limit that should not be crossed if the programming activity is ever to be mastered. The only realistic approach is thus to build "large" programs from "small" ones that are constructed independently of each other, and recursively so on (no matter whether one proceeds top-down or bottom-up). This is possible only because the specifications attached to programs allow us to consider them as new primitives of the programming language, no matter how large these programs are.[3] *All specifications should be of the same level of complexity, namely of the utmost simplicity.*

Second, intermediate specifications, i.e., specifications of sub-problems perceived as potentially useful during the design of the system's architecture, are necessary as a basis for the discussion between the computer scientist and the customer, because they are, in general, of too different backgrounds for coming up with the good specification the first time. Starting from the specification, the computer scientist must be able to make a reasoning to convince herself that she can construct the required program, whereas the customer must be able to make a reasoning to make sure the program will provide the expected service. The specification thus takes the role of a contract.

[3]Note that we do not assume a pure hierarchical organization of programs. For concurrent programs, for instance, a specification could (essentially) consist of a global invariant and some fairness properties.

Third, intermediate specifications are necessary during the design of an architecture for the program. Strictly linear top-down design is difficult, and the implementation of certain sub-problems may reveal inadequacies in earlier choices, forcing backtracking in the design, if not the deletion of already written code. Since programming is costly, there is a risk of trying to preserve at all cost what has already been done, even if this means going into blind alleys. A more reasonable approach is thus to write all specifications of all sub-programs before writing the first line of code. This requires mental persuasion that the program can be written using all and only the specified sub-problems. Designing such an architecture may still require backtracking, but it is less tedious to rewrite specifications than programs, and easier to persuade oneself that a program can be written than actually writing it.

## 2.4 Can there be Adequate Specifications (for Real-World Problems)?

We think that adequate specifications, according to our criteria, *can* be written, *even for real-world problems*. We know that most examples in this paper are small-scale—and space reasons prevent us from covering real-world problems—but our considerations *do* scale up, by their very nature. The first author has successfully applied them to rather complex medium-scale problems, as reported in (Le Charlier and Flener, 1997), and he believes that he was successful precisely because of this mind-set.

**A specification is not meant for everybody.** Only a program with a precise purpose should have a specification. Saying that a program has a precise purpose amounts to saying that somebody is able to exactly understand this purpose. So the specification of a "useful" program will always exist because somebody must be able to *say* what its purpose is. But this does not mean that everybody can understand this specification. It is only comprehensible by somebody having the "same background" as its author, at least as far as the application domain is concerned. The existence of satisfactory specifications is thus only possible because they are only meant to be read and understood by people already knowing almost *everything* of the application domain in which the program has its purpose. This does not imply that only the specifier will be able to understand it or that this privilege is reserved for a select few. It simply means that every user of the program must first make a careful and sufficiently long study of its application domain.

**Remark.** In practice, it is unfortunately rare that a person understanding the purpose of a program can express it simply. Programmers, for instance, tend to give incomprehensible technical gibberish about the implementation technique and run-time behavior when prompted to explain what their programs do, instead of talking about the essentials. The absence of specifications for many actually used programs stems from an inability of many people to express themselves clearly. (As already said by others before:) *Instead of including specification rules or formalisms in computer science curricula, it would be much better to teach students how to correctly use their native language (or natural language, in general).*

Another reason for the absence of convenient specifications is that programs are often constructed by successive approximations, by trial and error, so that there cannot possibly be a convenient specification, because nobody is able to understand how to use it. But it is precisely because the programmer was unable, or thought it useless, to write a specification that she, not knowing what to do and hoping to find it out progressively, constructed a mysterious program to which no specification can be attached.

**A specification should have an objective meaning.** Some will object to our notion of specification by saying that two different people never understand things in exactly the same way, so that we can never be sure whether a specification is correctly understood by all concerned people. However, it is not necessary that the programmer and all users of a program understand its specification in the same way. Note that such a condition is insufficient anyway, because it does not matter whether all people have understood exactly the same thing, but rather whether everybody has understood what is needed to do their job. And this new condition can be fulfilled because the specification of a program must express a property that has an objective meaning. It is true that nobody understands this meaning completely and in the same way as their neighbor, but everybody should understand that the question of correctness of the program with respect to its specification corresponds to a *fact*, and not to personal interpretation. The programmer must be able to construct the program by making a reasoning to persuade herself that it has the desired property; whereas the users must be able to derive other facts from it, such as the possibility of doing their job using the program.

For instance, consider a program computing the sine function under certain precise conventions. The programmer need not completely know the "essence" of this function, but only sufficient properties for constructing a correct program. The users need not understand the function in the same way as the programmer, but only other properties allowing them to solve their problems. So it is because of its objective nature that the specification of this program will be satisfactory: it expresses a fact, the same for everybody, even though they may understand it differently. Hence the specification should act as the "ultimate" reference, i.e., the last thing to be doubted about and hence the central pivot of any reasoning about the program.

A not completely unfounded objection to the previous example is that it is not realistic because the sine concept has been studied for such a long time that it would be foolish to deny its objective nature, but that not all specifications can be expressed in terms of such well-established concepts. Indeed, this objection pinpoints one of the fundamental difficulties of programming compared to, say, mathematics: one never has the time to polish all the needed concepts for a specification, because the program is needed urgently.

Nevertheless, the objectivity condition for specifications seems absolutely necessary for the correct communication of the purpose of programs, and, hence, for mastering the programming activity. According to us, without this condition, one would have to admit that the usage of programs for achieving a certain activity amounts to redefining that activity as being the exploitation of the results of the program without giving a satisfactory link between this redefinition and the initial concrete problem. Moreover, to us, this condition seems largely achievable, if one admits that the objectivity of the concepts necessary to the writing of good specifications can be founded on the creation of a "theory" of these concepts, with more or less detail according to the imperatives of the problem, a theory that can be studied by all concerned people until each of them has convinced themselves *personally* that it really corresponds to the intended object.

This perception of course has the "disadvantage" of founding the mastery of programming and its usage on the competence and responsibility of people, whereas some would prefer to found them on rules that are easy to apply and verify.

## 2.5 Role and Content of the "Theory of the Problem"

The intuitive knowledge necessary either to properly use a program or to construct it is generally not available at the beginning. In order to write good specifications of the program and of all its parts, one thus needs to build a "theory of the problem" that provides this necessary knowledge. In this section, we revisit a few classical problems in order to show that the main role of this theory is to identify useful properties of the actual, real-world problem, not of a more or less arbitrary and unreasoned redefinition of it; since the objective is to understand the problem as it is, we also dispute the idea that it is necessarily better to define concepts in an abstract data type or in non-executable style.

Since a program is normally constructed in order to help solve a practical preexisting problem, the concepts and objects of such a theory can be classified into two main categories: those whose identity was determined before and independently of the program, and those that are defined (or, better, identified) especially for the construction of the program. They should all have the same final status, namely to be known not by their definitions but by a sufficiently rich set of properties linking them to numerous other concepts. They thus have their own individuality, equivalent to an objective status. The theoretical development necessary for achieving this status is different and more or less long and difficult according to the category of concept. We elaborate on these issues in Sections 2.5.1 and 2.5.2. In Sections 2.5.3 and 2.5.4, we discuss some aspects of definition construction, stressing that there is no purely abstract way to define a concept and that the non-executability of definitions is not necessarily a desirable objective.

**2.5.1 On the Study of "Long-Established" Concepts.** Defining once again preexisting concepts is common practice in formal methods of program design. It is however unwise to start the study of a predetermined concept by defining it. Indeed, what is necessary is to study the concept as it is, but not another concept given the same name through a definition. Even if a "predetermined" concept can be considered completely determined by a certain property (i.e., all other properties useful to the problem at hand can be derived from that property), one cannot consider it a definition of the concept. On the contrary, one would have to ensure that the concept really has that property. The objective of the theory to be built is to ensure that things are sufficiently well-understood by all involved people. If one started redefining all the fundamental concepts of the problem, nothing would be known about the relationship between the (preexisting) problem and what has been done. In any case, all involved people have a preliminary understanding of the problem. The role of the theory is to make things precise, if not to correct them, but not to reconstruct everything from nothing. It is thus more important to stress the difficult or delicate issues than to try and found everything already known.

**The case of mathematical concepts.** Suppose the concept of "greatest common divisor" is needed in the resolution of a programming problem. It is not the following redefinition of this concept that makes its role in the problem more precise:

**Definition 2.1.** The *greatest common divisor* of two natural numbers $m$ and $n$ is a natural number $p$, denoted by $gcd(m, n)$, such that $p$ divides $m$ and $n$, and, for every natural number $i$, if $i$ divides $m$ and $n$, then $i$ divides $p$.

Indeed, if one does not already know the concept of greatest common divisor (gcd) and its applications, this definition will not, by itself, help one understand its purpose. But let us consider a person who already has a good idea about it. The only information she can draw from this definition is that it probably is the definition of the notion of greatest common divisor that she already knows. Therefore, *the only immediately useful part of this definition is the only word that is theoretically arbitrary*! Indeed, one could define the same concept by naming it "foo" or "Nabuchodonosor." Two things are possible from here. Either this person is satisfied with her conclusion, and then the definition has not brought any new information, or she wants to verify this first impression by examining whether the definition is compatible with her existing knowledge of the concept of gcd. In this case, she might not be able to do so immediately, because her definition rather says that $gcd(m, n)$ is the greatest of the divisors of $m$ and $n$, according to the usual ordering relation. To show that the two concepts coincide, she actually has to make a long reasoning, which should by the way conclude negatively, because they do not coincide when $m = n = 0$ (where the greatest common divisor is usually considered undefined, but the definition above gives $gcd(0, 0) = 0$). Anyway, at the end of this superb intellectual effort, she will still not know whether this definition was introduced for the fun of scrambling the message or for some better reason. To conclude, it would have been better to admit that the concept of gcd is predetermined beyond all definitions and to show why the very close concept of greatest common divisor according to the "divides" ordering relation was substituted for it. For instance, it could have been because one wanted to be able to apply, in all cases, the formula $gcd(m, gcd(n, p)) = gcd(gcd(m, n), p)$. (For $m = n = 0$ and $p \neq 0$, only the left-hand side of this equality is defined according to the usual definition.)

**The case of "non-mathematical" concepts.** The preceding precept applies unchanged to any kind of problem. It is not because the program to be written has its purpose in, say, an accounting setting, that one has to start by defining all involved concepts in order to understand its purpose.

For instance, in the payroll program, the "theory" of the problem should not start with definitions of employees, salaries, companies, etc. What is necessary is to arrive at a sufficient understanding of these concepts (which are perfectly determined, even if they might be poorly understood at the beginning) in order to solve the problem. It would not be acceptable either to define the effect of the program by the rules of computing the salaries in terms of the employee database. One should study the legislation, the structure of the company, etc., in sufficient detail so as to be able to *deduce* (i.e., to justify, by a rigorous reasoning) an adequate structure for said database as well as valid computation rules. The user of the program (i.e., the accountant) need not have studied all the details of the "theory" that the programmers have had to elaborate, but she should understand it sufficiently for correctly using the program. It would be hard to say where the limit is: it is her responsibility to decide herself how far to go in order to reach a sufficient understanding.

**2.5.2 On the Study of Concepts "Tailored for the Problem".** The writing and understanding of "good" specifications of programs nearly always requires using concepts especially tailored for the problem, discovered or created especially for constructing the program. Such concepts can only be introduced by definitions, but those definitions must be validated explicitly against our perception of the problem in order to ensure that the concepts adequately relate to the "real" problem.

**The case of simple concepts that are close to known ones.** One often has to deal with concepts that can be considered already implicitly known and understood by all people who have to use them, but whose relevancy is insufficient for having been given a name that is universally admitted. It is then necessary to have recourse to a definition for naming the concept and making everybody agree on some important details whose identification is necessary for correctly using it. When reading such definitions, it should be possible to "immediately see what they are about." The concept-specific theory then reduces to only a few things, because the concept "naturally takes its place" among already known ones.

Let us illustrate this with a specification of the classical plateau problem.

**Definition 2.2.** Let $S = (s_1, s_2, \dots, s_n)$ be a finite non-empty sequence of integers. A *plateau* of $S$ is an interval[4] $\langle i : j \rangle$ such that:

1. $1 \le i \le j \le n$
2. $s_i = s_{i+1} = \cdots = s_j$
3. $\langle i : j \rangle$ is not strictly included in any other interval with properties (1) and (2).

**Problem:** Given a non-empty initialized array $a[1..n]$ of integers, construct a program that assigns to integer variable *np* the number of plateaus of the sequence $(a[1], a[2], \dots, a[n])$, and to integer variable *maxlp* the maximum of their lengths (the length of a plateau is the number of its elements).

The definition in the specification above is sufficient for a satisfactory problem statement, for two reasons. First, the "technical" concept of plateau is not brand-new, but rather a particular and precise occurrence of a more general concept that we already know (the choice of the name "plateau" is thus *not* arbitrary). Second, this definition is sufficiently simple for linking this particular concept to the general one, that is for verifying whether the chosen terminology really corresponds to something intuitive. Moreover, the definition is necessary, because the intuitive notion of plateau is too vague for being able to rule out, in its absence, a misunderstanding of the notions of number and length of the plateaus of a sequence.

**On the usefulness of examples.** Specifications may be accompanied by carefully chosen examples, so as to facilitate their understanding. Since the role of a

[4]We assume the concept of interval is already known: $\langle i : j \rangle = \{x \mid x \text{ is an integer and } i \le x \le j\}$, where $i, j$ are integers.

definition, as considered here, is not to be formally irreproachable (i.e., non-contradictory, for instance), but to help understand something, there is no reason to reject other means of communication that might have other qualities. Some well-chosen examples often provide an intuitive understanding that no definition could achieve. The latter then only makes more precise the exact contours of the concept. Other examples could help eliminate certain risks of ambiguity in the definition by illustrating delicate issues that are likely to be misunderstood for whatever reason.

As far as the risk of contradiction between definition and examples is concerned, note that this kind of contradiction would only be a real disaster if it were the non-contradiction of a definition that would lend value to a concept. There is a confusion here between truth and non-contradiction. What is important is to make known what one wants to say, not to escape contradiction. One could even argue that the discovery of a contradiction between an example and a definition is the best thing that can happen in some cases, because it carries an undeniable message: something is wrong somewhere!

Personal experience[5] shows that a "poorly defined" concept can be perfectly understood thanks to examples, especially when the concept can be considered already implicitly known. Definition and examples are thus complementary means of designating the concept. And one may well conclude that there is only one concept corresponding to both the definition and the examples, even if one has spotted an apparent contradiction between them. What one already knows helps understand the error. Finally, note that the error risk is much higher in a definition than in an example, because it has to cover all cases. Examples are more reliable, because more "local," and are thus an ideal means of getting things straight.

Let us illustrate this on the plateau problem. Assume condition (3) was omitted from the definition above, but that the following example was added:

**Example 2.1.** If $S = (1,1,3,3,3,2,3,5,5)$, then there are 5 plateaus of $S$, namely $\langle 1:2\rangle$, $\langle 3:5\rangle$, $\langle 6:6\rangle$, $\langle 7:7\rangle$, and $\langle 8:9\rangle$. Also, its longest plateau is $\langle 3:5\rangle$, its length being 3.

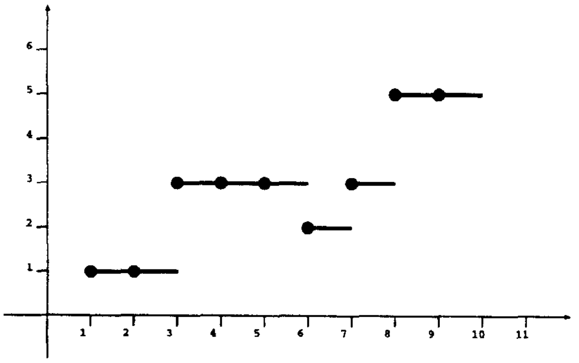
Starting from the definition and the example, one easily understands that plateaus are the *longest* non-empty intervals $\langle i:j\rangle$ included in $\langle 1:n\rangle$ such that (2) holds. One could even have understood this without noticing that the definition is incomplete.

**On the usefulness of remarks, or, better, of a "reasoned" presentation of definitions.** All this shows that it is difficult to correctly define a concept in order to explain it to somebody else. In a sense, *writing the definition is already a programming act.* A definition is thus always the product of a more or less explicit reasoning process. So if one wants to facilitate the correct understanding of a definition, one could point out delicate issues in remarks, or, better, one could make explicit this reasoning process.

For the plateau problem, one might want to point out that the notion of plateau only makes sense with respect to a sequence $S$, that a plateau of $S$ is always a non-empty interval, and that the set of plateaus of $S$ partitions the interval $\langle 1:n\rangle$, where $n$ is the number of elements of $S$.

To do even better, one might show how the given definition was reached from a "reasonable" intuition of the concept of plateau of a sequence. This could go as follows.

Let $S = (s_1, s_2, \ldots, s_n)$ be a finite non-empty sequence of integers. Let us draw a coordinate system, and mark the points at coordinates $(i, s_i)$, for $1 \le i \le n$. Let us now draw, from each of these points, a horizontal segment of unit length. Some of these segments can be merged, giving rise to disjoint segments of integer length, called *plateaus* of the sequence:



The plateaus of the sequence $(1,1,3,3,3,2,3,5,5)$.

[5]A few examples: as a student, the first author had to use the PL/I language and found it completely impossible to understand the manuals except from the examples. As teachers in program proving, both authors always give examples to support program specifications. In a few cases, we eventually discovered errors in our specifications, although the students had understood them perfectly well.

The objective is to write a program for computing the number of plateaus of a sequence and the maximum of their lengths. It is obvious that to each plateau corresponds an interval verifying conditions (1) to (3) [included here as above], and vice-versa. Moreover, the length of a plateau is the number of elements of such an interval. This leads us to the following redefinition of the plateau concept, for our problem: [here follows the definition above].

This presentation clearly allows one not only to understand the definition more quickly, but also to "verify" it according to one's own criteria. We even claim that such an intuitive presentation is self-sufficient and even preferable to the *definition*, as the latter *is only a property of plateaus that one might discover by oneself*.

A final remark: in practice, it is not always useful, nor possible (for "financial" reasons), to discuss all introduced "simple" concepts in this much detail. An acceptable compromise between the quality of the presentation and the time invested to its tuning must be found. Only experience shows where to situate this compromise. The most important thing is to understand that the objective is always the same: to capture the posed problem as well as possible, as it is.

**The case of more complicated or "new" concepts.** Sometimes, the solving of a programming problem requires the invention of relatively original concepts that one cannot pretend having known before tackling the problem. They can thus not be imagined from nothing, but only constructed in small steps after comparing the problem to what is already known. The role of a definition is then to anchor some ideas for further investigation: one must be able to deduce many other properties from it, establishing thus the usefulness of the concept for solving the problem at hand. The concept thus offers economies of thought and reveals ways of solving the problem. The choice of a definition is guided by an intuition, i.e., the impression of having perceived an analogy with something already known. This definition is, in general, not the good one, because it may later turn out that not all the "desirable" properties can be derived from it, so that it does not play an efficient role in the problem solving process. The definition then has to be modified, in the light of these first conclusions, and so on, until the "good" concept has been obtained, namely the one that holds the key to the solution, or part of it. At the end of this process, whose essential steps must be reconstituted by all "clients" of the concept, the latter is known beyond the finally adopted definition. It is known by numerous properties linking it to other concepts. It has become "intuitive and well-known" and its definition is only one of its properties, among many others.

**Remark.** The distinction made here between "simple, implicitly known concepts" and "complex, new concepts" is of course too crude. They are only the extremes between which intermediates can be found, corresponding to a gradation of the effort to be done in order to construct a theory.

**2.5.3 Only Representations of Concepts Can Be Defined, not the Concepts Themselves.** To justify even more that the definitions introduced during the analysis of a problem are not the ultimate reference point for judging the value of a "solution," but only (imperfect) means of communication or "transient" starting points that can be (or actually should be) forgotten once the concept is well-understood, it is interesting to remark that a definition never really defines a concept, but only a certain representation thereof. This remark ruins, by itself, the "absolute" character of definitions by showing why they can be "wrong": whereas a concept cannot be something else than itself, its representations can be incorrect, i.e., can fail to respect the (implicit or explicit) rules according to which they are supposed to represent the concept.

In our opinion, a concept worthy of this name must have a real and original identity that makes it indivisible, distinct from every more or less complex combination of "simpler" concepts. A concept is an atom of thought. Therefore, an interesting concept will always escape any particular definition, because one can define, from given concepts, only combinations thereof, i.e., nothing really new.

All this is particularly clear for "old," universally known concepts: for instance, whatever effort is undertaken to define natural numbers must be arbitrary. A natural number is what it is and cannot be reduced to anything else. Any definition thereof rests on representation conventions that had better be fixed very explicitly if one wants to wind up with a satisfactory definition.

But all this is still true for "new," problem-tailored concepts. For instance, the concept of plateau of a sequence introduced above by a definition corresponds, in fact, to an intuitive concept that is very precise, but impossible to communicate as it is. This is why that definition of a plateau only defines a representation of this concept, namely as an interval of integers. Other representation choices would have led to different definitions. For example, one could have decided to represent the intuitive notion of plateau by couples of integers instead, as follows:

**Definition 2.3.** Let $S = (s_1, s_2, \ldots, s_n)$ be a finite non-empty sequence of integers. A *plateau* of $S$ is a couple $(i, j)$ such that:

1. $1 \leq i \leq j \leq n$
2. $s_i = s_{i+1} = \cdots = s_j$
3. $i = 1$ or ($i > 1$ and $s_{i-1} \neq s_i$)
4. $j = n$ or ($j < n$ and $s_j \neq s_{j+1}$)

This definition seems (to us) less good than the previous one, because it handles differently the plateaus at the extremities of the sequence. This is due to the fact that one cannot talk about the inclusion of a couple in another one. The reasoning to be made for constructing and understanding this second definition is thus slightly more tedious and error-prone. The plateau concept is thus more easily assimilated from the first definition. In any case, in both approaches one has only defined a representation of the intuitive concept of plateau, which is the only really important thing to understand. One should not believe however that an axiomatic definition (e.g., an abstract datatype definition) would be immune from this. Consider, for example, the following definition:

**Definition 2.4.** Let $S = (s_1, s_2, \ldots, s_n)$ be a finite non-empty sequence of integers. A *plateau structure* on $S$ is defined by choosing a set $P$ and two functions $lb$, $rb$: $P \rightarrow \mathbb{N}$ such that the following conditions hold:

1. $\forall p \in P$: $1 \leq lb(p) \leq rb(p) \leq n$
2. $\forall p \in P$, $\forall i$: $lb(p) \leq i \leq rb(p)$: $s_i = s_{lb(p)}$
3. $\forall i, j$: ($1 \leq i \leq j \leq n$ and $s_i = s_{i+1} = \cdots = s_j$), $\exists ! p \in P$: $lb(p) \leq i$ and $j \leq rb(p)$

But this definition certainly says no more than the previous two about the essentials of the plateau concept. Refusing to say what plateaus are "made of" (be it intervals, couples, or beer bottles) is not sufficient for guaranteeing that the reader immediately understands the concept.

A concept is abstract not because it was introduced in a certain way, but because it has acquired an importance and identity in our thoughts. Therefore, the important issue is not to try and discover the good way of defining things, but to choose the adequate concepts, namely those that help us because we understand them the way they are.

**2.5.4 On the Usage of Executable Definitions.** Nevertheless, not all ways of defining a concept are equally good. The "style" of a good definition should be adapted to the problem at hand in order to allow one both to validate the definition and to derive useful properties of the concept. Rather than giving rules for writing definitions, we will criticize a commonly given one, i.e., that a good definition ought to be written in a non-executable language [Hayes and Jones, 1989]. To illustrate our point, we choose the very text formatting problem that was selected to show the virtues of declarative (and formal) specifications, and that was already discussed so much in the literature (see [Meyer, 1985] for an overview).

Most people involved with this problem sought to specify it well, because, according to them, the correctness of a program can only be judged against its specification. According to us, the correctness of a program corresponds above all to an objective fact, independently of the way the problem is posed. Indeed, posing a problem means first of all admitting that there *is* a problem, and, next, trying to understand it in order to be, finally, able to solve it.

Posing the text formatting problem requires first of all the definition of the input and output texts. This can only be done here after making some hypotheses on the "environment" of the user. If we had to solve this problem for a real environment rather than for the sake of this article, then we could not make any such hypotheses but should learn about the environment of the user so as to replace these hypotheses by facts, which would be substantially more complicated than those used here. We thus suppose the user "sees" texts as sequences of lines (corresponding, in general, to lines on the screen or on paper), each line being a sequence of characters. This leads to the following definition of the input text:

**Definition 2.5.** A *word* is a finite, non-empty sequence of non-blank characters.[6] A *line* is a finite, possibly empty sequence of characters and blanks. Every line $l$ can thus be uniquely decomposed as follows: $b_0\ w_1\ b_1\ w_2\ b_2 \ldots w_n\ b_n$, where $n \geq 0$, the $w_i$ are words, and the $b_i$ are sequences of blanks that are non-empty ex-

[6] We consider an alphabet with a single blank character, denoted by ⊔, and no layout characters, such as for tabulation and end-of-line.

cept possibly for $b_0$ and $b_n$. The sequence of words $(w_1, w_2, \ldots, w_n)$ is the *sequence of words represented by* $l$, which we denote by $l$ *repr* $(w_1, w_2, \ldots, w_n)$. A *text* is a finite, possibly empty sequence of lines. Let $t = (l_1, l_2, \ldots, l_p)$ be the input text. The *sequence of words represented by t* is the sequence of words $S$ such that $S = S_1 \bullet S_2 \bullet \cdots \bullet S_p$, where the $S_i$ are the sequences of words represented by the $l_i$ (and $\bullet$ denotes sequence concatenation). We denote this by $t$ *repr* $S$. Two texts are *equivalent* if they represent the same sequence of words.

Now we must define the output text corresponding to a given input text. Therefore, we first have to capture this concept from an intuitive point of view. So, to what does it correspond? The answer is: to the result of applying an algorithm! The best way to understand this concept is to imagine a human typist having a listing of the input text and a terminal where every line has a length of *maxpos* characters. The job of the typist is to type the input text into the terminal by filling every line as much as possible, without trespassing the limit of the screen nor breaking words. This clearly amounts to the application of an algorithm whose execution uniquely determines the output text. Therefore, if one absolutely wants to "mathematically" define the output text in terms of the input text (i.e., if the previous explanations are deemed insufficient), then the best one can do is to give a definition paraphrasing as closely as possible the typist's algorithm, because such a definition has the best chances of being correct and comprehensible. We thus propose the following definition:

**Definition 2.6.** Let $S = (w_1, w_2, \ldots, w_n)$ be a finite, possibly empty sequence of words that are each at most *maxpos* characters long. The *compact representation* of $S$, denoted by *compact*$(S)$, is the text defined as follows:

1. if $S = (\,)$, then *compact*$(S) = (\,)$;
2. else (i.e., if $n \geq 1$), let $i$ be the largest integer such that $1 \leq i \leq n$ and the line $w_1 \sqcup w_2 \cdots \sqcup w_i$ has no more than *maxpos* characters, and let $l = w_1 \sqcup w_2 \cdots \sqcup w_i$ and $S' = (w_{i+1}, \ldots, w_n)$, so that *compact*$(S) = (l) \bullet$ *compact*$(S')$.

Given an input text $t$ and the sequence of words $S$ represented by $t$, the *output text corresponding to t* is defined if and only if no word in $S$ is longer than *maxpos* characters. It is then equal to *compact*$(S)$.

Although it is not expressed in a strictly formal language and especially not in a programming language, the definition of *compact*$(S)$ can be viewed as executable since it strongly suggests a way of computing *compact*$(S)$. However, it can also be argued that the definition is declarative, because it does not prescribe an order for the computation. (The value of *compact*$(S)$ can be computed either top-down or bottom-up.) Nevertheless, the definition can be encoded relatively straightforwardly in any programming language embodying recursion. More importantly, *the definition expresses the effect of an algorithm executed by hand*, which constitutes our fundamental intuition of the problem. Hence the definition of *compact*$(S)$ is not really *a definition* of the problem but rather *an essential property* that we can, on the one hand, validate against our intuition, and, on the other hand, use to construct a correct program to solve the problem.[7]

Note that our definition of *compact*$(S)$ contains an over-specification, according to Meyer (1985), because we constrain the lines to be filled as much as possible in top-down order, rather than in non-determinate order.[8] We do not see the utility of preferring a non-deterministic specification in this case.

In conclusion, we agree on the value of declarative specifications if "declarative" means "as natural and close to the intuition as possible." But we dispute the idea that such specifications necessarily are "non-executable" or "non-deterministic." In our view, the specification in (Meyer, 1985) is unnatural, i.e., difficult to validate and difficult to use, because too much emphasis is put on non-executability and non-determinism, at the price of losing intelligibility.

## 2.6 The "General Form" of Specifications

We now try to capture the "general form" of specifications,[9] without however giving systematic rules for writing "good" specifications, as such is too problem-specific.

[7]An explicit proof of the correctness of a Pascal program solving the text processing problem, based on the definition of *compact*$(S)$, has been given by the first author in (Le Charlier, 1985).

[8]Notice that, although this constraint is a clear consequence of our basic intuition, the definition of *compact*$(S)$ does not impose a unique order of computation to the program.

[9]According to a suggestion by one of the anonymous reviewers, we can reexpress this in terms of the concepts of problem and solution, as follows: first, the specification of a program cites the (name of the) problem that it helps to solve; second, it provides the interpretation rules (or whatever other necessary representation conventions) allowing the user to solve the problem thanks to the results of the program.

The specification of a program should always have two parts that play very distinct roles:

1. a statement indicating the purpose of the program, i.e., the information that can be drawn from the results of its execution;
2. a list of representation conventions that are to be satisfied for using the program correctly and for interpreting its results correctly.

Statement (1) must always be very simple because the information produced by a program (after interpretation of its results) must have a simple meaning to the user. Without it, she would be unable to use the program to her advantage. The role of the "theory" of the problem is to make sure that this meaning exists and that it can be clearly and simply formulated. The list (2) must also be sufficiently simple to understand for the purpose of the program not to be completely annulled by the difficulty of its usage and the difficulty of interpretation of its results. This is not always easy to achieve due to the formal character of programming languages. It is thus sometimes necessary to construct another theory before being able to simply state the representation conventions.

We now state what the specifications of the three problems in Section 2.1 should contain.

**Example: The Belgian National Lottery.** The specification reduces to the indication of how to start the program and to the statement that it results in displaying the next draw of the Belgian national lottery. (It is practically useless to state the exact format of the produced character string and the rules for decoding this information, because everybody immediately understands how to interpret the message when it appears.)

**Example: A payroll program.** The accountant user of the payroll program must know the necessary information as well as the rules of its representation by the input data. She must be able to verify the correctness of these data. She also must know enough about the rules of representation of the results in order to be able to finish the payroll task (this is actually the responsibility of a bank, nowadays). The specification thus reduces to the indication of how to start the program and to the statement that, from correct input data, the program produces correct results according to the used representation rules.

**Example: A search sub-program.** Depending on the desired generality, the programming language used, and the general context of the problem at hand, there is a tremendous variety of possible specifications for a program performing a search in an array. A satisfactory specification, in some cases, could be the following:

**Specification 2.1**

The procedure *search* is a Pascal procedure declared as follows:

*function search*$(x$: *integer*$)$: *boolean*

Its declaration must figure within the scopes of the declarations of an integer constant $n$ (such that $n \geq 1$) and an array $a$ of type *array*$[1..n]$ of *integer*, which also is in the scope of the former. When calling the procedure, the elements of array $a$ must be in non-decreasing order. Let $v$ be the actual value of the formal parameter $x$. If at least one of the elements of $a$ is equal to $v$, then the call returns the value *true*, otherwise it returns *false*. (The contents of $a$ will be unchanged.)

The bulk of this specification is dedicated to the statement of the representation conventions and to technical details. These details are tedious but unavoidable because the used programming language is a formalism. They do not, however, render the specification unusable because the problem of knowing where to put the various declarations and how to write them can be solved separately as well as once and for all. When reasoning about it in the future, it suffices to remember how to call the procedure, that it answers the question "does $v$ belong to $a$?," and that the answer is given as a boolean value.

However, it is important to note that the introduction of general representation conventions that are specific to a particular problem (i.e., that are chosen for an application and used for the specifications of all the sub-programs of this application) can contribute to making much more manageable the amount of representation details specific to each specification.

## 2.7 Requirements Specifications and the Theory of the Problem (Are the Same Thing)

The process of elaborating requirements specifications is nowadays considered by many computer scientists as the most crucial stage of software development. Requirements engineering is thus emerging as a new and major branch of the software engineering discipline. It is primarily concerned with the identification of the user's needs, i.e., the so-called requirements elicitation process. As soon as the user's requirements are explicitly stated, they can (and must) be checked with respect to consistency and completeness. In fact, this is what we call "elaboration of the theory of the problem." Thus, requirements specifications are *not* specifications (in our sense), but rather an exposition of the very theory making it possible to specify the software system.

Formal specification languages are advocated by many researchers as the distinguished methodological tool for requirements engineers, because they allow them to make the user's informal statement precise, to check the requirements specification for consistency and completeness, and to ease the discussion with the user by means of prototyping, to name but a few advantages. In our opinion, the mechanical treatment of (formal translations of) the user's requirements can indeed possibly provide information that could not be easily inferred by hand. However, the formal translation process is completely similar to the writing of a program in that it necessitates giving precise specifications (in our sense) to most symbols and constructs of the formal text, in order to ensure that the formalization captures exactly what the user meant. Thus, the writing of (so-called) formal requirements specifications presupposes the existence of an already fully understood theory of the problem, in our sense. Finally, as seen in Section 2.5, even the elaboration of the theory of the problem may benefit from the use of specifications in our sense, in order to make explicit the rationales underlying the concepts introduced by means of definitions.

## 3. SPECIFICATIONS NECESSARILY ARE INFORMAL

### 3.1 Why Can't There Be Any Formal Specifications?

A "formal specification" is a statement in a formal specification language.[10] Such a statement is unintelligible "by itself," primarily because the concepts of the problem are not primitive concepts of the used formal language. Therefore, a formula can only be "understood" as a representation of an intuitive statement, according to explicitly given conventions. These conventions are in general that the formula is true, in the chosen interpretation of the language, if and only if the intuitive statement is true. The enunciation of such conventions is precisely what we call a specification, in the sense that we discussed in Section 2, although not the specification of a program but rather of a formula. Its role is to give a meaning and thus a purpose to something (the formula in this case) that would otherwise not have one. Whether a formula is true or false is of no interest whatsoever if this is the only thing we know about it. In general thus, a specification is necessary each time one wants to represent a *known* property or concept by a text written in an artificial language. This also shows that any "formal specification" of a (formal) program is much closer to the program itself than to a specification in our sense. A noticeable difference may be that it is not "executable" because it is written in a "non-executable" language. In our opinion, it is not important whether the chosen language is executable or not, but whether it allows us to say in the most direct way what the purpose of the program is. Such a condition cannot be fulfilled by any formal language, given the extremely low expressiveness of such languages. A formal language is always almost as bad as a programming language for communicating the purpose of a program. In other words: providing a formal specification of a program amounts almost to considering that the text of the program (or of another program) allows one to understand its purpose.

Some would now charge that our thesis is mainly definitional, and that we redefine the concept of specification in a way that rules out formality. However, we do not think that we actually redefine this concept, since it is generally agreed that the specification of a program is the statement with respect to which the correctness of the program becomes meaningful. Indeed, the crux of the question is not whether we have redefined the concept of specification in a way that rules out formality, but whether our view of the concept correctly captures the notion and makes it useful. Here we believe that our definition puts a better emphasis on the role of this notion and thus makes it more useful. So the possible charge can—and actually should—be reversed: it is the formalists who have *incorrectly* redefined the notion of specification, namely in a way that justifies the need for formal methods!

### 3.2 Seven Frequently Asked Questions about Formal Specifications

**Are informal specifications and formal ones complementary?** Many authors suggest that it is neces-

[10] Note that a statement in the "usual" mathematical language, such as $e^x = 1 + x + x^2/2! + \cdots + x^i/i! + \cdots$, is *not* a formal statement, but an informal one because that language is not predefined, nor syntactically checkable, and, more importantly, because its meaning rests on general human knowledge, not on the (obscure) semantics of a formal specification language. Hence such a statement essentially is part of the folklore.

sary to add an "informal comment" to a program that helps communicate the purpose of the program and that corresponds to our notion of specification. Similarly, many researchers argue that formal specifications ought to be complemented by informal statements [Hoare, 1996; Wordsworth, 1992; Zave and Jackson, 1996]. Nevertheless, such comments are considered insufficient to ensure that the effect of the program has been precisely defined. This corresponds to the frequent opposition of *intuition* and *rigor*, which considers that a fruitful intellectual activity should be driven by intuition (which is comprehensible but vague) so as to produce rigorous results (which are formal but incomprehensible). In our opinion, the correct usage of a program necessitates having understood intuitively and rigorously its purpose. There is no need to distinguish two notions of specification, one comprehensible and vague, the other one precise and unintelligible. If a specification features delicate issues that are likely to be misunderstood, it is only necessary to give more details about them. There is no reason to believe such difficulties are best resolved, in all cases, by using a formal language chosen once and for all.

If one thinks it is not safe to directly and simply explain the purpose of a program, i.e., in the way one understands it oneself, and that one had better define with absolute precision the "effect" of the program, even under the risk of incomprehensibility, by giving the readers "indications" on how to reconstruct a comprehensible specification for themselves, then one is confronted with the following difficulties. *It is almost as difficult to write without errors a formal specification as the program itself, and it is barely easier to "decipher the message," in the opposite direction.* To write a correct formal specification, one has to make an explicit detailed reasoning that is very different from a vague informal comment. In order to convince oneself of having understood the formal specification, another reasoning has to be done, which is extremely tedious if the formal specification is not accompanied by such comments. So, for a couple ⟨formal specification, informal specification⟩ to suitably play its intended role, it would have to be accompanied by a detailed reasoning fixing their representation relationships. However, this is only meaningful if the informal specification has been explicitly and precisely stated. The role of the formal specification and the reasoning is then reduced to lifting the last doubts and ambiguities. But this can be achieved at lower cost by other means, such as the inclusion of significant examples, the provision of the reasoning process leading to the definitions in the specification, etc.

**Are formal specifications a means of dividing the difficulty of programming?** Other people would rather say that the recourse to formal specifications is, if not a panacea, at least a means of division of the difficulty. Indeed, it would allow, on the one hand, the formal and mechanical proof of correctness of programs, and, on the other hand, the intuitive justification that the formal specifications correctly represent the problem to be solved. One could thus give much more confidence to programs, since everything reduces to the problem of validity of the formal specifications, formal correctness being established beyond all doubt.

This viewpoint rests on two forms of exaggerated optimism on formal methods. First, it is in general not significantly easier or safer to prove intuitively the correctness of formal specifications than that of programs. Second, formal proofs of program correctness are almost always infeasible in practice, whatever the available mechanical aid (proof verifier or theorem prover). For example, note that a formal proof of program correctness amounts to proving a formula whose length is at least the sum of the lengths of the formal specification and the program. So what will be the length of the proof?! This also assumes a *complete* formalization of the semantics of the programming language, which is already by itself an almost unrealizable task. If one considers that the time and budget allocated to the verification of program correctness is necessarily limited, it can be easily seen that one had better spend a bit more time justifying intuitively the correctness of the program and carefully choosing test cases, rather than making use of such formal methods.

More pragmatically and without aiming at complete correctness proofs, software tools could be used to check some "desirable" properties of programs. It is not our purpose to discuss the value and usefulness of such tools in this paper, since they are often more related to documentation and organizational issues than to correctness issues. The former issues are extremely important in practice, but their discussion is completely out of the scope of this paper. Nevertheless, we think that such tools can possibly become harmful, because the value of programs could be judged only with respect to the properties that are checkable. Hence we insist that correctness rests on largely unformalizable issues and should thus be addressed by making explicit informal reasonings and by keeping a record thereof.

**Remark.** In spite of the previous argumentation, we do not dismiss current research on automated program verification, provided it is understood as very long term research whose final outcome is still largely unclear. In fact, both authors of this paper *are* doing research related to program verification (Flener and Deville 1993; Flener et al., 1998; Le Charlier, 1994). Existing techniques, such as model-checking or abstract interpretation, can be applied to verify specific properties of program and systems. It is however unclear at the time of writing how this research will affect future practice. Moreover, the authors of this paper do think that their view on informal specifications remains of paramount importance to understand and master systems based on those emerging techniques (Le Charlier and Flener, 1997).

**Is it necessary to formalize specifications to prove their consistency and completeness?** Some people say that formal specifications allow systematic verification of their consistency and completeness. This deserves several remarks.

If it is desirable that a statement be consistent and complete, the precise meaning of these notions always strongly depends on the context of the statement, that is on a lot of things that are known about the subject of the statement before even examining it. If a statement defines a problem that has no solution, it is sometimes judged inconsistent, but, at other times, it is considered a perfectly consistent statement of a problem that just happens to have no solution; similarly for completeness, when the problem has many solutions. Since a formal statement only is, in general, a representation of a non-formal statement, which is the only one to be comprehensible, the consistency and completeness of a formal statement can only receive a precise meaning through this representation relation. As this relation is always chosen ad hoc, it is impossible to satisfactorily define (i.e., in a manner always corresponding to the intuitive concepts) consistency and completeness of formal specifications. Since this relation is thus totally exterior to the used formalism, consistency and completeness cannot be verified mechanically.

However, there is some belief and hope among many computer scientists that the "real world" can be modeled in some canonical way, provided that an adequate formalism is used. Such belief and hope rests on the assumption that such a formalism could reflect the structure of reality. Hence, incompleteness or inconsistency of a description of the world written in this formalism would be interpreted very naturally as incompleteness or inconsistency of our *understanding* of the world. This view is related to Hilbert's program for proving the non-contradiction of mathematics. His intuition was that all mathematics could be embodied in a uniform formal system whose non-contradiction could be proved by elementary arithmetic reasonings. Gödel's incompleteness theorem has definitely ruined this program. Hence, there is no natural formal structure to all of mathematics. A fortiori, there is no formalism allowing one to model the world in a natural way. Thus, consistency and completeness of specifications only are a by-product of the specifier's correct understanding and there is no a priori way to check that her understanding is correct.

**Are formal specifications more concise than informal ones?** A common argument is that formal specifications are more concise than informal ones. However, some people argue to the contrary. Strictly speaking, the raised question is meaningless for specifications in our sense, since they are only the way to link the (formal) program to its (informal) purpose. So the question in fact only applies to requirements specifications, or, in other words, to the theory of the problem.

During the elaboration of this theory, the usage and introduction of mathematical notations is certainly useful, but, in our view, usual mathematics are part of the folklore and hence mathematical notations are part of the natural language. Indeed, mathematical notations mainly are a way to make natural language more concise. Note however that an explanation of the link between these mathematical concepts and the concepts of the problem is generally needed, and this part of the "theory of the problem" necessarily requires using plain natural language. (Thus, it cannot be made concise by means of mathematical notations.)

Finally, what can be done with usual mathematics can be done to some extent within a formal specification language. However, the notations available in such a language are extremely less convenient than the usual mathematical notations, notably because such languages are syntactically checkable and have (or should have) a fixed (and often complicated) semantics. As a consequence, many more explanations are needed to link a formal requirements specification to what it stands for in the real world than to understand the "theory of the problem," in our sense.

**Are formal specifications more pragmatic than informal ones?** Some advocates of formal methods readily agree on the inevitability of informal specifications and informal verification, but they also point out that formal and informal specifications have different purposes and qualities. Indeed, formal specifications, whether executable or not, would offer a means of early feedback from the customer—through execution of the specification (early prototyping) or through demonstration of desired properties—and hence could allow significant cost savings. Otherwise, discrepancies between the specification and the customer's intentions might only be detected when the customer runs (an increment of) the final software. Indeed, one may certainly construct intermediate formal descriptions before constructing the final software, as they can help during the process of elaborating the theory of the problem. But one cannot call such a description a "formal specification" (and writing it is more of a programming activity than a specification activity), as it is not a specification at all (in our sense) and as it is incomprehensible by itself and must thus be explained to the customer (which explanation process provides the very part that is missing in the formalization), be it as a document or as an executable or demonstrable prototype.

**Can formal specifications be automatically generated from informal ones?** Some researchers advocate writing informal specifications in so-called "semi-formal" notation (such as SA/SD) or in some form of "controlled natural language" (in the sense that the vocabulary and grammar are restricted so as to give sentences a "clear" semantics), expecting that they can be (semi-)automatically translated into (executable) formal specifications. The problem with the former approach is that these languages essentially are informal ones (because they do not feature a predefined syntax *and* semantics), and are thus subject to our comments above on the complementarity of informal and formal specification fragments. There are no such things as "semi-formal languages." The problem with the latter approach is that these languages essentially are formal ones, and thus subject to the comments in this entire paper. There are no such things as "informal controlled natural languages." Since the descriptions are thus actually formal, it is only obvious that they can be automatically translated into some other formal languages. And, as formal statements, they cannot possibly be specifications, in our sense. For such specifications (in our sense), there is of course no way that they can be automatically formalized, as the link between the formal concepts and the real-life ones is not formalizable and as one would have to prove that the translation process is equivalent to the mechanisms of human knowledge acquisition.

**Are formal specifications necessary for safety-critical systems?** It is often argued that formal methods are necessary for the design of safety-critical systems, and some standards organizations even start imposing/recommending their usage for such projects. The rationale is that systems satisfying "specifications" in the form of, say, finite-state machines (that are deemed trivially correct after inspection) can be shown, say, to be free of deadlock and lifelock risks. Our objection to this formalist viewpoint is essentially the same as to the pragmatism issue above, because, once again, it is a delusion to believe that there can be "obviously correct formal specifications."

Note that we do support the idea that extra care and rigor are needed in the design of safety-critical systems: it can certainly be worthwhile to check via model-checking whether some hardware component complies with some formalized property. Nevertheless, the elaboration of such formalized properties requires substantial informal reasoning and specifications in our sense. We even believe that, in most cases, making completely explicit the informal reasoning leading to the design of a safety-critical system is more reliable than a formal verification. Of course, the formal verification can bring extra confidence or detect shortcomings in the informal reasoning, but, in our opinion, such benefits have been too much overvalued in the literature on formal methods.

## 4. CONCLUSION: WHY ARE THE ROLE AND NATURE OF SPECIFICATIONS SO OFTEN MISUNDERSTOOD?

We now explain why our notion of specifications is difficult to understand and to admit by many practitioners and theoreticians of computer science. But let us first summarize our viewpoint:

1. A program is useful because its results can help to solve a problem. There is no limit to the class of problems that we can imagine in the "real" world. Therefore, the understanding of the purpose of a program may necessitate the knowledge of notions as distant as desired from programming concepts (or from concepts used in formal specification languages).
2. The specification of a program essentially is the statement of its purpose.

3. A specification should not, nor can it provide all the knowledge necessary to the understanding of the purpose of the program. It must just try to state it in the most satisfactory possible way, that is in the most simple and direct way. That is why a specification is not meant for everybody, but only for those who can understand it.
4. For the specification to be comprehensible by sufficiently many people, it is, in general, necessary to "construct" a theory that can be studied and understood by all. Such a theory cannot be constructed from nothing, but assumes a considerable preliminary knowledge that is partly shared by all the considered people.

Now, there are at least two reasons why our view of specifications is very uncommon nowadays.

First, there is the influence of the currently dominating ideas on the nature of mathematics. Mathematical theories are supposed to be founded on formal axiomatized theories. This means that every intuitive statement of the theory is supposed to be only an "abbreviation" of a formal statement that is itself mechanically deducible from the axioms. From there to infer that every interesting result of a theory can be discovered relatively quickly as soon as the axioms of its theory are known is only a small step. And this is the "step" made, consciously or not, when asserting that the specification of a program should, above all, define with absolute precision the effect of the executions of this program. Indeed, it is clear that from the input/output relation determined by the executions of a program, one can theoretically deduce all other interesting properties of this program. Therefrom, some conclude that a specification reduces to such a definition, assuming that every reader is sufficiently intelligent to derive from it all other "interesting" properties of the program. (This means the reader is assumed to be omniscient, because if a program outputs the string "380,000", she would, for instance, have to derive from this observation that one of the properties of the program is to give the distance between the Earth and the Moon, expressed as a decimal amount of kilometers.)

Therefore, the idea that the specification of a program must be and can only be the definition of an input/output relation is a simple transposition of the idea that there is nothing more in a mathematical theory than in its axioms. But, in order to understand the exact role of specifications, one should realize that, to the contrary, there is infinitely more than that in an intuitive theory: every new concept, notation, or result adds value to it that is not at all contained in the statement of its axioms. The intuitive statement of an important theorem certainly is not a mechanical consequence of the axioms of a formal system, no more than the assertion of the "truth" equivalence between this statement and a formula. And this even holds for statements of the form "that formula is a theorem," because the meaning of the notions of formula and theorem is not derivable from the mechanical rules of the formal system.

In conclusion, *a correct understanding of the notion of specification necessitates, in our opinion, a return to a more intuitive and "transcendent" perception of mathematics.*

Second, there is the opinion according to which the mastery of the programming problem can only be achieved by recourse to effective and automatable methods. It seems (sadly) evident that few people are ready to admit that the mastery of programming will always depend, above all, on the competence of the involved people. The manager wants effective criteria evaluating the quality of the work done by the programmers. The programmer expects the "theoreticians" to provide rules that can be followed blindly. Nobody wants to admit that the best way to realize whatever task is to do one's best, by trying to stick to utmost intellectual honesty.

If, regarding specifications, we say that the best thing to do is to understand the exact role of this notion so as to be able to "see," in most cases, how to state them best, it will be considered that we have not brought anything interesting to the debate, because we have not given any rule or criterion for writing good specifications or for evaluating them. However, some people say that, as it is better to do something rather than nothing at all, it is better, all things considered, to give rules that are arbitrary but measurable.

For us, it is certain that little progress can be expected in programming as long as the opinion is so widespread that the value of a criterion is determined by its being measurable and computer readable. We think so because this idea can only prolong the illusions and avoid the real problems: thanks to such criteria, the manager can take decisions without having to get involved in the project, and this changes nothing to the quality of the programmers' work, except that they have to adjust themselves so as to respect these rules even when they do not bring any practical help, or, worse, when they complicate the construction of the program.

These remarks apply not only to software project managers, but also to the managers of research funding agencies. Academicians are almost "forced" by them to claim that their formal methods research will increase productivity and competitiveness.

Finally, let us stress once again that formal methods research is not sterile, especially in the long term, because it will allow us to understand better how to design convenient computer languages and systems. However, we do think that our view of informal reasoning and specifications will remain relevant in the long term, since no formal language can possibly refer to real-world concepts as conveniently as natural language can.

ACKNOWLEDGMENTS

The authors are indebted to Prof. Henri Leroy for his spiritual patronage. The central ideas of this paper have been deeply influenced by his teaching and the numerous nightly discussions with the first author. We would also like to acknowledge valuable feedback from Prof. C. A. R. Hoare, Prof. T. Winograd, and Dr. A. Hall, as well as the useful comments of the anonymous reviewers.

## REFERENCES

Balzer, R., A 15 year perspective on automatic programming. *IEEE Trans. on Software Engineering* 11(11):1257-1268 (1985).

Balzer, R., Goldman, N., and Wile, D., Informality in program specifications. *IEEE Trans. on Software Engineering* 4(2):94-102 (1978). Also in C. Rich and R. C. Waters (eds.), *Readings in Artificial Intelligence and Software Engineering*, pp. 223-232. Morgan Kaufmann, 1986.

Bowen, J. P., and Hinchey, M. G., Ten commandments of formal methods. *IEEE Computer* 28(4):56-63 (1995a).

Bowen, J. P., and Hinchey, M. G., Seven more myths of formal methods. *IEEE Software* 12(3):34-41 (1995b).

Craigen, D., Gerhart, S. L., and Ralston, T., Formal methods reality check: Industrial usage. *IEEE Trans. on Software Engineering* 21(2):90-98 (1995).

De Millo, R. A., Lipton, R. J., and Perlis, A. J., Social processes and proofs of theorems and programs. *Comm. of the ACM* 22(5):271-280 (1979). Reactions in *Comm. of the ACM* 22(11):621-630 (1979).

Denning, P. J. (ed.), A debate on teaching computing science. *Comm. of the ACM* 32(12):1397-1414 (1989).

Fetzer, J. H., Program verification: The very idea. *Comm. of the ACM* 31(9):1048-1063 (1988). Reactions in *Comm. of the ACM* 32(3):374-381 (1989).

Flener, P., and Deville, Y., Logic program synthesis from incomplete specifications. *Journal of Symbolic Computation: Special Issue on Automatic Programming* 5(5-6):775-805 (1993).

Flener, P., Lau, K.-K., and Ornaghi, M., On correct program schemas. In N. Fuchs (ed.), *Proc. of LOPSTR'97. LNCS*, Springer-Verlag, 1998.

Flener, P., and Popelínský, L., On the use of inductive reasoning in program synthesis: Prejudice and prospects. In L. Fribourg and F. Turini (eds.), *Proc. of META'94 and LOPSTR'94*, pp. 69-87. *LNCS* 883, Springer-Verlag, 1994.

Fraser, M. D., Kumar, K., and Vaishnavi, V. K., Informal and formal requirements specification languages: Bridging the gap. *IEEE Trans. on Software Engineering* 17(5):454-466 (1991).

Fraser, M. D., Kumar, K., and Vaishnavi, V. K., Strategies for incorporating formal specifications in software development. *Comm. of the ACM* 37(10):74-86 (1994).

Fuchs, N. E., Specifications are (preferably) executable. *Software Engineering Journal* 7:323-334 (1992).

Gerhart, S. L., Craigen, D., and Ralston, T., Experience with formal methods in critical systems. *IEEE Software* 11(1):21-28 (1994).

Gibbs, W. W., Software's chronic crisis. *Scientific American* 271(3):86-95 (1994).

Gravell, A., and Henderson, P., Executing formal specifications need not be harmful. *Software Engineering Journal* 11:104-110 (1996).

Guttag, J., Horning, J., and Wing, J., Some notes on putting formal specifications to productive use. *Science of Computer Programming* 2(1):53-68 (1982).

Hall, A., Seven myths of formal methods. *IEEE Software* 7(5):11-19 (1990).

Hayes, I. J., and Jones, C. B., Specifications are not (necessarily) executable. *Software Engineering Journal* 4(6):330-338 (1989).

Hoare, C. A. R., An overview of some formal methods for program design. *IEEE Computer* 20(9):85-91 (1987).

Hoare, C. A. R., How did software get so reliable without proof? In M.-C. Gaudel and J. Woodcock (eds.), *Proc. of FME'96, Industrial Benefit and Advances in Formal Methods. LNCS* 1051, Springer-Verlag, 1996.

Jackson, M., *Software Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudice*. Addison-Wesley, 1995.

Johnson, W. L., Deriving specifications from requirements. In *Proc. of ICSE'88*. IEEE Computer Science Press, 1988.

Larsen, P. G., Fitzgerald, J., and Brookes, T., Applying formal specification in industry. *IEEE Software* 13(7):48-56 (1996).

Le Charlier, B., *Réflexions sur le problème de la correction des programmes*. Ph.D. Thesis, University of Namur (Belgium), 1985.

Le Charlier, B. (ed.), *Proc. of the First International Static Analysis Symposium. LNCS* 864, Springer-Verlag, 1994.

Le Charlier, B., and Flener, P., On the desirable link between theory and practice in abstract interpretation. In P. Van Hentenryck (ed.), *Proc. of SAS'97*, pp. 379-387. *LNCS*, Springer-Verlag, 1997.

Meyer, B., On formalism in specifications. *IEEE Software* 2(1):6-26 (1985).

Parnas, D. L., Mathematical description and specification of software. In B. Pehrson and I. Simon (eds.), *Proc. of IFIP'94*, pp. 354–359. Elsevier Science, 1994.

Parnas, D. L., and Madey, J., Functional documentation for computer systems engineering. *Science of Computer Programming* 25:41–61 (1995).

Saiedian, H. (ed.), An invitation to formal methods. *IEEE Computer* 29(4):16–30 (1996).

Wing, J. M., A specifier's introduction to formal methods. *IEEE Computer* 7(5):8–24 (1990).

Wordsworth, J. B., *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley, 1992.

Zave, P., and Jackson, M., Four dark corners of requirements engineering. *ACM Trans. on Software Engineering and Methodology* 7(1):1–30 (1997).