SOME PROBLEMS OF SOFTWARE ENGINEERING

WLADYSLAW M. TURSKI

PROF. W.M. TURSKI
INSTITUTE OF INFORMATICS
WARSAW UNIVERSITY
WARSAW, POLAND

Wladyslaw M. Turski
Institute of Informatics
Warsaw University

The phrase "software engineering" was coined just over 13 years ago. It was considered a little provocative by its originators and was chosen explicitly for this reason. The group of coveners of the Garmisch (October 1968) and Rome (October 1969) conference sponsored by the NATO Science Committee deliberately selected a key phrase that contrasted wiht the then prevailing perception of software issues ("software chaos" and "software crisis" being then two phrases very much in vogue). The phrase "software engineering" hinted at law and order in an environment considered hopelessly anarchic, promised some rigour and discipline where there was none, whiffled of industrial approaches in the field where artisanship reigned unchallenged. The phrase "software engineering" evoked - no matter how nebulously - notions of standards, measures of productivity, industry-wide and commonly accepted codes of good practices. Unfortunately, the launching of a phrase seldom solves any real problems and sometimes creates new ones, primarily those of credibility.

Software engineering caught on. Conceived as an intellectual provocation by the mostly academic animators of the Garmisch-Rome conferences, the phrase was eagerly accepted by all sorts of industrial organisations. The younger set (and information processing community expands so quickly, that it constantly seems to consist more of the younger set) is now firmly convinced that software engineering is a well-established industrial discipline, opposed or at least neglected by the academia. (A year ago I witnessed a most revealing scene: at a computer-oriented gathering on the antipodes, a very articulate, but clearly not very well-read, industrial programmer addressed a professor of computing science with a question he obviously thought rethoric: when at long last, would the universities recognise the existence of software engineering, accept the inevitable and start doing something useful in this direction. The object of this tirade was the person who can justly be called the spiritus movens of the 68/69 conferences, one of the authors of the phrase "software engineering". Unlike social ones, this technological revolution turned against its fathers.

The software crisis of the 60's was very real. The quality of most computer programs was very poor, their documentation - at the very best - sketchy, the reusability of software components - practically unheard-of, software systems in use were growing patchier and patchier, software projects were notoriously behind schedule and above budget. The shortage of skilled programmers was crippling, good people were hard to get and once gotten would very often be soon bribed away. Programming languages were just about the only facet of programming sufficiently well-understood to have a comprehensive and consistent theory (even in this case the theory covered only part of the problem, i.e. syntax, leaving out the perhaps more important subject of programming language, which is semantics).

Ideally, software engineering should have cured all these and similar ills. Unfortunately these phenomena - very real ones - are merely symptoms of much deeper troubles and since most efforts directed at remedying the symptoms do not cure the sickness that causes them, even if taken jointly, all prescriptions for ameliorating particular aspects of the software crisis do not seem to appreciably improve the state of affairs. Even though the expression "software crisis" is by now nearly forgotten, almost all the complaints made fifteen years ago could be repeated today, perhaps more strongly. There is, however, one notable and very important exception: we have learned how to write correct programs given precise specifications.

I am fully aware that the ability to make correct programs given precise specifications is but a small consolation for someone faced with the full scope of issues involved in providing the software part of a computerised system, nevertheless I am going to spend some time analysing this achievement. I am going to do so for several reasons:

(i) because it shows what it takes to solve a problem in the area of software engineering,

(ii) because it bares some fundamental deficiencies of the common sense approach to software problems,

(iii) because it demonstrates the intellectual techniques involved in solving a methodological problem if software construction,

(iv) because it is the only part of software engineering that can be completely discussed in scientific terms.

Let us first consider why instead of a simpler expression "correct program" we are using a longer (and, admittedly, clumsier) one: "program correct with respect to its specification".

The only rational interpretation of the shorter term- "correct program" could relate to the internal correctness of a piece of code, i.e. to its syntactic correctness. For a reasonable grammer the question of syntactic correctness can be solved quite mechanically: every acceptable compiler does it all the time. On the other hand, if the syntactic correctness of a program cannot be resolved mechanically, we are dealing with a programming language too ambiguous for any semantic interpretation, with a text too vague to be considered meaningful. (A more puritan viev would be to state that syntactic unresolvability precludes semantic modelling and thus such texts are meaningless.) Since it is safe to assume that we are willing to restrict our considerations to programs that are guaranteed to be meaningful, i.e. to such programs that are guaranteed to have a nontrivial semantic model, we assume that programs we are considering are guaranteed to be syntactically correct.

As soon as we decide that we wish the term "correctness" to express more than just syntactic correctness, we must look for an external frame of reference. (Nothing suprising in it: in a scientific sense, correctness, just as truthfullness - in a slight departure from common usage - is always a dyadic relation that holds, or does not, between two entities rather than being a property enjoyed, or not, by an entity taken in isolation.) For a meaningful, purposful, useful program, the frame of reference that seems most natural for establishing the program's correctness is its specification, i.e., a statement of the program's purpose.

Thus we consider a program and a specification. For some such pairs we are entitled to state that program P is correct with respect to the specification S. The verifiaction of whether a particular pair (P,S) entitles us to make this statement depends, of course, on a great many additional considerations. (Incididentally, I am afraid that some readers are beginning to suspect that I am overly pedantic, that I indulge in an academic nitpicking; let me hasten to assure them that me concern is very practical: it is precisely because I am interested in useful programs that I am rather particular about expressing my objectives quite clearly and opt for writing a watertight warranty that the program will indeed satisfy my goals as expressed by the specifications.)

We leave aside, for the moment, the objections that one seldom is interested in actually verifying if an arbitrary program P is correct with respect to specification S, that our main interest is in producing programs that enjoy this property vis a vis given specification. After all, unless we have a better procedure, a thorough quality assurance test is a purchaser's best friend!

Perhaps the simplest form a program specification can take is a pair of statements, **IN** and **OUT**, each of which can be either true or false, depending on the environment in which it is to be evaluated. (Some sentences, tautologies, are true (or false) regardless of an environment; if we fancy it, we may introduce the statement **TRUE** which - by definition - is true in every environment, and the statement **FALSE** - false in every environment, these two statements are suprisingly useful!) Using an **IN/OUT** pair we express the following request: we want a program P such that if its execution starts in an environment in which **IN** is true then after its execution is completed we shall get an environment in which **OUT** will be true. This request can be written as a

formula:

(IN) P (OUT) (x)

in which P is an unknown, or desired, program. Thus this formula may be considered as an equation that defines program P.

For example,

(integers x, y are defined) P (integers x,x,m are defined and m = max(x,y))

is a specification for a (small) program P that culls the maximum of two given integers.

Now, how do we proceed to verify that a given program P is correct with respect to its IN/OUT specification? There are several ways of doing it, depending on the particular fashion in which semantics of the programming language employed for coding P is formulated. If we are to proceed at all, however, the semantics of this language should be expressed in such a way as to permit calculations of the environment transformations effected by the execution of the language instructions. Knowing how the execution of each instruction transforms its inherited environment into the environment for its successor, we can ascertain if the execution of P, starting from its first executable instruction initiated in the environment satisfying **IN**, will lead, transformation by transformation, to an environment satisfying **OUT**. Thus we can establish the correctness of P with respect to **IN/OUT** by proving the conjecture that arises when the text of program P is substituted in (x). Note that the ability to carry out this proof depends on the ability of a rigorous definition of the programming semantics.

For example, with IN and OUT as before, and

```
P : if x>=y --> m:=x
      y>=x --> m:=y fi
```

we get the conjecture

```
(integers x,y are defined)
 if x>=y --> m:=x
    y>=x --> m:=y fi
(integers x,y,m are defined and m = max(x,y))
```

There are two varieties of thus understood correctness: partial and total. The distinction relates to the fact that some programs are known to contain endless loops, or - what is more realistic - to contain instructions which initiated in some environments may loop forever. Partial correctness amounts to saying: it is guaranteed that provided the execution of P comes to a normal end the conjecture (x) holds. Total correctness amounts to a stronger: it is guaranteed that the execution of program P will come to a normal end and that conjecture (x) holds.

The investigations into the extent of the notion of program correctness led to many useful programming techniques. Methods of composing programs in such a way that their correctness with respect to given specifications would be guaranteed by virtue of construction steps taken were developed and made quite practical. The key to the success of these developments was the appreciation of the calculability of environment transformations effected by the execution of programming language instructions with well-defined senantics. This in turn led to a considerable effort in formulating calculable semantics of programming languages, and - in due course - to certain preferences in programming languages themselves.

The intuitive approach to programming language design (wouldn't it be nice if we had such and such feature in our language) was replaced by a more somber attitude: let's have in our language only such constructs which have calculable semantics, and preferably select those whose definitions make semantic calculations easy.

In recent years, many techniques based on calculable semantics and on the principle of provable program correctness with respect to its specification emerged and found practical applicaton. Even if the practiced version of a programming technique is not explicitly calculational (structured programming, stepwise refinement, Jackson method etc.), their origin is unmistakable and their soundness depends on the firmly established mathematical theory of program correctness.

It is often said that the formal methods of program verification and/or program derivation from specifications are applicible to small problems only, or less kindly spoken, toy problems. Two justifications are put forward in support of this thesis. The first one points out that the volume of formal manipulations needed to verify a program is usually an order of magnitude larger than the volume of the program text itself, which makes this approach impractical for large problems. The second one questions the basic premise of the method - the availability of precise specifications. Both objections are well-founded; the second one is however much more serious and will be dealt with somewhat later, in a broader context.

As far as the length of the verification proofs is concerned we should in all fairness observe that the verification of an existing program against an existing specification is a relatively infrequent event. A much more realistic approach is to use the calculable semantics for deriving the program.

The length of formal manipulations involved is still quite impressive, but in this case the effort spent on "formal manipulations" should not be considered as an addition to the cost of program development. If a programmer develops a habit of formally deriving programs from specifications, then all his activities related to program construction, indeed, the whole problem-solving process is carried out by these formal manipulations. Starting with the necessary problem analysis and derivation of auxiliary facts, through structutal analysis, decomposition and linguistic interpretation (stepwise refinement), and ending with final expansion (coding) - all these steps, which one way or another must be present in program derivation, are combined into a formal derivation of a program. Seen from this point of view the length of the derivation is a measure of the effort needed to properly construct the program. As usual, the derivation may be more or less detailed, some people learn to perform in their heads longer transformations than others, but the fact remains: formal derivation of programs is not any longer than an informal one. It is, however, more explicit, provides a better documentation and is a whole lot less vulnerable to a chance mistake or oversight. In a sense, it is a pity that the published examples of formal derivations of programs - for dydactic purposes - refer to very simple problems only: because it is so easy to derive the specified programs in one's head, an explicit formal protocol of the derivation seems too long and, perhaps, unnecessary.

The relationship between a specification and a program is not a function: given a specification there may be a great many different programs that satisfy it, i.e. are correct with respect to this particular specification. If the specification is too vague, i.e. if it does not capture all important requirements, a correctly constructed program may turn out to be not quite satisfactory. This raises a very

controversial issue of the extent of programmers' responsibility: does it cover a verification of the specifications? And if so, what is the frame of reference to be used in such a verification? With this problem, however, we are leaving that part of software engineering which could be conveniently called programming methodology, the only part in which solid progress over the last decade can be reported.

Contrary to a popular belief, the completeness of specifications - at least the mathematical completeness - would not necessarily be an unconditional blessing. First of all, it would be very difficult to achieve, secondly, it would almost invariable amount to overspecification in terms that matter for the ultimate use of the program.

Deriving a program to meet a given specification, a programmer is free to use his judgement, rely on his expertise, draw upon available knowledge and recources, make decisions in all issues that are left unspecified. For example, a specification may read as follows:

IN: A1,...,AN is a defined sequence of integers.

OUT: (i) B1,...,BN is a sequence of integers and
(ii) B1,...,BN is a defined permutation of A1,...AN and
(iii) i<=j ==> Bi =Bj for all 1<=i,j>=N

This is, of course, a specification for a sorting program. The choice of the sorting algorithm is left unspecified, the programmer may explore this freedom as he wishes - provided the program he produces satisfies the given specification. If the specification was a bit "more complete" and asked, for instance, that the cost of the program execution should not exceed kNlogN, a large class of algorithms would be

excluded; similarly, if it was specified that the programs should not use more than 10% extra memory on top of the N cells needed for the vector A. Observe, however, that in order to express such additional constraints on the program, the specification must be formulated in a language much richer than that needed for the initial one: the extended specification must encompass notions quite alien to the problem of sorting. (Incidently, the IN/OUT style of specifications does not lend itself very naturally to such additional specifications, but this is a relatively minor point.)

In aiming for completeness of specifications great care must be excercised that their consistency be preserved.

The inconsistency of specification is much more harmful than its incompleteness. An incomlete specification can be satisfied by many different programs, the only danger being that the one actually derived would not meet some unspecified requirements (while being in full accord with the specified ones!) An inconsistent specification cannot be met by any program! (In our example, it suffices to extend the given specification by the request that the cost of executing the program be less than kN, for fixed k and N, to make the thus extended specification inconsistent.)

Thus a modicum of incompleteness of the specification is harmless (and in practice unavoidable), whereas the inconsistency must be categorically avoided.

The ability to produce correct programs given consistent specifications, the ablility gained through caclulable formalisation of semantics of programming languages, has caused a marked shift of research interests away from issues of programming languages, towards the issues of specification.

There are several ways in which this relatively new research topic is likely to produce results important for practical work. (In some of these directions considerable progress has already been made, and practically significant results and techniques are available.)

The directions closest to the traditional programming activity is that of formalisation of program-objects specification (such as data types, modules, monitors etc.). In fact, the methods of specifying these objects are so closely related to programming techniques, that frequently they are considered simply as parts of programming methodology. Yet, it is worthwile to observe that the same techniques may be applied to specification of objects not necessarily related to programs.

Consider, for example, the abstract data type specifications. In the briefest possible exposition, one could say that an abstract data type is specified by two sets of formulae:

(i) syntactic rules,
(ii) semantic equations.

The syntactic rules describe the morphology of objects of the type being defined and the syntax of specified operations acting on these objects; the semantic equations express - in calculable fashion - the properties of objects and operations. The publicised examples relate to well-known program objects (stacks, queues, tables, etc.) but the very same technique can be applied to specification of any objects that can be abstractly viewed as many-sorted algebras. In fact, since there is no fundamental reason why this technique could not be applied to, say, data base specification and since (as we are repeatedly told) data

bases can be used to faithfully (well, sufficiently faithfully) represent almost anything of interest in buisness data processing, I see no reason whatsoever why the abstract data type specification techniques could not be applied to specification of software, e.g., for management information systems. (Indeed, some experimental results in this spirit have been reported in research publications.)

Similarly, the techniques used for specifying active software components, such as modules, monitors, and classes, can probably be used for specification of simulation software. In fact, since most of the work in this direction is based on the original contributions of SIMULA 67 - a language initially intended for programming simulation computations - using these techniques for specification of simulation software would in a sense complete a full cycle of development, which is always intellectually pleasing!

Several projects are currently under way trying to combine various specification techniques into specification languages, or more precisely, into software specification languages.

The most important advantage of specification languages based on formal specification techniques would be the availibility of an extensive calculable apparatus enabling the verification of software produced according to the specifications expressed in such languages. Let me once more stress the importance of such an apparatus. Unless the notation of satisfaction is formalized, it cannot be made calculable. And unless we have a calculable means of establishing whether a piece of software satisfies the specification, we are on the very shaky ground of debugging, test-case verification etc., which never leads to foolproof assurances. Recall that it was the introduction of calculable semantics of programming languages that made

possible a satisfactory interpretation of the program correctness problem, and, as a consequence, led to programming methods that guarantee the program correctness.

Another - and in a way no less important - advantage of formalized software specification languages rests in the ease with wich they permit the construction of assorted aids, facilitating the process of programming by performing a host of clerical functions (cross-referencing, indexing etc.), by executing various checks (inconsistency of interfaces, use/define matches etc.) and - in some instances - by simulated "execution" of specified, but not yet fully programmed, software. Especially in large software projects such aids reduce the burden of ancilary functions on programmers and thus increase their productivity by allowing a less diluted concentration on main tasks. Again it should be stressed that the formalisation of the specification language (both of its syntax and semantics) is the crucial factor in determining how extensive a set of aids can be constructed.

Another direction of research on specifications concerns operation on specifications, such as extending a specification by additional requirements and joining two specifications into one. Such operations closely correspond to situations frequently encountered in practice; in fact, as we shall see in a moment, manipulations with specifications are about the most important tool in software engineering. In order to attach meaning to results of formally defined operations on specifications, a suitable formal view of specifications had to be established. This was accomplished by considering specifications as algebraic theories, in which case the theory of categories provided the necessary framework. Burstall's language CLEAR has been explicitly designed for describing specifications as theories and for programming operations on them.

All the so-far discussed research directions on specification issues implicitly assume that the specifications are the ultimate source of inspiration for software. This view is clearly inadequate for practical applications, where the utimate source of inspiration is a need felt by the customer, a need usually poorly articulated and nearly always expressed in terms far removed from program-oriented terminology.

Theoretically, we could argue that the steps necessary to convert such a nebulously formulated need into a specification for a software (system) do not belong to software engineering. Personally, I do not subscribe to this view. First of all, if the pre-specification steps are excluded from the scope of software engineering we shall not have any real control over their quality, and there is very little sense in making an effort to produce high quality software hanging on low quality specifications. Secondly, it is in the link between the customer's needs and specifications that the most troublesome aspects of software engineering have their roots (as we shall see in a moment). Thirdly, an interface between the pre-specification problem analysis and the specification must be established: if we admit that the former is quite informal and the latter - formal, the interface could be extremely akward.

One way of extending the software engineering towards the analysis of customer's needs consists in providing semi-formal tools (such as, e.g. SofTech's SADT) to be applied to the analysis of customer's requests expressed in his language. The use of such tools imposes certain discipline on the formulation of the need, mostly syntactic and structural, thereby establishing syntactic relationships between various structural entities. Roughly speaking, on sucessful application of such various tools we get a

counterpart of syntactic rules of algebraic specification, albeit sometimes expressed in a form much less convenient for subsequent manipulations (in the case of SADT we get a pictorial presentation of syntactic rules). The semantic equations are not so easy to obtain by semi-formal analysis.

One can point out an analogy of a sort between the semi-formal requirements of analysis and flowcharting as a means of program design. It certainly is a step forward with respect to totally informal (unstructured) analysis, but without formalisation of corresponding semantic notions we are still left without means to verifiably establish the correctness of our proceedings. It should be observed that the commercial success of semi-formal techniques in customers' problem analysis provides an empirical proof of the practical recognition of advantages extending software engineering outside the specification/program bracket.

An alternative approach to the analysis of customer's problems has been motivated by considerations of software evolution. It is a well-known fact that software systems in continuous use over extended periods of time evolve. The causes of evolution fall roughly into two categories: internal and external.

Internal causes of software evolution include two major classes:

- corrections
- improvements.

Corrections are such software changes that remove discovered errors, i.e. violations of the satisfaction relationship between an existing specification and an existing program. Theoretically, if the software is correct with respect to

its specification, no corrections are necessary. In practice, they do occur, just as errata are occasionally necessary to texts of mathematical proofs.

Improvements are such software changes that leave the satisfaction relationship between the existing specification and program intact and exploit the freedom left by the specification in order to bring about advantages that cannot be described in the linguistic system employed for the specification. A typical improvement is the replacement of an algorithm be a less complex one or by a "faster" one.

External causes of software evolution may be also classified into two groups:

- related to the change of programming environment,
- related to the change of specification.

A change of programming environment occurs e.g. when the hardware system is extended by a new component or a new hardware facility is added that extends the repertiore of programming means of expression. A more radical change of programming environment is caused by replacement of hardware, in which case (all or some) programming means of expression lose their hardware interpretation. An extreme case of programming environment change is a switch over from one programming language to another, or from one operating system to another, or from one manufacturer's hardware to another's. It should be observed that:

(i) a change in programming enviromnent does not necessarily involve a change in software (e.g. we may ignore an added hardware facility), in which case the situation is roughly similiar to that of internal improvements,

(ii) if the change of programming environment is forcing a change in software, such change is to be effected under the invariance of specification, excepting the somewhat ludicrous cases where the specification contains the explicit request that products of ABC company are to be used.

Thus we have isolated the only kind of software change that is caused by an as it were spontaneous change in specifications. Why should a specification change at all? Well, there are several reasons:

(i) the original specification poorly captured the customer's needs,

(ii) the customer changes his mind,

(iii) the use of the system changed the customer's environment in such a way that his needs have materially changed.

All, who ever participated in a significant software developent project, are well familiar with at least one of these, more often - with all.

In the professional jargon, the activity of changing the software is circumstantially called "software maintenance". It is a particularly absurd choice of terminology to speak of software maintenance when we mean software changes; unfortunately it is being done all the time! The use of this misnomer is also exceptionally harmful from the psychological point of view: it is subconsciously expected that any maintenance should be relatively inexpensive (in comparision with the original investment). Software "maintenance" being anything but inexpensive, the software

modification activities are constantly challenged as wasteful and poorly managed. It would be much better to speak of specification maintenance in the presence of internal causes of software evolution and in the case of a change in programming environment, and to call a spade a spade in case of specification changes.

Two symmetrical errors, commonly committed, contribute to the bad reputation of software evolution:

(i) forgetting to maintain the specification when software is changed for internal reasons,

(ii) specifying software changes without making certain that the resulting specification is consistent.

An often encountered variation of the second error constists in making software changes when the customer's needs have changed, without bothering to modify the specification at all. This practice is supported by the befief that "software models the application", hence if the application changes, the software should change accordingly. In fact, the software is related to an application through the specifiction, and if the verb "to model" is to be used in its technical sense, then software models its specification. Thus if we want to stabilize the software evolution, we must jealously maintain the specification-software relationship, and allow such software changes only which either werifiably preserve this relationship under invariance of the specification, or re-establish this relationship when the specification has been modified in a consistent way.

The relationship between the specification and software being thus promoted to the role of main concern of the programmer, how do we envisage the pragmatically important

link between the customer's needs and the - necessarily formal - specification? In order to achieve a pleasing symmetry and a simple, conceptually unifying treatment of both considered environments (the program environment in which a program models a specification, and the application environment) I suggest that the particular application be considered a model of the specification in the application domain. In this way, formally at least, the relationship between the particular application and the specification is exactly of the same kind as the relationship between specification and software.

The major advantages of such an arrangement come from the obligation to prove that the application satisfies the specification. This means that the application must be presented so precisely that such a proof would be possible. At the same time, it does not mean that the application must be described in programming terms. The choice of the language used for application description is left entirely to the application experts, the only requirements being that it has a calculable semantics, just as the programming language has one.

Just as there may be many programs satisfying any given specification, many applications may fit a given specification. The freedom left by the specification in the application domain is now a measure of how precisely the specification captures the customer's needs. If, with a given specification, one gets too unrestricted application models the specification has to be tightned.

Naturally, in the temporal sequence of events, the specification hardly precedes the application model. In practice it will be somehow abstracted from a description of the application, but the abstraction process (present in all system design methods) is now verifiable.

In addition to the methodological advantages, the proposed arrangement may be used as a framework in which stable evolution of software may be clearly monitored, and thus at least some calamities may be prevented. Indeed, assume that the customer feels a need to modify the software. This need must be expressible as a change in the application model (no other means of articulation is admitted, or to put it bluntly: all other means of articulation of the customer's wishes are simply dangerous and should be disregarded). Thus a modification of the application model is considered as the only possible source of the initiative for software change. We may safely assume that such modification violates the existing relationship between the specification and the application model. (Even if the specification/application relationship is not broken there may be a valid reason to change the specification - it just has been demonstrated that it is too insensitive to application domain modifications!)

The ensuing change of the specification must be effected in a formal system in which the specification is written and the modified specification must be checked for consistency. At this stage some changes may be rejected, others may cause us to think very seriously if it is really worthwile to introduce them (if the resulting modifications of the specification are massive, the work involved in changing the software may be expected to be similiarly extensive.)

When the specification is changed and thus the satisfaction relationship between the specifications and the application domain model is re-established , a crucial decision must be taken: to modify the software (because it does not model the specification any more) or to construct a new software. This unavoidable decision is in the considered arrangement somewhat less arbitrary than in other set-ups because it is prededed by a full-scale formal modification process

performed on the specification.

Most importantly, if the specification changes are expected, the construction of software may be subject to certain rigours, making subsequent specification-directed modifications of software easier. (Modularity, seperation of concerns, splitting a specification into covering "subspecifications".) A particularly promising technique consists in anticipating (at least some) changes in specification, cataloging them and providing - a priori - algorithms for changing the software so as to incorporate any of the catalogues specification changes. .

REFERENCES

(The references listed here are recommended as "further reading" on topics discussed here in this paper.)

Berg, H.K. and Giloi, W.K. (Eds.): The Use of Formal Specifications of Software. Informatik-Fachberichte 36 (1980), Springer-Verlag.

Bjorner, D. (Ed.): Abstract Software Specifications. Lecture Notes in Computer Science 86 (1980), Springer-Verlag.

Floyd, C. und Koptez, H. (Hrsg.): Software Engineering - Entwurf und Spezifikation (1981), Teubner.

Jones, C.B.: Software Development - A Rigorous Approach (1980), Prentice-Hall.

Lehman, M.N.: Programs, life cycles, and the laws of software evolution. Proc. IEEE 68 (1980), 1060.

Turski, W.M.: Software Stability. In Systems Architecture, Proc. 6th ACM European Regional Conf. (1981), London.