A VIEW OF SOFTWARE PROBLEMS

W.M. Turski

Rapporteurs: Ms. D. Bowman
Mr. P. King

Abstract

Various aspects of software problems are addressed, with particular emphasis on correctness and reliability. The view is put forward that correctness is a relation between program text and specifications whereas reliability is the property of an executing program.

Part 1

No self respecting lecturer on computer programming these days can avoid the notion of 'correctness'. I would like to show you the first of three programs and ask 'How correct is this piece of software?'

```
      DO  100  I = 1,N
      DO  100  J = 1,N
100   X (I,J) = (I/J)*(J/I)
```

Is the second any better (more correct)?

```
      DØ  100  I = 1,N
      DØ  100  J = 1,N
100   X (I,J) = (I/J)*(J/I)
```

Or the third?

```
      DIMENSIØN  X (100,100)
      DØ  100  I = 1,100
      DØ  100  J = 1,100
100   X (I,J) = (I/J)*(J/I)
```

These three examples indicate that when you do not know what you are writing the program for, the problem of correctness is meaningless. Incidentally, the examples are some of the correct ways to initialise a unit matrix, remembering the peculiar round-off characteristics of FORTRAN integer division. The product (I/J)*(J/I) will be zero except where I equals J when it will be 1.

That is my first point, correctness is a relation defined on a product space of problems and solutions, or, in software terms, a relation between specifications and program texts.

Viewed in this way, correctness has a definite 'a posteriori' flavour, since it can only be verified or refuted when one has both elements of the constituent space, that is both the full specification and the program text. One of the aspects of change that we are witnessing now is an attempt to change this 'a posteriori' nature of correctness. It is important to realise that, whether 'a posteriori' or 'a priori', it is meaningless to analyse only the program text to determine whether or not it is correct.

The classical methods of correctness analysis, the 'a posteriori' methods, start with a complete specification and a program text and try to discover whether the latter is correct. I include in these methods the time honoured practice of debugging, which some people feel is wrong because it does not pose sufficient intellectual challenge or because it may 'show the presence but not the absence of bugs'. Debugging plays the same role as proof reading does in mathematical proofs. No mathematical proof is correct unless it is correctly represented. Similarly, no program is correct unless the text is correct. That is, unless the text is the correct embodiment of the idea of the program. In this role debugging is necessary. Other typical examples of the 'a posteriori' methods are program verification, which attempts to decide whether the point in the product space representing the given program text and the given specification lies in the appropriate relation subset, and testing. In problems which exhibit continuity, in some sense, a combination of thorough testing and extrapolation is as satisfactory

a proof as any mathematical proof.

Unfortunately, these methods share one basic shortcoming: the length of the proof text. You will find that the texts of all 'a posteriori' proofs are very long. Where this appears not to be the case it is because some essential step has been omitted. This applies very often to debugging. A thorough report of the debugging performed is seldom seen, even if a large amount was done. If this report were to be included, the text would not be any shorter than that of one of the fancy mathematical proofs which might appear in the literature. As a rule of thumb, the text of an 'a posteriori' method of proof is between one and two orders of magnitude larger than the text which is being proved. And although length is only a poor measure of the effort involved in its construction, if the proving and programming efforts are measured by the lengths of their respective texts, we must come to the terrible conclusion that the proof is an order of magnitude more effort-consuming than the program itself.

Hence, it appears that we should treat the proof as the primary and the program as a secondary by-product. We can see this approach in the 'a priori' methods of establishing the correctness, which attempt to make program texts correct by virtue of construction. In these methods we try to formulate the problem in such a way that it naturally leads to an 'a priori' proof of correctness. As before, we start with a complete specification, but we also use the basic axioms of correct transformations and we apply a schemata of correct composition. These ingredients are also used in the 'a posteriori' methods, but in the role of tools of analysis, whereas in the 'a priori' methods they are construction tools. The 'a priori' methods are better suited to our abilities since they reverse the positions of primary and secondary tasks. Unfortunately, while I found only one shortcoming for the 'a posteriori' methods, viz. the length of the proof text, I thought of three shortcomings for the 'a priori' methods: How does one guess proper schemata, how does one direct one's design activity and how does one apply the method

to problems of a non-transformational nature? These are shortcomings of an entirely different nature. In the 'a posteriori' case, we encounter a physical fact, 'the proof text is too long'. In the 'a priori' case, the problems all start with the word 'how'.

Let me now explain what I mean by 'problems of a non-transformational nature'. It has struck me as odd that whenever I see a beautiful example of a well-composed program, the problem for which it has been composed is extremely well defined as a transformation. One can sense that the problem is a clear cut transformation of the input state into the output state. It is very seldom, in my experience, that real world software problems are so well defined as transformations, and I dare to say that solving even the most difficult transformational problem is trivial compared with the effort required to produce a transformational specification for a practical software problem. Moreover, the latter effort very often involves a large amount of extra-programming knowledge. I shall try to illustrate the amount of extra-programming knowledge which is required to solve the exemplary problems which are published and exhibited to teach us how to solve normal problems. Consider for example 'Compute the largest prime factor of a given natural (integer > 0) number'. Since most of us have some knowledge of mathematics, we do not protest at being given such a specification, but 'prime factor' is a purely technical term from another discipline which no programmer has any business knowing. Some programmers might ask 'what is a natural number'. Since there are different schools of thought on the matter I included 'integer > 0' in the specification. Although there are many excellent examples of how to solve this problem, they <u>all</u> rely on extra-programming information. The beauty of the program-solution lies not in the programming but in the use of several facts which are unrelated to programming. A programmer who knows no number theory is likely to ask 'what is a prime?', 'what is a factor?' and even 'what is "largest prime factor"?', but he will never know, nor ask for the information, that the largest prime factor of an integer is not

greater than the square root of the integer, which gives an extremely valuable hint on how to construct a solution.

Since one is not provided with transformational specifications very often and they seem to be an essential ingredient of a methodological, disciplined approach, is there another guiding principle which can be applied to programming as problem-solving?

Increasingly, I am coming to the belief that it is the problem oriented analysis that should direct programming activities. This gives rise to many practical decisions that must be made. First of all, referring to the famous top-down versus bottom-up controversy, I find it helpful to do both. At the first NATO Software Engineering Conference, Stan Gill argued that all implementation was a mixture of top-down and bottom-up methods, but since the religious wars broke out, such compromises have been forgotten. I find it profitable to have different guiding principles when doing the analytical (or top-down) part of the design and when doing the synthetic (or bottom-up) part. In the analytical part, I think as a user, that is, in terms of the problem. The gross design and initial decisions seem to be done better if they are not based on programming considerations, but on the basis of how the program is to be used and what problem it has to solve. We should not lose all programming aspects from the analytical reasoning, though. I would like to do the problem analysis in such a way that I have a chain of reasoning which has two basic properties. The best way to describe these properties is by the mathematical terms 'completeness' and 'consistency'. If we know how to make the problem analysis in such a way that the solution remains complete and consistent as development proceeds we would be much closer to solving the basic problems of program construction.

I believe that we have some of the necessary tools in incomplete specifications. Complete specifications never exist since a complete specification would include the program text, the compiler text, the operating system text, the engineering specifications of the machine, the blood pressure of the operator and so on. All specifications are incomplete. I am not discussing all possible incomplete specifications, but incomplete specifications such as Parnas' [1] functional specification of a module (in terms of the observable changes in the state vector), DeRemer and Krons' [2] hierarchical decomposition and 'use' function, and Guttag and Hornings' [3] syntactic (I/O transformation), semantic (axioms of transformation) and implementational constraints. Incomplete specifications are a fact of life for managers, imposed by evolving specifications and the 'staged payment contract'. These are variously viewed in the profession as a curse and as a benefit. I hold the latter view. Since fixed specifications are a myth, it is better to admit the fact and follow the evolving specifications, which have obvious user appeal. They equally appeal to the manager of the project who is never more than one stage unfunded and can get the user 'hooked on' the system gently, rather than hitting him on the head with delivery of a complete system, which causes a natural rejection by the user. Therefore, I suggest that incomplete specifications should become a consciously applied tool in program development. This presupposes bringing in the user at each stage of the design, which is very unpleasant for our ego, since it is not any longer the programmer who is in the driver's seat during the program design. This is not the paradox it might seem because we should never program for the sake of programming.

If this principle is consciously applied, we have corresponding to the layers of program, layers of specification. These start with the statement of the problem which is successively decomposed into a number of incomplete specifications. At each decomposition stage I would try to obtain a proof of completeness and consistency using the terms and relations of the level of specification reached. An amazing amount of progress can be made by applying formal methods.

to informal objects. (It is a common misconception that one must have formal objects to apply formal methods.) In the design of a program we are translating from the user's, problem-oriented terms to the program oriented terms. This transition is very difficult unless you apply formal reasoning.

The design process is not program refinement, but the completion of the specification which invariably brings in extra-programming knowledge. There is, of course, a limitation. If we have formal reasoning with the objects treated formally, it is very difficult to map the formal objects of one level into the formal objects at lower levels. This difficulty is considerable if we treat formally arbitrary objects. If we do that then the bringing together of consecutive layers may prove to be rather difficult. I see a glimmer of hope in the algebraic methods of describing programming objects such as that developed by Guttag.

Recall that correctness is defined in terms of problems and solutions. Now, the other highly abused term, 'reliability', is defined in terms of objects and uses, rather than problems and solutions. As soon as one realises this, one has no more doubts that correctness and reliability are independent notions and they do not in any way entail one another. You may also recall that in programming terms correctness translates into the relation between specifications and program texts, whereas reliability is the property of a program in use. As long as we have just the program text, it is rather meaningless to say whether it is reliable or unreliable, except if we have uses for that text other than execution by such an executing agent as the case may be. From this we conclude that reliability cannot be established from analysis of program and specifications texts alone. This is perhaps a very common and very obvious statement. But, at least for my private use, I find it very illuminating and instructive, and it saves me a lot of effort: I don't read any attempts to establish the reliability of a piece of program in its textual form. I have learned by hard experience not

to read any considerations of reliability based on the program text alone.

Now, of course, 'reliability' is a very messy notion. To someone whose natural language is not English, the notion of reliability is extremely confusing. It appears by the properties of English, that 'reliability' is the primitive notion and 'unreliability' is the derivative notion. In some languages, Polish included, it is the other way around. Somehow I think that ours is a more humble mental attitude, and it helps us to understand that the primitive notion is failure, and reliability is the lack of failure if you're lucky!

I have many colleagues, especially at technical universities, who believe that reliability is a statistical notion, and is a number between 0 and 1. They take great delight in decomposing objects and then attaching these numbers between 0 and 1 to the 'sub objects' obtained by decomposition. Then, they multiply these numbers, obtaining successively smaller numbers. Somehow, I fail to see the relevance of that sort of reasoning for our problem, although this apparently charming approach is a good subject for Ph.D. dissertations. The program can be decomposed into modules, the modules into constituent parts, the constituent parts into instructions, and the instructions into sequences of micro-instructions. This goes down to the gate level and here, at last, we reach the reliability as measured by technicians in numerical terms. Since this is only a finite expansion, after only a finite number of multiplications you will arrive at a numerical value of the reliability of any piece of code (correct or incorrect). This is a perfectly valid 'scientific' approach. However, an estimate of program reliability obtained in this way will be too pessimistic because of the tremendous number of components which may go wrong, and yet would tell us nothing about the semantic aspects of failures. For that reason I believe this is a totally wrong approach.

I always try to think of software failures as belonging to two distinct classes. The first type is the failure of a program to perform as specified. This may be caused, for example, by mismatched data - data which is just slightly wrong and, if the specification is vague enough, then one could argue that even with mismatched data the program should perform in such and such a way. To the second class belong failures to perform safely in unforseen circumstances. Of course I am now committing the logical crime of replacing one undefined term by another. You will be quite right in wondering what I mean by 'safely'. However, I think that the use of the term 'safely' is a mental step forward because this term is so obviously related to the user. Of course, very few programs physically blow up the computer installation when executed. However, the user is quite likely to describe what he considers is unsafe for him. A violation of user safety is, for example, the destruction of master files in a bank or in an insurance company: obliterating master records is one thing your system should always avoid. Another example is the destruction of data from a very expensive and unique experiment which you hand to your assistant for processing. I just want to stress that it is important to identify what is unsafe from the user point of view. Our aim should be that under no circumstances will a program that we deliver endanger the safety of the user. This is a basic aspect of program reliability, as I see it.

How do you achieve program reliability? How do you eliminate the failures from your system? Well, you don't - you cannot. Nothing in this world (the world included) is perfect: even engineering construction is not perfect, despite what some people say about mechanical engineering feats and so on. A bridge in Vienna which lasted since before the war, which stood through the bombing, collapsed one night last month for no apparent reason. Nobody knows where the bug was, and that wasn't even a mechanical construction - it was a civil engineering construction. Since nothing is really perfect, no program will ever be absolutely reliable, except when it's specified to do nothing, in which case don't bother to switch

the computer on so as to save electricity.

The techniques of defensive programming which provide alternate courses of computation if something wrong is detected are an obvious, although not necessarily economical, way out of the reliability problems. It is difficult to say something interesting on recovery blocks in Newcastle, but if you think in other terms about the recovery block structure, you may discern that one uses the primary block as the program aimed at achieving a desired result, whereas, the alternatives are intended to be just acceptable. Now if we make this distinction, then we have come across a totally new approach to program design. Normally we are programming to achieve an objective or a set of objectives. Sometimes it is advisable to be more humble, and say 'well if I can't get this or that, what are the things with which I will be nevertheless satisfied or at least dissatisfied?' Again, these things, in my opinion, should be stated at the very beginning of the design stage.

Another aspect of reliability which has recently gained some support amongst programmers is security, represented for instance by capabilities (that's a nice misnomer, if I ever met one) in the HYDRA project that Bill Wulf [4] is working on. Here you impose restrictions not only on the things accessible, but also you admit that the right to access may be a function of the execution process. By doing this you enlarge the degree of safety control in your program.

I will just briefly mention some less frequently considered aspects of reliability. One of these is unreliability of software which comes from poor user documentation. Insufficient user documentation is an invitation to disaster. Another aspect of unreliability which is not frequently discussed is the by-product of ready-made software packages, which have been my personal enemy ever since I started programming. I wasted a tremendous amount of time by using pre-packaged universal solutions rather than designing specific solutions from scratch. One time I was doing a lot of

numerical computations in astronomy which, of course, require many trigonometric series expansions. I was using the simplest of the packages: the sine and cosine routines, which calculated the sine and cosine functions to the full precision of the computer, even though I had only three digits worth of coefficients. I would have been quite happy with three-digit accuracy which can be obtained from a small table with simple interpolation; this speeds up the program about fifteen times. That's not unreliability, but it explains why I do not like packages. Packages are unreliable because they are given to users who will seldom consider whether it is applicable to use a given package or not. If you give the user a numerical integration package, he will integrate strongly discontinuous functions to his heart's content, and he will believe the numbers that he gets because they came out of the computer.

At this point Professor I.C. Pyle raised a question over this integration example. He wanted to know which definition of failure did it satisfy? Professor Pyle was of the opinion that the program had performed safely and it had performed as specified. He suggested that it was not consistent with Professor Turski's definition of reliability and that perhaps an even broader definition of what was wanted from reliability is needed. Professor Turski maintained that the program did not perform as specified because the specification would probably mention that the package is not to be used for strongly discontinuous functions and so these specifications should be carried out in the program and therefore an attempt to use the package for such application should result in a rejection. Besides, producing a 'reasonably' looking, yet meaningless output certainly endangers the user's safety.

Part 2

In the second part of this talk, I intend to cover only two topics. One topic is the software programmer's inventiveness of creativity. This is one of the biggest vices a programmer may have in an industrial environment. I hasten to add that I consider creativity to be one of the fundamental virtues we should instil in a programmer when educating him. (That's a beautiful contradiction which is just life itself.) Now, what do I mean by a programmer's creativity or inventiveness? In an industrial environment it is a preoccupation with self-posed and self-feeding problems. It is the programmer's arrogance to give preference to the programmer's very own insignificant problems over the problems of his clients and customers. I do admit that it might come from the very fast development of our profession and that it might be partially due to the fact that the programmer considers himself as the high priest of new religion in the industrial environment and therefore, of course, his occupation is the most important. However, we will be much better off if the programmers who do the commercial programming realise that they are performing a service, and this it is the client's problems that are important.

A programmer who is inventive will take the client's problem and spend six months thinking how this problem may be re-formulated in order that a new programming technique may be applied. This attitude has to be cured. Another effect of programmers' inventiveness is the blank rejection of imported software (which is everything not done by the programmer and his colleagues). Speaking now from under my managerial cap, the problem of 'a priori rejection' of imported software is a part of the inventiveness syndrome. Another symptom of this syndrome is all those 'wouldn't it be nice if' programs. Each programming establishment that I know abounds in such programs which lack applications. A further reflection of the programmer's creativity is his dislike of menial maintenance jobs.

By this criticism of creativity and inventiveness I put myself in an untenable position, so I hasten to add that I do see the beauty of creativity. Now I can safely proceed to list the cures for the programmer's inventiveness. Most of all, we must teach problem oriented thinking. We must insist that our programmers think in terms of the problems they are solving, not in terms of the programming techniques they are applying. Another cure is to let the programmer know the global objectives of the program he is working on, even if he is doing only a small portion of it. He will apply this knowledge in his attitude towards what he is doing. For example, if he were working on a banking system then he would raise an alarm if he discovered anything that could destroy the master files. If he were just given a module to program he may not react to things like that.

Thirdly, I am in favour of the technique of staff 'brain-storming' sessions where all the people on a project gather periodically to have a burst of controlled inventiveness. The ideas are frozen after the session is over. This session allows the programmer to display his inventiveness among his peers (which is a very good thing). In addition, it helps to bring the global objectives down to the level of each programmer.

Yet another cure for arrogant inventiveness is persistent documentation. The necessity of providing full documentation has a curbing effect on the unbridled inventiveness.

Another managerial tool in which I happen to believe very strongly is evolving specifications and staged payment contracts. These blend the programmers and the users represented by managers or others. I would like to mention Professor Barron's letter in Computing (I believe in January) in which he insisted that staged payment is the best protection against the incompetence of software producers. I would like to add that it is also the best protection that software developers have against the users who change their mind twice a week. An additional technique which is clearly didactical

is to instil the responsibility for one's product.

I believe the dull jobs should be given to the best programmers and the most attractive assignments should be given to the novices. Giving a novice a maintenance job is one of the best ways of encouraging the inventiveness that we have been talking about. The immaturity of our profession shows in the dislike of the maintenance jobs.

The programmer's inventiveness should be taught in an academic environment. Inventiveness is needed because we do face a great many problems for which solutions have to be invented. How can you teach inventiveness at university or in a programming course? Unfortunately it cannot be done by showing examples of somebody else's inventions; however, it is useful because it helps the person acquire a 'taste' for inventiveness. It is very difficult to be creative without the appreciation of the beauty of other people's programs.

Here are a few ideas which I think are useful in trying to encourage the development of inventiveness. First of all, vaguely specified problems from outside computer science should be given very early in the course (even before standard techniques are taught). Universities have an atrocious habit in giving problems with terribly exact specifications. I believe this is wrong: primarily because in real life the students will not meet such specifications and secondly because such complete specification is only just short of describing how to solve the problem.

Universities have the bad habit of extremely good researchers giving courses, and a researcher giving a course on his subject will, in general, explain his particular approach which is not necessarily the only approach or even the best approach. Hence, we should insist that critical surveys be presented with the relative merits discussed.

All sorts of games and competitions in programming should be encouraged during the programmer's education. For example, they could be organised like Jim Horning's software huts which are great fun and teach a lot. A technique which I like very much is to have the students review other student's assignments. Creativity is developed by requiring complete documentation but allowing considerable latitude in practice. For example, I do not say what 'complete' means and then I ask the student to convince me that his documentation is in fact complete.

(Note: at this point time was running out for the session so Professor Turski quickly showed his last two slides on buying software. The following is a summary of his ideas on this subject.)

Existing software can be either inherited or purchased. In either case it has to be maintained, adapted in the latter case, and extended sooner or later.

1. Somebody must identify himself with an existing system, treat it as his own, and be the 'in-house' expert. Invent a job to achieve it (translate I/O messages, diagnostics etc; balance and tune to preferred hardware/documents).
2. Never let it rust (purging from library).
3. Even if modifications are reasonably small try to re-edit user documentation (periodically).
4. If possible, test for reliability by overloading.
5. Never let junior programmers maintain existing software.
6. Be cost-effective conscious: investment in existing software and routines is large but not always worth maintaining forever. A small revolution may be cheaper in the long run.
7. Carefully analyse alternatives.

Buying:

1. Turn-key contracts including evolving specifications, staged payments.
2. Insist on proper education including your programmers in development team.
3. Insist on full access to documentation (you will not get it, but observe where the line is drawn).
4. Buy problem solutions, not particular techniques!

When buying is advisable:

1. As a part and parcel of complete system delivery (not a computer system, but e.g. a factory) if contract specific on system goals.
2. For a well-defined application (preferably from a user, or a copy of user's version).
3. When standards of quality are reasonably well-established (e.g. a compiler).

When buying is inadvisable:

1. When the documentation is clearly insufficient to permit in-house modifications and the software is for an evolving problem (examples: data bases, telecomputing).
2. Very general packages (they are unreliable).

Discussion

Professor I.C. Pyle opened the discussion by inquiring what Professor Turski meant by a 'very general' package? He maintained that packages and libraries were very useful and cited SPSS and the NAG library as examples. Professor Turski responded that by 'very general' he meant a package which was aimed at a totality of problems. You would just give the problem to the package, and it would decide what to do next. Professor Pyle asked for an illustration, whereupon Professor E.S. Page stated that there were quite a number of general statistical packages which provide a negative

estimate of variance for suitably chosen input data and users are not warned about the choice of input data. At this point Miss E.D. Barraclough came to the defence of Professor Turski. She thought he meant that if the package does too much for one, then the user doesn't have to understand what he is doing, but that if the package does too little then the user has got to understand it before he can use it. Professor Turski agreed that this was indeed a clear statement of what he had meant.

References

[1] Parnas, D.L. A technique for software module specifications with examples. CACM 15 No. 5 (May 1972).

[2] DeRemer, F., Kron, H. Programming-in-the-large versus programming-in-the-small. Proc. of the International Conf. on Reliable Software April, 1975.

[3] Guttag, J.V. The specification and application to programming of abstract data types. (Ph.D. thesis). U. of Toronto, Department of Computer Science, 1975.

[4] Wulf, W.A. Reliable hardware-software architecture. Proc. of the International Conf. on Reliable Software, April, 1975.