“The Role of Error in Computer Science” first appeared in *Philosophia* Vol. 19, No. 4 (1989), pp. 403-416.

The aim of this paper is to explore the concept of error as it applies to computer science. Now, it is customary in computer science (hereafter "CS") to distinguish between hardware (the actual computing equipment) and software (the programs which run on that equipment). We will not be concerned with error in hardware, since that is a matter of engineering design and manufacturing quality control. We are specifically interested in the notion of error in software. For the purposes of this paper, CS will be synonymous with software science. There are a number of interesting features to error avoidance in software that bear examination.

We can distinguish various levels of software. At the most basic or micro level, there are algorithms, that is, step by step logical procedures, which become elements of programs when implemented in a programming language. Next up in the hierarchy are the medium sized programs that function to accomplish specific tasks such as (say) sorting personnel files or computing the trajectory of a rocket. We can call this the mini level. At a third or macro level are large-scale integrated programs, such as air-traffic control or commercial word-processing programs. These high level programs generally are produced by teams of computer programmers and analysts over months or even years, and usually involve very sophisticated "user-interfaces" (i.e., elements which allow untrained data entry clerks or businessmen to be able to run the software successfully). Finally, at a "meta" level are programs in Artificial Intelligence which seek to mimic human reasoning.

The distinction made above (between micro, mini, macro, and meta level software) is of course somewhat arbitrary, and leaves out much of out much of the story (such as the sorts of data structures and programming languages used). But it will do to allow us to frame our survey of techniques of error analysis and avoidance. I will allow the term "error" to remain only implicitly defined for now; after we survey some material we will be able to define the term more precisely.

Let's start at the micro level, i.e., at the level of algorithms. The study of the efficiency of algorithms is a highly mathematical branch of theoretical computer science called *computational complexity.* Computational complexity applies mathematics to the measure of the time and space requirements of algorithms, that is, it aims to determine precisely how much running time and how much memory a given algorithm will need. Let's review computational complexity analysis briefly.

Simplifying considerably, and speaking somewhat imprecisely, we say that a function g(n) is "on the order of" some other function f(n) if g(n) grows no faster than Kf(n) after a certain value of n and for some positive value K. For example, the function 3x2 + 2x + 5 is on the order of x2, since it grows no faster than (say) 4x2 after x = 5. Typically, the complexity analyst uses the following 0-notation to characterize classes of running times:

|  |  |
| --- | --- |
| constant | 0(1) |
| log log | 0(log log n) |
| logarithmic | 0(log n) |
| linear | 0(n) |
| n log n | 0(n log n) |
| quadratic | 0(n2) |
| cubic | 0(n3) |
| exponential | 0(2n) |

(Here, n represents some basic measure of the size of the task the algorithm is meant to accomplish, for example, the number of positions in a table to be searched, or vertices in graph to be examined). Algorithms are evaluated as to their complexity by determining into which classes they fall. An algorithm that runs in linear, logarithmic, or log log time is considered to be fast. An algorithm that runs in exponential time is considered to be hopelessly slow. To get an intuitive feeling for these classes, Standish (1980, p. 14) gives this table:

|  |  |  |  |
| --- | --- | --- | --- |
|  f(n) | n = 2 | n =16 | n = 1024 |
|  log log n | 0  | 2 | 3.32 |
|  log n | 1 | 4 | 10 |
|  n | 2 | 16 | 1,020 |
|  n log n | 2 | 64 | 10,200 |
|  n2 | 4 | 256 | 1.05 x 1012 |
|  n3 | 8 | 4,100 | 1.07 x 1018 |
|  2n | 4 | 65,000 | 1.80 x1038 |

By way of comparison, consider two algorithms for searching a list of names, one of which runs in linear time and the other in exponential time. In the time it takes the exponential time algorithm to search a list of forty-four names, the linear time algorithm could search a list of thirty thousand *billion* names.

Determining the actual time (or space) complexity of an algorithm usually requires subtle mathematical argumentation. For one thing, a distinction must be made between worst case, best case and average case time/space complexity. Also, a distinction must be made between the running time and the preprocessing time of an algorithm. But while the technical details of such proofs are not of much interest to the philosopher, the existence of such techniques should be. For it occasionally occurs in the literature of Artificial Intelligence that an author will claim to have devised a program which supposedly mimics some aspect of human reasoning, yet which turns out to be exponential in time complexity. When run on tiny ("toy") data sets the program does the job, and much fanfare is raised about AI growing closer to achieving full machine intelligence. Such a claim can be resisted if the inherent limitations of the underlying algorithm are recognized, since obviously the human mind does not operate exponentially slowly.

Let us move next to the level of mini scale programs. At this level, work on error avoidance has focused on the formal specification of programs. Considerable effort has been spent developing techniques for specifying formally the requirements for a program, i.e., for specifying what the output and input should be, and verifying whether in fact it will produce that specified output given that specified input. The formal approach to program verification was first explored by Floyd and Hoare in the 1960's (see Floyd (1967) and Hoare (1962)). A brief review of this work is worthwhile.

We begin with the concept of a specification, consider small routine to compute x2 + 2, given that x = 2:

Start

 | [X:=2]

|

a-----------------|

[Temp:= X]

|

b-----------------|

[X:=X2]

|

c-----------------| [X:=X+Temp]

|

d-----------------|

[end]

At point a, after assigning 2 to X, we require/specify that X = 2. At point b, temp should have the value 2 as well. At c, X should have the value 4. At d, X should have the value 6. These statements about the values of the program variables at each stage in the program are called assertions. We can specify the conditions before and after any statement in a program by assertions, and (since a program is a sequence of simple or compound statements), we can specify the input and output of any program by assertions. Precisely defined, the specification

{P} S {Q} specifies that if the computer executes instruction S in any state described by P, then if S terminates, it will be in a state described by Q. P is called the pre-condition and Q the post- condition. Thus, looking at post-condition C in our earlier example, we can write (X = 2) X: = X2 (X = 4).

Now, again, a program is just a sequence of simple or compound instructions, so we can represent a program with the pre-and post-conditions of the statements as a tableau (with the program input as the first precondition and the program output as the post-condition). In our simple example, the tableaux would be:

(true  X: = 2

(X = 2)

temp: = X (temp = 2) X: = X2

 (X = 4)

 X = X + temp

 (X = 6)

Such a tableau is valid if (a) when a triple (P) S (Q) occurs in the tableaux, it is true; and (b) whenever a pair of the form (P) (Q) occurs, assertion P implies assertion Q. We can utilize inference rules to establish the validity of proofs. A few of the standard rules are:

Assignment:

{P|EX }X: = E{P}

where {P|EX} stands for the result of substituting E for X in P.

Compounding:

1. S1 (Q)

(Q) S2 (R)

 --------------

1. S1S2(R)

 Strengthening Precedent:

P Q

1. S (R)

---------------

 (P) S (R)

Weakening Consequent:

 Q R

 (P) S (Q)

---------------

 (P) S (R)

Of course, more rules are needed for a complete set. But with these we can work through a simple example:

(X < 5 & X > 1)

Y: = 3X

(Y < 15 & Y > 3)

Z: = Y (Z < 15)

Step 1:

-------------------------------------------------- (X < 5 & X > 1) Y: = 3X (Y <15 & Y>3)

by assignment.

Step 2:

------------------------------------------------- (Y < 15 & Y > 3) Z: = Y (Z <15 & Z >3)

by assignment.

Step 3:

(Z< 15 & Z > 3) (Z<15)

(Y < 15 & Y > 3) Z: = Y (Z <15 & Z >3)

----------------------------------------------- (Y < 15 & Y > 3) Z: = Y (Z <15)

By completing the above set of rules with rules for if-then and do-while statements, proofs of program correctness are readily obtainable.

The formal specification and verification of programs has been welcomed by most software engineers. In fact, computer scientists have extended these techniques in a number of ways. Some have used temporal logic for the verification of concurrent programs. It also may be possible to use erotetic logic to specify and verify database queries. And a number of programs (such as GYPSY) or extensions to existing programming languages (such as ANNA, i.e., annotated Ada) have been devised which allow the inclusion of assertions into the code and the automatic verification of programs.

At the macro level of software, the concern for error avoidance is central to the discipline called *software engineering*. (For a good overview of software engineering, see Pressman (1984)). Basically, the field of software engineering generally, and the "structured revolution" in particular, grew out of the software crisis of the 1970's.

In the 1950's and 1960's, hardware developed rapidly, but as it grew in complexity and power, software costs grew even more rapidly. By the early 1970's software became the biggest cost in computer systems, and cost/time overruns became rampant. "Horror stories" abounded of software projects disastrously gone awry, such as the multimillion dollar rocket which had to be destroyed when it strayed off course due to a simple programming error (viz., the truncation of a real number by default). (For a classic discussion of a case of a software project that led to a crisis, see Brooks (1975)). Moreover, the gap between what was needed and what was available kept widening rapidly.

The first (and narrow) response to the software crisis was the rise of "structured" programming languages and "top down" design methodologies. Specifically, it was proven (in the 1960's) that any possible program could be constructed out of just three structures: if –then- else (i.e., simple branching), do-while (i.e., simple looping), and sequence (simple direct instructions such as move and store). This led to a movement to eliminate "go-tos" in programs, that is, to write programs in which the program flow was linear, so could be followed easily (as opposed to jumping all over the place, making error detection difficult). To facilitate "go-to-less" programming, new languages such as Algol and Pascal were developed, and older languages such as COBOL and FORTRAN were revised. Moreover, it was recognized that programs could be more reliably designed and more collectively written if the tasks in a program were modularized, i.e., segregated into separate blocks of instructions. Again, the structured programming languages were created with an eye to encouraging the top-down decomposition of a program into modules.

Consider an example. Suppose we are writing a "program" for making an omelet. The major subtasks are: get the ingredients from the refrigerator; get the utensils; mix the omelet; and cook it. We can "code" this task top-down:

 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| start

| <get ingredients >

 Make | <get utensils>

 Omelet | <mix omelet>

| <cook omelet>

| stop

 |

We can then "program" the modules:

 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 | enter module;

 | open refrigerator;

 Get | take out the eggs; ingredients | take out the butter;

 | make omelet

 | take out the cheese;

 | return to the higher module;

 |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

and so on. To see how if-then and do-while loops can be employed, consider the manner in which we mix the ingredients. We have in mind using a certain number of eggs (say, six, if we are ravenous or cooking for two). We use a counter variable to keep track of the number of eggs in the bowl. We might also check the cheese to see if the end is dry, and if so, cut off the dry part before grating the cheese.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | enter the module;| set the egg-count to zero;| inspect the cheese;| if the cheese is dry| |  |  |
| | then | | cut off the dry end; |
| | | | grate the cheese; |  |
| | | |  |  |
| mix | | |  |  |
| omelet | | else | | grate the cheese; |  |
|  | | | |  |  |
|  | | |  |  |
|  | | |  |  |
|  | | do while the egg -counter < 6 | | crack an egg; |  |
|  | | | | split it over the bowl; |  |
|  | | | | add one to the egg-count; |  |
|  | | | |  |  |
|  | | |  |  |
|  | | pick up a fork; |  |  |

 | mix the ingredients with the fork;

 | return to the higher module;

 |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The idea of using a top-down approach and structured languages was first extensively tried and empirically evaluated in The New York Times IBM project in the late 1960's, and it was shown that programmers who used structured programming techniques were many times faster at producing error-free code than programmers who did not use the structured approach. This result was well publicized, and contributed to the rapid spread of structured programming.

But the structured revolution proved to be insufficient to solve the software crisis. A more broad set of techniques had to be developed to address software inadequacies. Software engineers came to realize that software has a "life cycle" that needs to be understood. Specifically, the development of any piece of software takes place in stages, and the development of error-free software requires the application of techniques appropriate to each stage.

To be more concrete, let us look at commonly accepted model of the software life cycle (although it is not the only accepted model, and in fact is a bit out of date):

Requirements analysis

 |\_\_\_\_\_\_\_\_\_design

 | coding

 |\_\_\_\_\_ testing

 | maintenance

This simplified model, often called "the waterfall model", views the typical development of software as beginning with a clear determination of the needs for which the software is to be designed to handle. Then the software is designed at a highly abstract level, with the implementation details left out. Only after the design is reviewed and clarified does the actual coding (i.e., programmers writing the detailed machine- executable code) begin. After the program is written, it is tested in hopes of removing all the major defects ("bugs"). Finally, the software is installed on the customers computer system, and long-term maintenance gets under way: small bugs that surface when the software is running full-time are corrected, and modifications needed because of the customer's changing requirements are made.

The point I want to make is that for each stage of the software life-cycle, techniques have been developed to minimize error. At the stage of requirements analysis, the goal is to produce a "software requirements specification," which is a precise statement (either informal or formal) of what the proposed software is to do. Many requirements analysis techniques have been developed, some manual and some automated (i.e., computerized), and a number of companies market these tools. One popular group of techniques involves the use of data flow diagrams. For example, we might specify the requirements for a program to do payroll for a small company as follows:

Level 1:

[Payroll Clerk]---hours worked--->{Payroll Program}---paycheck amounts--->[Paycheck Generator]

↑

<Employee Records>-------------exemptions---- hourly wage

Here the rectangles represent sources of external inputs or outputs, the circle a process which transforms data, the double line a data repository, and the arrows the transfer of data.

Level 2 (decomposes the payroll module further)

Hours worked------>{Gross Pay Calculator}----gross pay------->{Tax Calculator}---net pay---->

 

Hourly wage Exemptions

Notice that all we are trying to do is to specify exactly the requirements of the system, before we even try to design it. We can keep drawing specifying the requirements to any degree of precision desired. And again, computer-aided software engineering (CASE) workstations exist which allow the software teams to devise the data-flow diagrams automatically.

At the stage of design and coding, all the concepts of structured programming discussed earlier apply. Warnier diagrams, flowcharts, decision tables, and a dozen other devices are used commonly in the design of programs. And coding is done typically in structured high-level languages.

Techniques for improving the efficiency of the testing and maintenance phases also have been developed. One important class of these is techniques, called *metrics*, for measuring the amount of errors in a piece of software. For example, the eminent software engineer McCabe devised a metric that, when applied to the control flow graph for a program (a graph similar to a data flow diagram), estimates the complexity of the program. This, in turn, indicates the testing difficulty and (ultimately) the reliability of the software. Another class of useful software engineering techniques which apply at the latter phases of the software-life-cycle is the set of strategies for testing (such as "black-box", "white-box", top-down and bottom-up testing strategies). I will not go into those strategies here as they would lead us far afield. Suffice it to say that the software engineer has a tremendous number of tools which allow him to minimize errors at each stage of the software life cycle.

At an even higher level, and complementary to the work outlined above in software engineering, there is a whole realm of diverse work which might be called "social impact engineering" for lack of any commonly accepted term. One long-established area of social impact work is computer ergonomics, which aims to design hardware which minimizes negative impact upon the workers who use those systems. For instance, non-glare screens have been developed to reduce eye strain, and special ergonomic chairs have been developed to reduce back strain in the data entry clerks who must spend all day in front of the computer. Similar to ergonomic design, but more relevant to the topic at hand is the recent work in design of user- oriented (or "user- friendly") software is software designed to minimize the problems of the user using it. For example, one user-oriented consideration is the degree to which the software "forgives" improper data entry (such as entering negative instead of positive amounts when recording payments at a bank).

More broad still is the design of fault-tolerant software. Software now impacts not just the user but society at large. Software controls the running of nuclear power plants, air traffic control systems, ICBM's and so on. Since the social consequences of software failure in such cases can be profound, a growing concern is to design the software so that minor faults or errors result in the least social harm possible.

Let us move lastly to what I have termed the "meta" level. I have in mind here work in Artificial Intelligence ("AI") which seeks to design software which can "reason" under uncertainty or in the face of error. (I put the scare quotes around the word "reason" since I don't want to get embroiled in the controversy about whether machines can think.) This level is "meta" in the sense that the researchers are not seeking to design methodologies for minimizing software error, but are instead designing intelligent software that can itself deal with error.

I will not attempt an extensive review of the AI literature on the topics of reasoning under uncertainty and error. About reasoning under uncertainty, I will just mention the major approaches. One very well-known way of dealing with uncertainty is to utilize Bayesian inference, which involves assigning prior probabilities and altering them in the light of new information. Another approach utilizes fuzzy logic, which allows vague statements (such as "Chester is balding") to be expressed formally as partial class membership. Still another approach is the endorsement model, in which decisions are reached on the basis of pro or con endorsements. For example, uncertain about whether or not I should buy a Ford Bronco, I might call all the people I know who have owned similar cars, and ask each of them to give his opinion of the car. (For more detail on these approaches to uncertainty, see Cohen (1985)).

Regarding reasoning in the face of error, an AI approach of great interest is non- monotonic logic. Non-monotonic logic was developed to handle situations in which initial assumptions have to be retracted in the light of new information. For example, I might initially order an expensive meal in a restaurant, assuming that is accepts credit cards. Upon learning from the waiter that the restaurant does not accept credit cards, I may decide to revise my order to a less expensive meal. The term "non-monotonic logic" refers to the fact that, whereas in standard logic the addition of new axioms cannot decrease the set of theorems (so the set grows monotonically), in this new logic the set of theorems can shrink in size. Again, I will merely mention the major lines of attack. One line of research focuses on reasoning by default, that is, cases in which a rule is assumed to hold unless an exception is explicitly involved. For instance, the rule "birds can fly" applies by default to any bird, unless it is explicitly mentioned that the bird is a penguin. A second line of research focuses on reasoning by circumscription, that is, cases in which it is assumed parsimoniously that only the objects and relations mentioned explicitly are relevant to the problem at hand. For instance, in the simple algebra word problem

A car is travelling at 60 KPH towards a town 120 Km. away. How long will it take to reach the town?

it would be natural to assume that the car is not going to be vaporized by a bolt of lightning.

The four-fold hierarchy of software I have used to frame the discussion in this paper is, as I admitted at the outset, simplistic. There are sorts of software which fall between levels. Thus for instance I have not explored the realm of "domain algebra," which is an area of applied mathematics of great use in database theory. Specifically, domain algebra is a formalism directed at resolving anomalies which arise in the management of large stores of data (such as the information kept by a large insurance company on its policy-holders). Databases are of great and growing interest to computer scientists, and the programs which manage them (called "database management systems") are of special importance. But I trust that our survey has been complete enough for us to draw a few broad conclusions about the nature of software error and its role in computer science.

To begin with, what is of interest to computer scientists (as opposed to professional programmers, systems analysts, and computer operators) are not particular pieces of software *per se*, but rather the methodologies by which software is produced. That is, part of what makes a computer scientist a scientist is that he is interested in the process of producing error-free software, rather than in maintaining a particular product or system. Naturally enough, the failures or successes of particular products of a given software design process are grist for the mill of improving software methodology, but the focus is still on process rather than product. Note that philosophers who distinguish between discovery and justification, and view only the latter as being of epistemic importance, are ill-suited indeed to appreciate the essence of computer science.

A second point is that the evaluation of rival methodologies in computer science is usually attempted empirically (or at least mathematically). For example, the claim that the structured approach is a better approach to programming than the unstructured approach has been tested repeatedly by empirical studies. Again, various software metrics have been compared against test programs to see which metric has the best predictive power. This tendency to test empirically is again the mark of a science.

A third point to note is the desire of the computer scientist to move to higher orders of design. That is, no sooner is a methodology for designing software enunciated then some computer scientist is attempting to automate that methodology. If data flow diagrams prove useful in stating the overall requirements for a software project, well, then, it is not long before a computer program is written which produces data flow graphs from a statement of the basic data sources, transforms and data items. If a metric which estimates the number of errors in a given program from the number of key words in it proves useful in testing and maintenance, it is not long before a program is written which estimates errors (using that metric) given other programs as input.

To summarize the thesis of this paper, in the realm of software science, an *error* is a failure of a given piece of software to perform as required in a given context. This includes errors in the narrow sense of algorithms which fail to terminate or terminate with an erroneous result, to the broader sense of software systems which fall short or the performance specified in the approved requirements analysis documents, to the very broad sense of software whose functioning involves unacceptable social risks. The role of error in computer science is in fact positive: software errors are what fuel the search for better methodologies for detecting and eliminating error. In this latter trait, computer science is not unlike other sciences, in which theorizing often is driven by observed failure of prior theory.

**Gary James Jason**

**Department of Philosophy**

**San Diego State University**

BIBLIOGRAPHY

1. Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley (1974).
2. Brooks, F., *The Mythical Man-Month*, Reading MA: Addison-Wesley (1975).
3. Cohen, Paul , *Heuristic Reasoning about Uncertainty:. An Artificial Intelligence Approach*, Marshfield, MA: Pitman Publishing (1985).
4. Floyd, R.W., "Assigning Meanings to Programs," in *Mathematical Aspects of Computer Science*, ed. J.T. Schwartz, *Proceedings of Symposia in Applied Mathematics* 19, AMS, Providence (1967).
5. Gries, David, *The Science of Programming*, Berlin: Springer-Verlag (1981).
6. Hoare, C.A.R. (1962) "An Axiomatic Basis for Computer Programming" CACM 12 (10)
7. Pressman, Roger *Software Engineering: A Practitioner's Approach*, N.Y.: McGraw-Hill (1987).
8. Reynolds, John, *The Craft of Programming*, London: Prentice-Hall International (1981).
9. Standish, Thomas, *Data Structure Techniques*, Reading, M.A.: Addison-Wesley (1980).