Formally Unsolvable Problems in Software Engineering

J. P. Lewis
zilla@computer.org

Engineering has been defined as the solution of practical problems using mathematical and scientific principles. Is it possible to apply mathematical and scientific principles to the solution of important problems in software engineering, such as the prediction of development schedules and the assessment of productivity and quality? The debate over this question is familiar. In this article, we will look at software engineering from a different point of view, that of algorithmic complexity. *Algorithmic complexity* (see sidebar) defines the complexity of a digital object to be the length of the shortest computer program that produces that object. We will find that algorithmic complexity results can be directly interpreted as indicating that software development schedules, productivity, and reliability cannot be objectively and feasibly estimated and so will remain a matter of intuition.

The state of software engineering

There are a large number of design methods, development processes, and programming methodologies that are said to improve software productivity, predictability, and reliability. Despite the existence of these techniques apparent large scale failures of software development are a regular occurrence. During the writing of this article, for example, it was reported that the state of California is considering scrapping an unfinished $300 million system intended to track child support payments, recent failures to modernize software at the Internal Revenue Service were divulged (cumulative cost of these efforts: at least several billion dollars), and the ship date of a new release of a major personal computer operating system was rumored to have slipped into the next calendar year.

It seems fair to say that if the solution to the software crisis is in the published literature waiting for adoption by all, there is not yet agreement on which of the many techniques it is. Each technique has its proponents and arguments in its favor, but it is also easy to find arguments against most proposed techniques. For example, the claimed benefits of object-oriented programming have been questioned.[1,2] Naur has written extensively on the misuse of formalisms. In one study he found no relation between the use of formal descriptions and program comprehension.[3] Gabriel argues that staged development methodologies "are among the least productive and possibly produce the lowest quality".[4] Glass surveyed a number of methods and found that there is little research that supports their quality and productivity claims.[5]

Given the wide variety of competing claims one may be tempted to adopt an outside perspective and ask whether all of the stated goals of various software development methods are possible even in principle. In fact we will show that any claims of objectively characterizing software reliability, absolute productivity, or expected development times are unfounded.

Software predictability

Software seems to be delivered late as a rule rather than an exception. Late software is not necessarily of poor quality, nor are the programmers responsible for late software necessarily unproductive. Poor estimation of software development time may nevertheless be the software industry's largest problem — software teams loose credibility, promising projects are canceled because their ultimate cost is unknown, and companies are fiscally threatened. Brooks[6] writes that "More programming projects have gone awry for lack of calendar time than for all other causes combined."

Various software design methods and processes address the issue of predicting development times. Software cost models estimate development time as a function of a size measure such as source line counts or function points. Software process literature and commercial software management tools have suggested that historical data on development times can help in the prediction of future projects.

In one of the best researched empirical studies to date, Kemerer benchmarked four software effort estimation algorithms on data gathered from 15 large projects for which accurate records were available. It was found that these models had only limited predictive ability in ex post facto estimation of the development times for completed projects — the selected error measure (magnitude of error normalized by the actual development time) ranged from 85 percent to more than 700 percent.[7] Despite poor statistical efficiency of these models they are objective and have some predictive value. Kemerer indicates that the limited accuracy of these models may be accounted for by variations in problem domain and other factors, and that the models may be tuned to be more accurate in specific development environments.

The limited accuracy of these models is not the biggest obstacle to software predictability however. Rather, since cost models are a function of a size or complexity measure, the issue is how to estimate the size or complexity of a new project. Consider the following scenario: A software company wishes to improve the predictability of its software process, so it decides to gather data on development times. To speed this effort, it decides to give each programmer a timed series of exercises. It is found that the average programmer at the company can complete each exercise in an average of

3.7 hours. Now the company is asked to bid on the development of an object-oriented operating system for a major company that is years behind schedule on their own operating system development project. Based on the newly gathered data, the company estimates that it can deliver the new operating system in about 3.7 hours using one average programmer.

This absurd example is intended to clearly illustrate that estimates of development time depend on estimates of program size or complexity, and historical data do not provide the latter.

Can complexity itself be formally and feasibly determined or estimated a priori? Here and elsewhere in this article, 'formally' means that there should be some formal process or algorithm that could be followed to allow independent and disagreeing persons to objectively arrive at the same estimate, thereby removing estimation from the realm of intuition. We assume the Church-Turing thesis, i.e., that any such process is essentially an algorithm even if the process is in fact followed by humans rather than a computer. By 'feasible' it is meant that there is some process or algorithm as above, and that the algorithm can be completed in a reasonable amount of time assuming normal human and computer capabilities.

Claim 1: **program size and complexity cannot be feasibly estimated a priori.**

Algorithmic complexity shows that the minimal program size for a desired task cannot be feasibly computed, and simple reasoning will show that a trivial upper bound on program size exists but is not useful.

Before discussing these results further we need to relate algorithmic complexity to real world programs. Recall that algorithmic complexity (AC) is defined as the minimal program size needed to produce a desired output string. Since this definition deals with output only we will briefly sketch how input, state, and interactivity might be accommodated:

- *interactivity:* An interactive program can be considered as a collection of subprograms that are called in sequence according to the user commands. These subprograms will share subroutines and a global state. Ignoring input and state for the moment, the AC of the program is the AC of the combined subprograms, plus the AC of a small event loop that calls the subprograms based on user input.
- *input:* The AC of a function that depends on arguments can be defined as the AC of a large table containing the input-output pairs interleaved in some fashion, plus the AC of some scheme for delimiting the input-output pairs, plus the AC of a small program that retrieves an output given the corresponding input. The size of this uncompressed tabular representation will be called 'tabular size' below.
- *state:* State can be considered as an implicit argument to any routines whose behavior is affected.

These comments are only a first attempt at formulating the AC of real-world programs, but the accuracy of this formulation is not crucial to our argument: if the size of an output-only program cannot be objectively determined, the addition of input, state, and interactivity will not simplify things.

There is no algorithm for computing the AC of an arbitrary string. Rephrasing this for our purposes, it means that there is no general algorithm for finding the shortest program with a desired behavior. In general such a shortest program could only be found by examining all possible programs smaller than the string itself. We can imagine a program `program-check` that enumerates all such programs in lexicographic order and runs them to find the shortest program that produces a desired string. Turing's halting theorem says that `program-check` cannot determine if a candidate program is looping or merely taking a long time to finish. Thus, `program-check` will become stuck when it encounters a program that loops.

An asymptotic upper bound to AC can be computed, but is not feasible. We can produce a variation of the previous program called `program-interleave` that interleaves construction and execution of the programs. That is, it runs each program constructed so far for a certain number of time steps, collects results on programs that finish during this time, and then it constructs the next program and begins running it along with any previous programs that are still active. Program `program-interleave` will asymptotically identify smaller programs but there is no way to know if a long-running program is looping or if it will finish and prove to be still smaller representation of the desired string. This approach is infeasible, primarily because the number of programs to be checked is exponential in the program size.

An upper bound on program size (tabular size) can be defined but is not feasible. A trivial upper bound on program size is easy to define — it is simply size of the table or tables describing the program's input-output behavior, as sketched above. This 'tabular size' bound is not feasible however. Consider a simple function that accepts two 32-bit integer arguments and produces an integer result. This function can be represented as an integer-valued table with 2^{64} entries. While small data types such as characters are sometimes processed in a tabular fashion, this example makes it clear that tabular specification becomes infeasible even for small functions involving several integers.

The preceding comments indicate that there is no way to objectively find usable bounds on the size of a new program. The minimal program size cannot be feasibly computed, and the trivial upper bound vastly overestimates the size of a realistic program. In fact, the ratio between the tabular size and the (uncomputable) AC, considered as a function of the tabular size, is known to grow as fast as any computable function.

Corollary: **development time cannot be objectively predicted.**

This is a direct consequence of the fact that program size cannot be objectively predicted. There is some limit on the speed at which a programmer can write code, so any development time estimate that is supposed to be independent of program size will be wrong if the program size turns out to be larger than can be constructed during the estimated time period.

These comments applies to programming problems in general. Is it possible to do better in particular cases?

For example, suppose a company has developed a particular program. If it is asked to write the same program again, it now has an objective estimate of the program size and development time. Is there a middle ground between having no objective estimate and an estimate based on an identical completed project? It would seem that the answer should be yes — one can estimate the size and development time of a new project by comparing it to one's experience of projects that seem similar. Surprisingly, however, judgments of similarity are not enough for a software engineering method or process to make any claims of objectivity. Programming projects require by definition at least a small amount of code that is different from code in previous projects. As argued above, the development time for this new code cannot be objectively estimated.

To illustrate this important point, suppose a programmer says "This new program will be just like a program I wrote previously except that X, and X is a trivial problem." The programmer's assertion may be correct. But it might also turn out that the programmer's assumptions about problem X are wrong and it takes much longer than expected to solve. Solving X may even require a redesign of the rest of the program. Most developers have encountered projects where an apparently trivial subproblem turns out to be more difficult than the major anticipated problems.

We conclude that *development time will remain a matter of the programmer's intuition.*

Productivity

Claim 2: **absolute productivity cannot be objectively determined.**

Consider a statement that N lines of source code were developed in T time, with N/T higher than measured on other projects. This suggests that higher productivity has been achieved. But productivity is relevantly defined as the speed of *solving the problem,* not the speed of developing lines of code. If the N is significantly higher than it needs to be for the particular problem then a high N/T ratio may actually represent low productivity. This is not merely a theoretical possibility: DeMarco and Lister's programming benchmarks empirically showed a 10-to-1 size range among programs written in the same language to the same specification.[8]

We conclude that since there is no feasible way to determine the minimal program size, *the evaluation of absolute productivity is a matter of intuition.*

Proviso: *Relative productivity can be determined.* The relative effectiveness of various software engineering methods can obviously be determined in a comparative experiment in which a particular problem is solved by teams using different methods. This has been attempted in a few cases, but it is believed that the effects of programmer variability are much stronger than those due to development method. With small or few programming teams programmer variability may be overwhelming the method effects. The author is not aware of any studies using a factorial design that would separate programmer and method variability.

Many software methods claim to provide the most benefits at large scales, i.e., over the lifetime of a project that requires the coordination of many people. Given the large cost of software development and the effect of programmer variability, such claims may be too costly to ever verify in an experiment. Many organizations have difficulty finishing even individual software packages and do not have the resources to develop the same package using several different methods. Considering the large and continuing cost of software development, however, the evaluation of software methods on a realistically sized project would be worthwhile despite the enormous cost.

Reliability

Claim 3: **program correctness cannot be objectively determined.**

The reasoning to be used in defending this claim will be introduced by reviewing a well known result in AC, algorithmic incompleteness.

Incompleteness theorem (Chaitin's theorem): *A formal theory with N bits of axioms cannot prove statements of the form '$AC(x) > c$' if c is much greater than N.*

The proof is by contradiction. One makes the reasonable assumption that if a statement $AC(x) > c$ can be proved then it should be possible to extract from the proof the particular x that is used. Then by appending this extraction method to the proof sequence (of $AC \approx N$) one can generate the string x using approximately N bits. But the proof has shown that the AC of x is $AC(x) > c > N$ resulting in contradiction.

The proof is illuminated by recasting the formal system in computational form. A formal system consists of a set of symbols; a grammar for combining the symbols into statements; a set of axioms, or statements that are accepted without proof; and rules of inference for deriving new statements (theorems). A proof is a listing of the sequence of inferences that derive a theorem. It is required that a proof be formally (i.e. mechanically) verifiable. Thus, there is a correspondence between a formal system and a computational system whereby a proof is essentially a string processing computation:[9]

$$\begin{array}{rcl} \text{axioms} & \Longleftrightarrow & \text{program input or initial state} \\ \text{rules of inference} & \Longleftrightarrow & \text{program interpreter} \\ \text{theorem(s)} & \Longleftrightarrow & \text{program output} \\ \text{derivation} & \Longleftrightarrow & \text{computation} \end{array}$$

(This is a purely formal and possibly restrictive notion of proof — see Naur[3] for a discussion. Also note that there are several ways of defining a formal system $\leftrightarrow$ computation correspondence since the division of a computational system into "computer" and "program" can be done in different ways.)

Consider a program that will successively enumerate all possible proofs in length order until it finds a proof of the form $AC(x) > c$, which it will then output. The program and computer will need to encode the axioms and rules of inference, which are N bits by definition. The program will also need to encode the constant c. It is possible to choose a c, such as $2^{2^{2\cdots}}$, that is much larger than N but whose AC is small. Say that the size of the search program including the encoded c is $AC(c)+N$. Append a second algorithmically simple program that extracts and prints the x from the proof found in the search. Let k be the AC of this program, plus, if necessary, the AC of some scheme for delimiting and sequentially running the two programs. We now have a compound program of complexity $AC(c) + N + k$.

If c is picked as $c > AC(c)+N+k$ then either we have a program of $AC < c$ that generates a string that is proved to have $AC > c$ (such a proof is then wrong, so the formal system is unsound) or the program cannot prove statements of the form $AC(x) > c$. Since the details of the formal system were not specified, it is concluded that no formal system can prove that strings are much more complicated than the axioms of the system itself.

Now we will apply this sort of reasoning to the issue of program correctness. A program is incorrect when its behavior differs from the desired behavior. A program would be considered proved correct if it were shown that its behavior matches that of a desired specification.

It is known that the obvious means of accomplishing a correctness proof are not practical; the difficulties involved are instructive, however. Suppose that one wishes to verify the operation of an arithmetic operation in a microprocessor. The operation can be checked against a table constructed using a different processor that is assumed to be correct. If the operation takes two 32-bit arguments and produces a 32-bit result then the required table will have 2^{64} entries, which is clearly impractical. If the operation is checked by directly consulting a second processor rather than a stored table then speed becomes the issue. If both processors are fast and 2^{32} operations can be checked per second then it will take 2^{32} seconds to complete the test of this simple operator.

A better way of verifying program behavior is through examination of its logic rather than an exhaustive examination of its behavior. This is where algorithmic complexity comes in. Consider a proof $P(A, S)$ that the behavior of program A matches a specification S. The specification must appear in the proof in some form, and the specification must be formal in order to participate in the proof. Since the specification fully specifies the behavior of the program, *the AC of the specification must be at least approximately as large as the complexity of a minimal program for the task at hand.* More specifically, given some input it must be possible to determine the desired behavior of the program from the specification. Using this, a small program of AC k (with $k \ll AC(A)$ generally) can be written that, given some input, exhaustively queries the specification until the corresponding output bit pattern (if any) is determined; this bit pattern is then output, thereby simulating the desired program. Formally, $AC(S) + k \geq AC(A)$. We are left with the obvious question: if the specification is approximately of the same order of complexity as the program, how can we know that the specification is correct?

This conclusion has been arrived at previously and without the help of AC. The AC viewpoint strengthens and formalizes the conclusion, however, by making it clear that a previously intuitive quantity (complexity) can be formally defined and expressed in terms of a familiar measure (bits). Whereas intuitively we have the question, “how do we know the specification is correct”, formally we have the assertion *a specification capable of proving program correctness must be at least as complex as the minimal program.* Thus if the program is sufficiently complex that its correctness is not evident, the same may be true of the specification as well.

Proviso: How do formal methods relate to this conclusion?

The term formal method describes a programming strategy in which a high level specification of the program is written in a formal notation.[10] This specification helps (and forces) the developers to think about the program before writing. The specification usually does not fully constrain the eventual program, however, so it does not guarantee correctness.

In some cases the formal specification is sufficiently complete that it can be transformed into a working program, and this transformation is in essence a proof that the program’s behavior matches the specification. In this case the formal specification is essentially a program written in a high-level language so in general we cannot know if the specification is correct. (This does not mean that nothing has been gained however — the specification may be as algorithmically complex as the resulting program but it is presumably easier to produce, understand, or modify).

Conclusions

Some of the issues raised in debates about software engineering will now be briefly discussed from our perspective.

- *Is software engineering a form of engineering?* Software is often compared to other branches of engineering, and solutions to the software crisis are sometimes sought in this comparison. The engineering that “software engineering” seeks to emulate is mature engineering, where the principles for solving a class of problems are codified. The question of whether software engineering can achieve such maturity is debated; this article shows that fundamental quantities such as reliability will remain outside the scope of software engineering.

  In fact some issues of software development resemble problems in the social sciences more than those in engineering. For example, the factors influencing software productivity cannot be determined by axiomatic reasoning or appeal to scientific laws, but it was noted earlier in this article that these factors can potentially be estimated by experiments of the sort common in psychology. Likewise, the characterization of software using code metrics resembles the definition and measurement of psychological characteristics such as intelligence: There is no absolute and objective measure of intelligence, but 'intelligence' can be somewhat circularly defined as the ability measured by an intelligence test. The merits of particular intelligence tests are then debated by considering their correlation with measurable things (e.g. school grades) that are considered to be related to intelligence. Similarly, there is no feasible objective measure of complexity, but 'complexity' can be defined as the property measured by a proposed code metric. The particular metric chosen must then be justified; this can be done by demonstrating a correlation between the measured property and an expected correlate of complexity such as measured development time.

- *We should be honest, or we will be forced to be honest.* It has been argued that if the software industry's credibility continues to erode software development will become subject to unpleasant external regulation. Voas et. al.[11] write,

  > "…software developers are living in a liability grace period: The courts have not demanded that software practitioners meet the absolute standards of professionalism required of other engineering disciplines."

  Credibility will not be achieved by continuing to promise that software quality and predictability are just around the corner (as soon as we utter the latest buzzword enough times). Instead, the software industry should promote the idea that software development has intrinsic uncertainties and risks. With the advent of computerized vehicle control systems, citizen databases, and so on, the public discussion and honest assessment of software risks and limitations is increasingly important.

The points made in this paper will seem obvious to many, but pessimistic to those who advertise "defect-free development" or suggest that development schedules can be objectively predicted from historical data. These points must also seem pessimistic to those who ask questions such as "Why does software cost so much?" As DeMarco persuasively argues,[8] the premise of this and similar questions should itself be questioned — there is no reason to assume that software development should be much more predictable, productive, or reliable than it has been. On the contrary, realistic consideration of the essential limitations of our field may engender a respect for what has been accomplished and an understanding of the continuing risks, whereas disregard of these limitations is a recipe for continued disappointment.

References

[1] Scholtz et. al., "Object-oriented Programming: the promise and the reality," *Software Practitioner,* Jan. 1994, pp. 4-7.

[2] R. L. Glass, "Object-Orientation is Least Successful Technology" *Software Practitioner,* Jan. 1994, p. 1-3.

[3] P. Naur, *Knowing and the Mystique of Logic and Rules,* Kluwer Academic, Dordrecht, 1995.

[4] R. Gabriel, *Patterns of Software*, Oxford, 1996, p. 128.

[5] R. L. Glass, "What are the Realities of Software Productivity/Quality Improvements," *Software Practitioner,* November 1995, pp. 4-9.

[6] F. P. Brooks, Jr., *The Mythical Man-Month* (anniversary edition), Addison-Wesley, Reading Mass., 1995, p. 231.

[7] C. F. Kemerer, "An Empirical Validation of Software Cost Estimation Models," *Communications ACM*, Vol. 30, No. 5, May 1987, pp. 416-429.

[8] T. DeMarco, *Why Does Software Cost So Much? and other Puzzles of the Information Age,* Dorset, New York, 1995.

[9] K. Svozil, *Randomness and Undecidability in Physics,* World Scientific, Singapore, 1993, pp. 30-35.

[10] A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, Sept. 1990, pp. 11-19.

[11] J. Voas, G. McGraw, L. Kassab and L. Voas, "A 'Crystal Ball' for Software Liability," *IEEE Computer*, June 1997, pp. 29-36.

Sidebar: Algorithmic Complexity

Algorithmic complexity (AC) was defined in the 1960s by Kolmogorov, Chaitin and Solomonoff and has been developed more recently by Chaitin and others; it is alternately known as algorithmic information theory, Kolmogorov complexity, and KCS complexity. AC is relevant to and related to a number of fields such as compression, machine learning, information theory, the philosophy of science, and metamathematics. The abbreviation 'AC' will be used to refer both to the field and to the algorithmic complexity of a specific object such as $AC(x)$.

AC defines the complexity of a digital object as the length of the shortest computer program that produces that object. This definition formalizes an intuitive notion of complexity. Consider the three patterns:

```
11111111111111...
12341234123412...
30547430729732...
```

These strings may be of the same length but the first two strings appears to be simpler than the third. This subjective ranking is reflected in the length of the programs needed to produce these strings. For the first string this program is a few bytes in length, e.g.,

```
for i:=1 to n print('1');
```

The program for the second string is slightly longer since it will contain either nested loops or the literal `'1234'`. If there is no obvious pattern to the third string, the shortest program to produce it is the program that includes the whole string as literal data and prints it — the string is incompressible or algorithmically random. Kolmogorov defined randomness in this way, and in fact algorithmically random strings satisfy the standard statistical tests for randomness. If the strings are long the length of the first program will be dominated by the cost of representing the string length `n`; thus its length (complexity) will be approximately $O(log(n))$ or less (if n is a constant then it may be possible to compress it beyond $log(n)$. Some large n are very regular, e.g. in a decimal representation 100,000 is more regular and compressible than 997,898 although both are approximately the same size). The length/complexity of the program for the third string is $O(n)$.

Ideally the complexity of an object should be a property only of the the object itself, but the choice of computer and programming language affects program lengths. How can the complexity of an object be defined independently of the choice of computer? Computers can simulate other computers, so consider an interpreter (virtual machine) that emulates another computer or computer language on the computer of your choice. This interpreter has some fixed size such as 200K bytes. When the objects being represented are very large this constant size becomes insignificant. AC is thus independent of the choice of computer in the limit of very large objects.

The following is an example of AC reasoning: *Most strings are not compressible.* There are 2^n different binary strings that are n-bits in length. Also, there are potentially $\sum_1^{n-1} 2^k = 2^n - 2 \approx 2^n$ different program texts of length less than n bits. Most of these texts do not contain meaningful programs, but included among them are all possible programs that can be used to compress n-bit strings. To compress a n-bit string by m bits we seek an $(n-m)$-bit program that when run will print the desired string. By comparing the number of n-bit strings (2^n) and the number of programs of size $n-m$ bits or less ($\approx 2^{n-m}$) we see there are not enough programs to go around — fewer than one in 1000 (2^{-m} with $m = 10$) strings can be compressed by as much as 10 bits, for example, even with the implausible assumption that all possible program texts of $n-m$ bits or less in size are in fact correct compression programs. As seen in this example, AC results have a different flavor than statements in other branches of mathematics. AC often describes the properties or limitations of a class of objects without providing an explicit means of constructing the objects in question.

Other AC results of interest include a relationship between AC and entropy, a formalization of Occam's razor, and the demonstration of a family of mathematical propositions whose truth value is algorithmically random (the last result might suggest that "God does play dice" in mathematics). An intuitive presentation of AC and its role in metamathematics is found in Rucker. The excellent textbook by Li and Vitányi is a self-contained guide to the field and should be consulted concerning aspects of the formal definition of AC that are not essential to our presentation.

R. Rucker, *Mind Tools: the Five Levels of Mathematical Reality*, Houghton Mifflin, Boston, 1987.

M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and its Applications* Second Ed., Springer-Verlag, New York, 1997.

COMMENT FOR REVIEWERS. This section is not intended for publication.

Quotes and Claims

The authors of various development methods and commercial products are understandably optimistic about the potential of their products and methods. It is not the purpose of this article to criticize specific methods, so the paper does not cite particular claims of the 'defect-free' movement (for example) or quote statements such as

> "[In a mature organization] There is an objective quantitative basis for judging product quality and analyzing problems with the product and process. Schedules and budgets are based on historical performance and are realistic"

A survey of the software engineering shelf of your local bookstore will show similarly unrealistic claims concerning the impact of various methodologies and processes on software development times, productivity, and quality (as well as, of course, some well reasoned and carefully considered viewpoints).

Additional References

The following references are omitted due to the 12-reference limit. Internal Revenue Service software difficulties are described in Richard Stengel, ''An Overtaxed IRS,'' *Time*, April 7 1997 pp. 59-62. The child support-tracking software was reported in H. Jordan, ''Computer Snafu May Cost State Millions'', *San Jose Mercury News,* April 1997. Contact the author for additional references.