WHY ENGINEERS SHOULD CONSIDER FORMAL METHODS

C. Michael Holloway

NASA Langley Research Center
Mail Stop 130 / 1 South Wright Street
Hampton, Virginia 23681-0001
E-mail: c.m.holloway@larc.nasa.gov

ABSTRACT

This paper presents a logical analysis of a typical argument favoring the use of formal methods for software development, and suggests an alternative argument that is simpler and stronger than the typical one.

INTRODUCTION

For more than twenty-five years, some people have touted formal methods as the best means available for developing safe and reliable digital systems. To many within the research community, the efficacy — or more accurately, the necessity — of formal methods is now accepted as proved. One well-known researcher expressed this attitude succinctly when he wrote concerning software engineering: "It is clear to all the best minds in the field that a more mathematical approach is needed for software to progress much." [1]

Despite this bold assertion, the attitude of many of the best minds among practicing engineers has been quite different, with far more rejecting formal methods than embracing them. Although the situation has changed some within the last several years, especially within the hardware design community [2], the acceptance and regular use of formal methods is still far less than proponents want. Formal methods researchers and practitioners have tried to analyze the causes of this lack of acceptance in opinion pieces [3, 4], case studies [5], and small experiments [6]. Suggested causes include lack of adequate tools, lack of mathematical sophistication in developers, incompatibility with current techniques, high costs, and over- selling by advocates.

Despite reaching different conclusions, all of these attempts (my own included [7]) have, by and large, addressed the issue in a similar way. They have each attempted to determine why engineers are not routinely using existing formal techniques and tools. The shared assumption seems to be that the idea of formal methods has been proved to be good; the acceptance problem lies in the details, not the idea. That formal methods advocates share this assumption is not surprising. Nevertheless, the reluctance of many engineers to use, or support the development of, any formal method or tool suggests another possibility: perhaps the acceptance problem lies not in the details, but in the way the idea has been communicated to engineers. This paper presents the preliminary results of my effort to investigate this possibility.

The structure of the paper is as follows. The next section states the specific question I considered. This is followed by an example of a typical rationale for formal methods. A logical analysis of this rationale is then given, followed by a revised rationale designed to correct the flaws in the original one. Brief concluding remarks complete the body of the paper. An appendix provides an overview of the basic principles of logical reasoning that are used in this paper. Readers unfamiliar with the definitions of terms such as proposition, deductive argument, and inductive argument should read this appendix before the next section.

THE QUESTION

In as simple and abstract terms as possible, and ignoring possible subtleties, the problem we are considering can be stated as follows. Group one makes an assertion and provides arguments they believe prove this assertion. Group two, by their actions if not necessarily their words, denies the assertion. Group one's assertion is either true or false. If the assertion is false, then group two is justified in denying it.

If the assertion is true, then group two may or may not be justified in denying it. They are justified in denying the assertion if the arguments supplied by group one are insufficient to prove the truth of the assertion. They are not justified in denying the assertion if the arguments supplied by group one are sufficient. Note that, by definition, if the assertion is false, group one's arguments supporting it cannot be sufficient. Thus, to determine whether group two's denial of the assertion is justified, we need only consider the sufficiency of the arguments supplied by group one.

In our particular case, group one consists of formal methods advocates. Group two consists of industry engineers. The assertion is that engineers of computer systems should use appropriate formal methods. To simplify our discussion, we will restrict ourselves to computer software, recognizing that the line between software and hardware is becoming increasingly blurred. Thus, the question is: Do the arguments supplied by formal methods advocates adequately support the assertion that software engineers should use appropriate formal methods?

TYPICAL RATIONALE

To begin to answer this question, let us consider a typical rationale for formal methods. The rationale given here is based on the arguments given previously by NASA Langley formal methods team members (myself included) [8], augmented by arguments from other Langley-sponsored work [9, 10].

Software is notorious for being late in delivery and unpredictable and unreliable in operation. According to a 1994 article by Wayt Gibbs, "Studies have shown that for every six new large-scale software systems that are put into operation, two others are cancelled. The average software development project overshoots its schedule by half; larger projects generally do worse. And three quarters of all large systems are operating failures that either do not function as intended or are not used at all." [11]

When compared to other engineering disciplines, software engineering does not come out looking good. But this should not be surprising, because in at least two respects, software is different from the physical objects, materials, and systems with which traditional engineers work.

First, in physical systems smooth changes in inputs usually produce smooth changes in outputs. That is, most physical systems are continuous. This allows the behavior of the system to be determined by testing only certain inputs, and using extrapolation and interpolation to determine the behaviors for untested inputs.

Software systems are, by their very nature, discontinuous. A small change in input may change the outcomes at several decision points within the software, causing very different execution paths and major changes in output behavior. As a result, using extrapolation and interpolation to estimate output behaviors for untested inputs is risky at best, and exceedingly dangerous at worst.

Software differs from physical systems in another way: its complexity. Much of the functionality of modern systems is provided by software; therefore, much of the complexity of these systems is expressed in the software, also. The greater the complexity, the more likely design flaws — flaws in the intellectual construction of the system that cause it to do the wrong thing under some conditions — are to occur. Design flaws are the only way that software can go wrong; software does not wear out like physical components. Thus, to ensure that a software system does what it is intended to do, design flaws must be handled in some way.

Many different approaches to handling design flaws have been proposed. All of these may be grouped in one of the following three categories: testing, design diversity, or fault avoidance.

The discontinuity of software poses problems for testing-based approaches. For systems with low reliability requirements, testing for long enough to show statistically that the system meets its requirements may be possible. But for high integrity software systems, such testing would require much more time than is feasible. For example, to measure a 10^{-9} probability of failure for a 1 hour mission, one must test for more than 10^{9} hours (114,000 years) [12]. Thus, for such systems, testing-based approaches are inadequate.

The basic idea behind approaches of the design diversity type is to use separate teams to produce multiple versions of the software. The hope is that the design flaws will manifest errors independently or nearly so, and that voters can be used at run-time to mask the effect of those flaws. If the independence assumption is valid, ultrareliable-level estimates of system reliability can be obtained even with failure rates for individual versions of 10^{-4}/hour. However, the independence assumption does not appear to be valid. In several experiments for low reliability software, the assumption was rejected at the 99% confidence level [13, 14]. Furthermore, the independence assumption cannot be validated for high reliability software because of the exorbitant test times required [12]. As a result, design diversity is inadequate, also.

Because design flaws cannot be handled adequately by approaches based on either testing or design diversity, fault avoidance techniques offer the best hope. Of possible fault avoidance techniques, formal methods are the most rigorous; therefore, they are the most promising. Hence, to phrase the conclusion in the language used earlier, software engineers should use appropriate formal methods.

CRITIQUE

We want to determine if this argument provides sufficient justification for its conclusion. To do this, we can examine the structure of the argument by stripping away the verbiage. This will leave us with only the essential propositions and the relationships between them. Doing this yields the following (for reference, each proposition is given a label).

Software is bad (P1). Software differs from physical systems in at least two ways (P2): software is discontinuous (P3), and software is complex (P4). Software is complex (P4), and complexity results in design flaws (P5); therefore, software has design flaws (P6). Design flaws must be handled (P7). The three ways to handle design flaws are testing, design diversity, and fault avoidance (P8). Because software is discontinuous (P3), testing is inadequate (P9). Also, because software is discontinuous (P3), design diversity is inadequate (P10). Because there are only three ways to handle design flaws (P8), and the other two are inadequate (P9, P10), fault avoidance must be used to handle design flaws (P11). Because formal methods are the most rigorous fault avoidance method (P12), and the greater the rigor, the more promising the method (P13), formal methods are the most promising fault avoidance method (P14). Because software has design flaws (P6), and design flaws must be handled (P7), and fault avoidance methods must be used to handle design flaws (P11), and formal methods are the most promising of these methods (P14), software engineers should use appropriate formal methods (P15).

Figure 1 gives a graphical depiction of the structure of the argument.

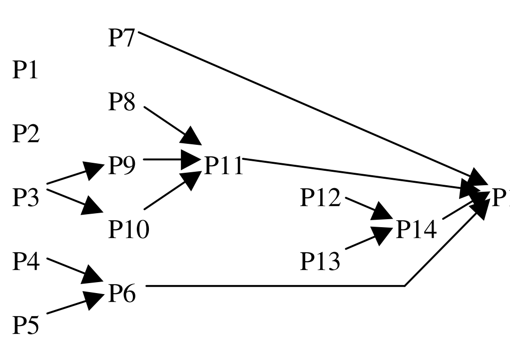

Figure 1: Structure of Typical Argument

After examining this structure, we can make the following observations:

- The argument is fairly complicated.
- The following two propositions play no part in establishing the conclusion:
  1. Software is bad (P1)
  2. Software differs from physical systems in at least two ways (P2)
- The conclusion depends immediately on the following propositions:
  1. Software has design flaws (P6)
  2. Design flaws must be handled (P7),
  3. Fault avoidance methods must be used to handle design flaws (P11)
  4. Formal methods are the most promising of these methods (P14).

We will now consider the implications of each of these observations.

Complexity

A complex argument is not necessarily a bad argument. Some conclusions can only be reached by long, complicated arguments. In such cases, complexity is essential; however, one should keep in mind that most people react to a complicated argument in one of two ways. Some reject it out of hand. Being unwilling to invest the effort needed to analyze the argument carefully, these people are also unwilling to believe that which they do not understand. Others accept a complicated argument without question, assuming that anything so complicated must be true.

In trying to make the case for formal methods, we certainly do not want to give the former group cause to reject our argument out of hand. Nor do we want the latter group to accept our argument unthinkingly; those who do so are likely to give up when practical difficulties arise. If it is possible to construct a simpler argument, we should do so. To paraphrase C.A.R. Hoare's comment on software design [15], there are two ways of constructing most arguments. One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies. In the revised rationale that I present later, I opt for the first approach.

Unnecessary Propositions

From a strictly logical point of view, unnecessary propositions are just that: unnecessary. Taking this view, however, ignores the fact that other reasons besides logical necessity may exist for including certain propositions in an argument. For example, beginning the argument for formal methods with a discussion of the sad state of current software development practices may well serve to encourage an audience to listen closely to what follows. On the other hand, if someone is unconvinced that the state of practice is as bad as is claimed, that person may be less likely to listen to what follows.

Logically, propositions P1 and P2 do not need to be in the argument. Rhetorically, cases can be made both for and against including one or both of them. In my revised rationale, I leave them out.

Immediate Dependency

It is upon the truth or falsity of the propositions on which the conclusion immediately depends that the sufficiency of this argument rests. If it is certain that software has design flaws, design flaws must be handled, fault avoidance methods must be used to handle design flaws, and formal methods are the most promising of these methods, then it is equally certain that formal methods will benefit software engineers. Let us look at each proposition and see how certain it is.

Does software have design flaws? Anyone who has ever spent more than a few minutes in front of a computer knows that it does. We do not even need the two propositions used in the argument as support. This proposition is indisputably true.

Must design flaws be handled? In computer games, VCRs, and personal entertainment systems, failing to handle design flaws might not have serious consequences. In avionics, reactor control, and anti-lock brakes, failing to handle design flaws might have life threatening consequences. Thus, for the most important types of computer systems, this proposition is also indisputably true.

Must fault avoidance techniques be used to handle design flaws? In the given rationale, this proposition is claimed to follow from three other propositions: (P8) the three ways to handle design flaws are testing, design diversity, and fault avoidance, (P9) testing is inadequate, and (P10) design diversity is inadequate.

This approach seems to me to be unnecessary for, and potentially harmful to, the argument. It is unnecessary because not even the most ardent supporters of testing or design diversity argue that fault avoidance techniques should be abandoned. It is potentially harmful because some people who are unconvinced by the arguments against testing or design diversity might not listen to the rest of the argument. The need for fault avoidance techniques is as self-evidently clear as the previous two propositions we have considered. This proposition could be established much more easily than is done here.

So far, the three important propositions we have examined are true. If the fourth proposition is true, then the rationale will turn out to be a sound deductive argument for its conclusion.

Are formal methods the most promising fault avoidance method? The rationale claims they are because formal methods are the most rigorous fault avoidance method (P12), and the greater the rigor, the more promising the method (P13).

Alas, this is begging the question. All the claimed benefits from formal methods are derived from the rigor they enforce. If rigor is promising, then formal methods are promising. But the argument does not prove that rigor is promising, it simply asserts it: P14 and P13 assert the same thing, using different words.

Thus, although three of the four essential propositions have been shown to be true, the fourth has not. Only those who already believe that rigor is good should find the given rationale sufficient. Everyone else should remain unconvinced.

REVISED RATIONALE

The typical rationale failed, but it came close. It could be completed by simply establishing the truth of P14 without begging the question; however, doing so would result in a rationale that still retains the unnecessary complexity noted in the previous section. So, instead of attempting a repair job, let us develop a different rationale all together.

Using the same style as used in the previous section, the revised rationale is as follows. Notice that the original fifteen propositions have been replaced by only five, and that the structure is so simple as to not need a graphical representation.

Software engineers strive to be true engineers (Q1); true engineers use appropriate mathematics (Q2); therefore, software engineers should use appropriate mathematics (Q3). Thus, given that formal methods is the mathematics of software (Q4), software engineers should use appropriate formal methods (Q5).

This is a valid deductive argument, in which the truth of the conclusion rests upon the truth of two premises: Q3 and Q4. In turn, the truth of Q3 rests upon the truth of two other premises: Q1 and Q2 (and unstated premises that relate striving to be an engineer with doing what engineers do). To show that the conclusion is true, we need only show that Q1, Q2, and Q4 are each true. This is a simple task.

Because formal methods are defined as the mathematics of computer software and hardware systems [16], Q4 is true by definition.

Hundreds, perhaps thousands, of references could be cited from the late 1960's (when the term "software engineering" was coined) to the present to establish the truth of Q1. For example, Roger Pressman writes [17]: "An early definition of software engineering was proposed by Fritz Bauer at the first major conference dedicated to the subject: *The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.* Although many more comprehensive definitions have been proposed, all reinforce the importance of engineering discipline in software development."

Similarly, the truth of Q2 can be established by myriad citations. Again, one will suffice: "Professional engineers are expected to use discipline, science, and mathematics to assure that their products are reliable and robust." [18]

We have proven Q1, Q2, and Q4 to be true. Q3 follows by deduction from Q1 and Q2. Q5 follows by deduction from Q3 and Q4. Thus, software engineers should use appropriate formal methods. Please note that the word "appropriate" is important. Using inaccurate or incomplete mathematics can cause disasters in traditional engineering [19]. There is no reason to suspect that the same cannot happen with formal methods.

CONCLUDING REMARKS

In this paper, I presented an analysis of a typical rationale used to convince engineers of the potential usefulness of formal methods. This analysis revealed that the typical rationale is complicated, but fails to establish the truth of an essential proposition. As a result, I presented a simple revised rationale, which I believe shows conclusively why engineers should consider formal methods. The ideas in this revised rationale are not original. Rushby includes the basic concepts, although his other detailed discussions tend to distract from them [9, 10]; and Parnas states them succinctly [20]. The contribution of this paper is in presenting the ideas in the context of an analysis of other approaches, and in a forum likely to be populated by engineers.

I believe that engineers will consider formal methods, and that, as one industry engineer says, "formality will eventually become the norm in software development." [21] This does not mean that all current formal methods tools and techniques are ready for immediate use. Unfortunately many current formal methods tools and techniques more closely resemble the Wright Flyer than the 777, but with the diligent, cooperative work of mathematicians, logicians, and engineers, researchers and practitioners, the situation can change quickly. I believe it will.

APPENDIX: LOGICAL REASONING

This appendix summarizes the basic ideas of logical reasoning [22]. The reader familiar with these ideas may skip it. The reader interested in more information should consult [23, 24].

Propositions, Premises, and Conclusions

A logical argument consists of a series of statements. These statements are not just any old statements; each one must be either true or false. Such statements are called *propositions*. Commands, questions, and requests are not propositions, and thus are not formally part of a logical argument.

To construct an argument, propositions are grouped in such a way that one of them is asserted to follow from the others. The proposition that is affirmed on the basis of the others is called the *conclusion*; the other propositions in the argument are called the *premises*. In the following example, the first two propositions are the premises, and the third proposition is the conclusion: All cats are clever. Dixie is a cat. Therefore, Dixie is clever.

Of course, most arguments in real life are not written so simply as this example, which means that identifying the premises and conclusions can be more difficult. Not only do real life arguments frequently contain extraneous information, all of the premises and conclusions are often not stated explicitly. The technical term for an argument in which only parts are stated is an *enthymeme*.

Because of the vast amount of knowledge that is assumed in almost any statement we make, most real-world arguments are, in fact, almost always enthymemes of some type. Those given in the body of the text are, too. For example, in my revised rationale, I assert that *software engineers should use appropriate mathematics* follows from *software engineers strive to be true engineers* and *true engineers use appropriate mathematics*. Strictly speaking, additional premises are needed to define the meanings of, at least, *strive* and *should*.

Validity and Soundness

In a *valid* argument, if all of the premises are true, then the conclusion must necessarily be true. An argument is *sound* if it is valid and all of its premises are known to be true. A sound argument proves its conclusion. That is, if an argument is sound, then we have no choice — short of abandoning reason — but to believe its conclusion.

An *unsound* argument is a valid argument with at least one false premise. An *invalid* argument is one in which all of the premises can be true, but the conclusion still be false. Neither unsound nor invalid arguments tell us anything about whether their conclusion is true or false.

The following is an example of an argument form that is always valid:

- Premise 1: If P, then Q
- Premise 2: P
- Conclusion: Therefore, Q

The first premise asserts nothing about the truth or falsity of either P or Q alone, but it does say that if P is true, then Q will also be true. The second premise asserts that P is in fact true. From these two premises, concluding that Q is true is always valid. This particular form of argument is called *modus ponens* (from the Latin *modus*, meaning "method", and *ponere*, meaning "to affirm"). By substituting various propositions for P and Q, many valid arguments can be created. Whether such arguments are sound depends on the truthfulness of the chosen P and Q and on the truthfulness of "If P, then Q."

For example, if we let P be "I work for NASA," and Q be "I am a civil servant", we get the following sound argument: If I work for NASA, then I am a civil servant. I work for NASA. Therefore I am a civil servant.

On the other hand, if we let P be "I work for NASA," and Q be "I am involved in the space program", the resulting argument is valid, but unsound.

Fallacies

An invalid argument will contain either a formal fallacy or an informal fallacy. A *formal fallacy* is one in which there is something incorrect about the form of the argument. A common example, which is a perversion of modus ponens, is known as *affirming the consequent*. It looks like this:

- Premise 1: If P, then Q
- Premise 2: Q
- Conclusion: Therefore, P

Here is an example: If Boeing built it, the plane is a jet; the plane is a jet; therefore, Boeing built it.

An *informal fallacy* is one in which something other than the form is wrong. There are many types of informal fallacies. The only one important to us here is called *petitio principii* in Latin, and begging the question in English. In an argument that commits this fallacy, one of the premises from which the conclusion is deduced is the conclusion itself, usually in different words. Such an argument is valid, because P does imply P, but useless. Here is an example: Volleyball is more fun to play than baseball, because baseball is not as fun to play as volleyball. Of course, in real life, arguments that beg the question tend to do so more cleverly than that.

Inductive Arguments

The discussion so far has been about deductive arguments. *Inductive arguments* are different. Rather than establishing the truth of a conclusion with certainty, an inductive argument only establishes the truth of a conclusion with probability. We do not speak of the validity or soundness of an inductive argument; we speak of its *strength*. A strong inductive argument has high probability that its conclusion is true; a weak inductive argument has low probability that its conclusion is true. Strong arguments are often said to be compelling or convincing.

Here are three examples of strong inductive arguments:

- Greg Maddux is pitching today; therefore, the Braves will win the game.
- Children who study Latin score higher on English vocabulary tests than do children who do not study Latin; therefore, studying Latin improves a child's vocabulary.
- Many people who spend a lot of time in the sun get skin cancer; therefore, if you spend a lot of time in the sun, you will get skin cancer.

Each of these is an inductive argument because its premises do not guarantee the truth of its conclusion. Greg Maddux occasionally loses a game. Factors other than studying Latin might account for the differences in vocabulary. Not everyone who spends a lot of time in the sun gets skin cancer.

Each of these is a strong inductive argument, because its premises make the probability high that its conclusion is true. Greg Maddux does not lose often. Many English words come from Latin. A sun worshipper's probability of getting skin cancer is high.

Strictly speaking, one ought never use the term *prove* in connection with inductive arguments; even the strongest possible inductive argument does not prove anything. Nevertheless, the term is often used in common speech. For example, only a particularly petulant person is likely to object to someone saying, "Studies have proven that prolonged exposure to the sun increases one's chances of getting skin cancer."

REFERENCES

[1] Betrand Meyer. From Process to Product: Where is Software Headed? *IEEE Computer*, 28(8):23, August 1995.

[2] David Dill and John Rushby. Acceptance of Formal Methods: Lessons from Hardware Design. *IEEE Computer*, 29(4):23-24, April 1996.

[3] Hossein Saiedian. An Invitation to Formal Methods. *IEEE Computer*, pages 16-30, April 1996.

[4] Betrand Meyer. The Next Software Breakthrough. *IEEE Computer*, 30(7):113-114, July 1997.

[5] Susan Gerhart Dan Craigen and Ted Ralston. Formal Methods Technology Transfer: Impediments and Innovation. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 17, pages 399-419. Prentice Hall, Great Britain, 1995.

[6] John C. Knight, Colleen L. DeJong, Matthew S. Gibble, and Luis G. Nakano. Why Are Formal Methods Not Used More Widely? In *Lfm97: The Fourth NASA Langley Formal Methods Workshop*, pages 1-12, September 1997. NASA Conference Publication 3356.

[7] C. Michael Holloway and Ricky W. Butler. Impediments to Industrial Use of Formal Methods. *IEEE Computer*, 29(4):25-26, April 1996.

[8] Ricky W. Butler, James L. Caldwell, Victor A. Carreno, C. Michael Holloway, Paul S. Miner, and Ben L. Di Vito. NASA Langley's Research and Technology Transfer Program in Formal Methods. In *Tenth Annual Conference on Computer Assurance (COMPASS 95),* Gaithersburg, MD, June 1995.

[9] John Rushby. Formal Methods and Their Role in Digital Systems Validation for Airborne Systems. NASA Contractor Report 4673, August 1995.

[10] John Rushby. Formal Methods and Digital Systems Validation for Airborne Systems. NASA Contractor Report 4551, December 1993.

[11] W. Wayt Gibbs. Software's Chronic Crisis. *Scientific American*, pages 86-95, September 1994.

[12] Ricky W. Butler and George B. Finelli. The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Transactions on Software Engineering*, 19(1):3-12, January 1993.

[13] John C. Knight and Nancy G. Leveson. An Experimental Evaluation of the Assumptions of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, SE-12(1):96-109, January 1986.

[14] John C. Knight and Nancy. G. Leveson. A Reply To the Criticisms Of The Knight & Leveson Experiment. *ACM SIGSOFT Software Engineering Notes*, January 1990.

[15] Alan M. Davis. *201 Principles of Software Development*. McGraw-Hill, New York, 1995. Quoted on page 80.

[16] C. Neville Dean and Michael G. Hinchey (editors). *Teaching and Learning Formal Methods*. Academic Press International Series in Formal Methods. Academic Press, London, 1996.

[17] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Book Company, New York, 2nd edition, 1987.

[18] David Lorge Parnas. Teaching Programming as Engineering. In *Teaching and Learning Formal Methods*, Academic Press International Series in Formal Methods, pages 43-55. Academic Press, London, 1996.

[19] For an example, see pages 285-308 of Henry Petroski. *Engineers of Dreams: Great Bridge Builders and the Spanning of America*. Alfred A. Knopf, New York, 1995.

[20] David Lorge Parnas. Mathematical Methods: What We Need and Don't Need. *IEEE Computer*, 29(4):28-29, April 1996.

[21] James M. Sutton. Plotting the Escape from the Tower: A Formalist's Practicality Primer. In *Lfm97: The Fourth NASA Langley Formal Methods Workshop*, pages 13-20, September 1997. NASA Conference Publication 3356.

[22] C. Michael Holloway. Necessary Consequence. *Calvary Herald*, 1993-1997. Calvary Reformed Presbyterian Church, Hampton, Virginia. The discussion in the appendix is adopted from various installments of this column.

[23] Gordon H. Clark. *Logic*. Trinity Foundation, Jefferson, MD, 1998.

[24] Irving M. Copi and Carl Cohen. *Introduction to Logic*. Macmillan Publishing Company, New York, 9th edition, 1994.

BIOGRAPHICAL SKETCH

C. Michael Holloway is a research engineer at the NASA Langley Research Center in Hampton, Virginia. He has been a member of the NASA Langley formal methods team since 1992, and is the creator and maintainer of the team's World-Wide Web pages. His professional interests include programming language theory and high integrity software.

His personal interests include theology, history, education, and volleyball. Mr. Holloway was graduated from the School of Engineering and Applied Science at the University of Virginia with a B.S. in Computer Science in 1983.