REALLY RETHINKING 'FORMAL METHODS'

David Lorge Parnas, *Middle Road Software*

We must question the assumptions underlying the well-known current formal software development methods to see why they have not been widely adopted and what should be changed.

The theme of the feature articles in *Computer*'s September 2009 issue was "Rethinking Formal Methods." Rethinking this subject is certainly long overdue.

It has been more than 40 years since the late Robert Floyd showed us how to "assign meaning to programs" and demonstrated how we could verify that programs will do what they are intended to do.[1] It has been at least 35 years since I first heard Jean-Raymond Abrial present the ideas that were the basis of Z and its many dialects. The Vienna Development Method (VDM) community began its work about the same time. Even second- and third-generation formal methods show signs of age.

Since 1967, there have been numerous "revolutions" on the hardware side and amazing improvements in man-machine interfaces. The computer systems on my desk today were unimaginable when Floyd wrote that article. Unfortunately, there hasn't been comparable progress in formal methods. There have been new languages and new logics, but the program design errors we saw in 1967 can still be found in today's software. Applications of formal methods to industrial practice remain such exceptions that they confirm that the use of formal methods is not common practice.

We must question the assumptions underlying current formal methods to see what needs to be changed. The articles published in *Computer*'s September issue did not do that; they presented minor variations of ideas that their authors, and other researchers, have advocated for years.

CLAIMS OF PROGRESS AND INDUSTRIAL ADOPTION

The computer science research literature reveals that "formal methods for software development" are a popular research area. Variants of these approaches are frequently discussed and debated at conferences and in journals.

Funding agencies often require that larger research-funded projects include some cooperation with industrial organizations and demonstrate the practicality of an approach on "real" examples. When authors report such efforts, they state that they are successful. Paradoxically, such success stories reveal the failure of industry to adopt formal methods as standard procedures; if using these methods was routine, papers describing successful use would not be published.

 Published by the IEEE Computer Society 0018-9162/10/$26.00 © 2010 IEEE

Industry is so plagued by errors and high maintenance costs that it would use any method it thought would help; it chooses not to use methods such as Z or VDM.

Reports of successful industrial adoption do not always stand up to scrutiny. Sometimes, the authors are just playing with words. For example, the technique of placing debugging statements in code, taught to me in 1959, has recently been trumpeted as "industrial use of assertions."

In other cases, close scrutiny reveals truly heroic efforts with very complex formal models but little evidence that the actual code is correct. Often, these efforts do not lead to repeat use or broader adoption of the method. Development organizations that routinely use these methods for actual products are rare.

Some of the reported success may be attributable to having two people looking hard at the problem and the code. Thirty years ago, in a paper that is still worth reading today, H.S. Elovitz described an experiment in which a program in a second programming language was used in the way that formal method advocates suggest that their notations be used.[2] One programming language was called the *specification language*; the other was the *implementation language*. One programmer wrote the "specification" and gave it to another, who translated it to the implementation language. The specifier reviewed the translation. The error rate was reduced, and this technique (an earlier version of pair programming) was considered successful. This effect alone could explain the few successes in formal methods application. Reports that formal methods are ready for industrial use must be taken with a grain of salt; if they were ready, their use would be widespread.

THREE ALARMING GAPS

The past 40 years have seen some negative developments in the software field.

Gap between research and practice

The gap between formal methods research and practical software development is much larger today than it was when Floyd wrote his paper. Floyd had been a successful, innovative, and productive programmer. His article clearly reveals the connection between mathematical expressions and the programmer's product.

Today, many research papers are written as if the mathematics were all that matters. They do not show how to relate the formal models and results to the actual code on real machines. Further, they offer no way to deal with the complexity of software systems.

On the other side, most software developers perceive formal methods as useless theory that has no connection with what they do. There is no quicker way to lose the attention of a room full of programmers than to show them a mathematical formula. Developers see the need for improvement and will try almost any new method—provided that it does not look like mathematics.

Gap between software development and older engineering disciplines

There is also a disturbing gap between software development and traditional engineering disciplines. Software developers often identify themselves as engineers, but their education and way of working are not at all like those of traditional engineers. Engineering programs teach basic science, applicable mathematics, and how to apply mathematics to predict the behavior of products. Most computer science departments teach technology (which is often of fleeting value in our rapidly changing field) and abstract mathematics that the students do not learn to apply.

There is a disturbing gap between software development and traditional engineering disciplines.

Twenty years ago, I heard a wise and experienced top-level manager complain that his software developers and his other engineers spoke such different "languages" and thought in such different ways that it was difficult for them to work together. That gap is still with us.

Gap between computer science and classical mathematics

A more unexpected development is the separation between theoretical computer scientists and classical mathematics. Many computer science programs present their students with a very narrow slice of mathematics, usually stressing recent developments and "pure" mathematics over older work and what is called "applied mathematics." Mathematics has evolved slowly and contains many mature and general concepts that can be used when describing and analyzing computer systems.

As a result of the "split" between computer science and mathematics departments, formal methods often use the notation of mathematics but do not take advantage of potentially useful mature concepts. It is easy to find situations in which a computer science researcher might invent an approach in which a classically trained mathematician would recognize that it was possible to use an older (and simpler) concept to solve the problem.

An insight-provoking illustration of this is the contrast between the approaches of two people at the same institution, mathematician N.G. de Bruijn and computer scientist E.W. Dijkstra, to programming semantics and verification. De Bruijn applied the classical concept of relations, while

Dijkstra invented his "predicate transformers." In my work, I have found the relational approach far more convenient and easier to use.

BEAUTY AND THE BEASTS

Most articles about the use of mathematics in software development contain two distinct messages.

- A general message that reminds us of the ubiquity of faults in software and argues that the use of mathematical notation and reasoning can ameliorate the situation.

We need to question, and be prepared to discard, most of the methods that we have been discussing and promoting for all these years.

- A specific message that describes a detailed syntax and semantics for a language that can be used to describe a model and rules that allow us to "reason about" that model and thereby check certain properties of it.

I always find the general message convincing. Educated as an electrical engineer, I know that mathematics is a valuable tool for all engineering disciplines and that there is no reason that software should be an exception. As a daily user of current software, I see faults that I am convinced would not be there if mathematics had been used for software in the same way that engineers use it for designing physical products. However, I find the specific messages unconvincing.

- Even on small examples, it often appears that the model is more complex than the code.
- The model is a program (a sequence of data transformations); it is not easier to write or understand the model than it would be to write or read the program that would actually run.
- The models often oversimplify the problem by ignoring many of the ugly details that are likely to lead to bugs.
- Often the example does not include the final code, and, if it does, it is difficult to "connect" the model with the code; it seems possible for the model to be proven correct in spite of subtle errors in the code.

The problems are exacerbated in larger examples. Often, it is necessary to understand a lot about the model to answer even simple questions about the design. Sometimes the models are "write only" because no one but the author can answer questions.

We must learn to use mathematics in software development, but we need to question, and be prepared to discard, most of the methods that we have been discussing and promoting for all these years. We must examine the assumptions on which these methods are based and see which ones stand up to scrutiny.

WHAT TO RETHINK?

If we are really going to rethink formal methods, we need to objectively reconsider a set of issues.

Identifiers and variables

Robert Floyd, and most who followed him, represented the state of an executing program using the identifiers that appear in the program. Thus, if a program had a variable identified by "dogs," that string would appear in the predicate expressions used to characterize the state.

Variables in a program are finite state machines; what we generally call the value of a variable is that machine's state. The identifier is a string that we use to refer to a variable. A variable may have more than one identifier (aliasing), and one identifier may refer to different variables in different parts of a program.

In mathematics, variables are placeholders used to define functions/relations; they have neither state nor value. The difference is not always noted because the identifiers in programs look like the variables in mathematics.

Floyd implicitly assumed a one-to-one correspondence between the program variables and the identifiers. He also assumed that if an identifier does not appear in an expression or program line, the corresponding variable is not involved in the calculations when that line is executed. This is not always the case. "Workarounds" have been proposed, but they tend to complicate use of the methods. Many researchers have suggested that using programming techniques that destroy a simple relationship between identifiers and variables—for example, pointers—is bad programming practice; those practices are useful, and researchers are trying to cover up weaknesses in their methods by casting aspersions on things they cannot handle.

Arrays are a particularly vexing example of this problem. "A[j]" and "A[2]" may be identifying the same variable. If we treat them as different, we can prove a program correct when it is not. Edsger Dijkstra proposed treating the whole array as a single variable. This works nicely when all elements of an array are used in the same way, but is not always helpful.

A better way to deal with arrays is needed. In general, it is time to rethink how states should be represented.

Conventional expressions or something more structured?

The mathematical expressions used in most methods are relatively easy to read for simple expressions and when they are used to describe continuous functions. For complex programs, we are describing piecewise-continuous or discrete valued functions; in those cases, conventional expressions can be very hard to read and write correctly.

It is time to look for new forms of expressions that are designed for use with the functions implemented by digital computer programs.

Hidden state: Normal or extension?

Variables in the programming languages of the 1960s had the property that there was a simple relation between the visible value and the state. With the introduction of abstraction or information-hiding, and the subsequent introduction of user-defined "abstract types" into programming languages, it became possible to have variables such as stacks where the visible values are only part of the state. To deal with this, formal models often include ad hoc explicit state representations. These are arbitrary and often add complexity.

It is time to consider hidden state to be the normal case and develop methods that deal with it systematically.

Termination: Normal or exception?

The earliest formal methods assumed that a program would be started with initial values of a data structure, and execution would terminate with an answer in that data. They introduced the concept of partial correctness, which meant that if a program terminated, the answer would be correct, but the program might not terminate. Yet some programs fail by not terminating and others are intended to execute indefinitely. Extensions and notations that deal with nonterminating programs have been added to a basic model that assumes otherwise.

Rethinking requires asking if nontermination should be treated as the normal case in a way that lets us treat terminating programs as a special case.

Time: A special variable or another variable?

When the original formal methods were developed, execution time was not a major concern. If a program executed more quickly than expected, users were happy. If a program was too slow, the user might become annoyed, but the answer was still useful. Time was not a factor in the program correctness proofs.

With the advent of real-time systems, this all changed; programs that are too fast or too slow are incorrect. Special logics were developed for dealing with time issues. This is quite different from such areas as control theory and circuit theory, where time is represented by an additional variable that is not treated in any special way.

Rethinking would require serious consideration of this alternative. We gain simplicity if we do not have to treat time as anything special.

Axioms: Assignment or relational algebra?

In his early paper titled "An Axiomatic Basis for Computer Programming," C.A.R. Hoare introduced about a dozen axioms.[3] In logic, axioms are usually simple, intuitive, and obviously universally true. Hoare's axioms (which I believe to be essentially the same as Floyd's) don't have those properties. The axioms describing the arithmetic are not true of any practical computer. The axiom of assignment is only true under very restrictive conditions. The axiom given for iteration is more a sketch of a method of proof than an axiom since it requires identifying the right invariant.

It is time to look for new forms of expressions that are designed for use with the functions implemented by digital computer programs.

There is an alternative. Some researchers have been studying the use of relational methods in computer science; they note that the effect of a terminating program could be described by a relation on states and that the well-known laws of relational algebra can serve as the axiomatic basis for programming. The axioms of relational algebra are simple and universal. They do not embody the characteristics of any particular type of program and can be used with any set of primitive programs. This approach seems to have been neglected by most "mainline" researchers in the area of formal methods.

Direction of analysis: Forward, backward, or inside out?

Floyd, Hoare, and most others analyzed a program in the direction of execution—that is, starting with the first statement and initial conditions and continuing to the end of the program. Loops required an "inductive assertion" or an "invariant," but otherwise the direction was forward. In a break from previous work, Dijkstra proposed going in the opposite direction. His "weakest precondition" approach starts with the desired postcondition and determines what had to be true before the program ran to get the stated result. Few have followed him.

We can work either way. More important, going "bottom up" or "inside out," summarizing inner programs until the whole program has been summarized is also a possibility. In this approach, the relational method has an advantage because the axioms are not based on particular

programs (such as assignment) but are general and apply to programs of any size.

Side effects: Normal or bad?

One limitation on the axioms in the Hoare paper is that they are not true if the programs that evaluate an expression have side effects—that is, if they affect the values of other variables. Most mainline methods disparage side effects as a bad programming practice. Yet even in well-structured, reliable software, many components do have side effects; side effects are very useful in practice. It is time to investigate methods that deal with side effects as the normal case.

Perhaps a formal method should treat nondeterminism as the normal case and deterministic programs as a special case.

Nondeterminism: Normal or extension?

Early formal methods dealt with deterministic programs—programs in which the starting state determines the final state. When such methods are extended to deal with other programs, it is usually an afterthought and more complex. In practice, there are many reasons to deal with a component of a system as nondeterministic. Specifications are usually nondeterministic. Perhaps a formal method should treat nondeterminism as the normal case and deterministic programs as a special case.

Models, descriptions, and specifications

Many papers on formal methods use the words "model" and "specification" interchangeably, perhaps based on a standard dictionary definition of specification as specific information. This definition does not correspond to engineering use, and there are alternatives that should be considered.

- In engineering, the word "specification" is used in a narrower sense, denoting a detailed statement of requirements.
- A "model" of a product is something that resembles the original system but is simpler. Some properties of the model may not be properties of the original.
- A model or document is a "description" if everything that you can learn from it is true of the real thing. A specification is a description of a satisfactory product, but some descriptions are not specifications because they describe properties that are not required.

Using models that are not descriptions is dangerous because they can lead to incorrect conclusions about the real product. Treating descriptions that describe unnecessary properties as specifications may result in a product that is overdesigned or unnecessarily expensive. Methods that will be useful in practice must use models that are descriptions and clearly state whether or not they can be interpreted as specifications.

Specifications: Programs or predicates?

Before Floyd's work became known, some of the researchers interested in verification argued that since we had no way to state what a program should do, we could only prove program equivalence. They would write a program that was "obviously right," then prove that a more complex, usually more efficient, program would get the same answers.

This way of thinking seems to live on in some approaches to formal methods. The "specification" is a program that describes a sequence of state changes or data transformations.

Unless the sequence of transformations is a requirement, programs should not be used as specifications. An alternative view that has not received enough attention is to view a specification as a predicate. With a predicate you cannot directly compute an answer but you can easily check the correctness of a proposed answer.

It is time to look for methods that use predicates on observable behavior as specifications.

Specification language: Programming language or mathematical description?

The term "specification language" often causes confusion. Since a specification is also a description, specification languages are actually description languages. Further, most notations that are presented as specification languages are actually programming languages.

We should be considering methods that do not use the term "specification language."

What can be ignored?

A formal analysis uses a simplified description of the real system—that is, a model. Simplification is achieved by ignoring certain facts such as the limits in the sizes of data elements and the errors in arithmetic operations. Unfortunately, these are exactly the type of details that can cause faults and lead to failures. No formal analysis of such a model that leaves out critical limits can reveal faults attributable to those limits.

We should be looking for methods that do not ignore the finite limits that are one of the most frequent causes of bugs.

How do we establish correspondence between model and code?

Because practical programming languages often do not have a complete formal semantics, one that takes into account such issues as the support for software behavior

and finite limits, many formal methods work with a model quite different from the actual code. Often the connection between the code and the model is complex and not clearly described; in such cases, there is a question about whether the model could be (proven) correct while serious bugs remain in the code. In electrical engineering, a mathematical model is usually a set of equations that can be derived from the circuit systematically by, for example, using Kirchoff's laws. Some "idealization" happens in practice, but often this is taken into account by stating tolerances and deriving the possible inaccuracy in the computed result.

We need similar techniques for deriving mathematical models from program text. Floyd was careful to do this, but many of today's methods do not.

Mathematics in documentation

To do their job properly, both programmers and users need precise information. They need to know what is expected of their products and what they can expect of the programs they use. Even "small" details are important because, in software, small errors can cause serious failures.

Experience has shown that natural-language documentation rarely provides what is needed. Those documents are usually incomplete, imprecise, and poorly organized. The information in them is often wrong either because the original writer made an error or because the document was not properly updated when a change was made. There is no way to test an informal document. If the documentation is not trustworthy, a programmer in search of information must either find a knowledgeable colleague or read and understand thousands of lines of code written over many years by many other people. Neither technique is likely to provide complete and accurate information.

One of the most important roles that mathematics could play in software development would be to provide precise, provably complete, easy-to-use, testable documents. The popular formal methods have not been designed with use in documentation as the main goal.

When advocates of formal methods do provide documentation that can help programmers understand a program, it is usually in the form of assertions within the code. This works well for small programs but is impractical for large ones. For large programs and components, there is a need for external documentation that summarizes the behavior of hundreds or thousands of lines of code, allowing a programmer who uses that code to avoid reading it. Some formal methods use abstract models for this purpose, but these models do not usually capture all the details that programmers need to know about the programs they use.

Mathematical documentation should be a major research area in formal methods. It is not.

Pre- and Postconditions

The earliest formal methods were based on associating a program with a pair of predicates. One was the precondition, describing a class of states that must hold before the program is executed; the other was the postcondition, describing the states that must hold afterward. More than 30 years ago Susan Gerhart and Lawrence Yelowitz clearly showed that these are not really two separate conditions.[4] They specified a sort program with a precondition requiring only that there be some values in an array and a postcondition that the array be sorted. A program that assigned the value *j* to the *j*th element of the array would satisfy this specification.

Natural-language documentation is usually incomplete, imprecise, and poorly organized.

Instead of two separate conditions, we need a relation between starting state and stopping state. A few researchers have used this approach, but in most methods we still see pre- and post- as separate conditions. Extra variables are often added to allow the initial values to be mentioned in the postcondition.

It is time to consider abandoning the idea of pre- and postconditions.

Correctness proof or property calculation?

Computer scientists interested in mathematical software development methods have focused on "proof of correctness." Strangely, although engineers heavily use mathematics, they rarely use that phrase. Instead, they use mathematics to calculate properties of a product such as voltage on the output, harmonic distortion, heat loss, and so on. They use these calculations to evaluate and compare designs. "Correctness" is a useful term in mathematics, but not in engineering. In engineering, it is usually a serious challenge to define "correctness" for any application. Moreover, engineers are often interested in choosing the best design from a set of "correct" designs. Correctness is not the issue.

Researchers interested in developing practical formal methods should consider the engineering viewpoint; it replaces a vague general question with a set of specific ones.

OBSERVATIONS

This article is intended to ask questions, not answer them. There are, however, some observations that can be made.

Software is broken, but broken formal methods won't fix it

There is widespread agreement that something must be done to improve the quality of software. We have nothing better than mathematics for that purpose. However, there are serious questions about the popular formal methods, and researchers must find answers that are more convincing.

We need research, not advocacy

When we find that people are not adopting our methods, it is tempting to try "technology transfer" and other forms of advocacy. When that fails, which it has, we must return to research and look seriously for ways to improve those methods. It is our job to improve these methods, not sell them. Good methods, properly explained, sell themselves. Our present methods don't sell beyond the first trial.

Reach Higher

Advancing in the IEEE Computer Society can elevate your standing in the profession.

- Application in Senior-grade membership recognizes ten years or more of professional expertise.
- Nomination to Fellow-grade membership recognizes exemplary accomplishments in computer engineering.

GIVE YOUR CAREER A BOOST

UPGRADE YOUR MEMBERSHIP

www.computer.org/join/grades.htm

Step by step from user to code

Software is complex, and the only way to deal with this complexity is to proceed in small, systematic steps so that the relation between the abstract view given the user and the concrete code that runs on the machines can be followed. Any mathematics-based method must be an integrated set of techniques that supports a systematic step-by-step process. The integration means that consistent notation must be used at every step.

Abstract views must be simpler but true

Nobody doubts the value of abstraction, but it is essential to remember that everything that we can derive from an abstraction must be true of the real thing. If we can derive something that is not actually true, what we have is not an abstraction but a lie.

Our role model should be engineers, not philosophers or logicians

Engineers use mathematics in very different ways from pure mathematicians and logicians. Mathematicians who prove theorems use axiom systems that allow them to search for a proof. Engineers usually evaluate expressions, a process that requires no search, just repeated substitution of values for variables and application of functions.

More money is not the answer. It is common for researchers who do not achieve what they set out to achieve to blame the funding. Research in formal methods for software development has been very well funded. More money won't help; more thinking will.

References

1. H.S. Elovitz, "An Experiment in Software Engineering: The Architecture Research Facility as a Case Study," *Proc. 4th Int'l Conf. Software Eng.*, ACM Press, 1979, pp. 145-152.
2. R.W. Floyd, "Assigning Meanings to Programs," *Proc. Symp. Applied Mathematics,* Am. Mathematical Soc., vol. 19, 1967, pp. 19-31.
3. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, Oct. 1969, pp. 576-580.
4. S.L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Trans. Software Eng.*, vol. 2, no. 3, 1976, pp. 195-207.

David Lorge Parnas *is professor emeritus at McMaster University, Canada, and the University of Limerick, Ireland, as well as president of Middle Road Software. His nearly 50 years of research have delved into many topics looking for ways to connect theory and practice in the field of software design. Parnas received a PhD in electrical engineering from Carnegie Mellon University. Contact him at parnas_☺ mcmaster□.ca.*