



# "Formal Methods" Technology Transfer Will Fail

David Lorge Parnas

*Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada L8S 4K1*

For formal methods technology transfer to succeed, we must do two things: (1) integrate formal methods into basic university programming courses, and (2) improve the methods until they are better suited for practical application. Formal methods cannot be an "add-on" that is taught after programmers acquire bad habits, and they must produce documents that are easier to read than the programs that they describe.  
© 1998 Elsevier Science Inc.

## 1. WHY I DISLIKE THE PHRASE "FORMAL METHODS"

For someone who was educated in engineering, the phrase "formal methods" is strange and the issue of technology transfer of such methods even stranger. It is not that, as some non-engineers have suggested, engineers don't use mathematics in their work, but that engineers do not treat the use of mathematics as something separate from their normal work. Mathematics is an intrinsic part of everything that engineers do. In contrast, discussions of "formal methods" and their technology transfer treat the use of mathematical notation and methods as a distinct "add-on" that goes beyond the "normal" things that software developers already do.

The very first year of an engineering programme is filled with mathematics courses, science courses that use mathematics, and often an introductory course in which both mathematics and science are used to demonstrate how to solve engineering problems. Throughout the education, every engineering course, even those with a big laboratory component, makes extensive use of mathematics. In other words, mathematics is an integral and ubiquitous part of

what engineers do. It is not treated as a separate topic or something to be "transferred" after the basic engineering education is completed.

Mathematics is such an integral part of engineering that many engineers often are unconscious of the fact that they are using mathematics in their work. An engineer who considers certain design parameters for a component may not be conscious of the fact that those parameters have meaning only because they appear in equations that were solved long ago. I have known engineers who denied that they used mathematics, but carry a calculator and frequently use it to evaluate complex formulae. When engineers say that they do not use mathematics, it often means only that they do not develop new mathematics, i.e., prove new theorems.

## 2. HOW MATHEMATICAL TECHNOLOGY IS TRANSFERRED IN ENGINEERING

In the 35 years since I completed my basic engineering education, new mathematical tools have been developed for many parts of engineering (e.g., signal processing). Once the mathematics was found to be sound and useful, it was incorporated into undergraduate programs so that today's engineering undergraduates are learning mathematical techniques that were unknown when I was a student. Engineers who graduated before these techniques were taught knew that, as professionals, they would have to keep up to date and have since learned the new concepts that they need for their job. Nothing called "technology transfer" was necessary. Nobody had to "sell" these techniques; if they were useful, they caught on. If professionals see that a method is helping competitors to do a job better, they will learn it.

It is frequently suggested that new methods must be proven by formal experiments before transfer can be successful. People who advocate techniques with-

---

*Address correspondence to Dr. D. L. Parnas, McMaster University, Department of Engineering, Hamilton, Ontario, L8S 4K1 Canada. E-mail: parnas@qusunt.crl.McMaster.ca*

out supportive experimental data are criticised as unscientific. Such positions are not supported by history. If any formal experiments "proved" that those who used Kirchoff's laws in circuit design did a better job than those who did not, those experiments have long been forgotten. Similarly, I cannot imagine anyone using control groups in an experiment to demonstrate that solving differential equations was useful in designing control systems. The mechanism for adding these techniques to the engineer's tool kit was much less formal. New ideas were added to university curricula as they matured. When practising engineers found that new graduates or consultants could do things that they could not do, they knew that market forces would work against them if they did not upgrade their own capabilities to match those of other engineers.

Of course, not all engineers keep up with the latest mathematics. Some don't need to keep learning because their job involves doing routine work for which the methods they learn in school are adequate. Others stop doing engineering and work as managers or salesmen. Still, every engineer learned a lot of mathematics while in university and is aware that some tasks may require them to learn more.

### 3. WHY SOFTWARE DEVELOPMENT IS DIFFERENT

Something has gone wrong in the field that we (improperly) call "software engineering." Software practitioners see their real job as something that they can do without using mathematics; they view mathematics or "formal methods" as a kind of add-on, an option.

Most computer science programs are schizophrenic. In some courses, we teach mathematical theory but spend little time showing how to apply it in software development. In other courses, we teach programming but rarely apply any theory. I have even seen books that were really two books in one, a book on some aspect of mathematics, and a completely separate book on some practical aspect of programming. One could read either sub-book without the other, or read them in arbitrary order, so weak was the connection between the two.

The split in the educational programs corresponds to a split in departments. On one side, most computer science departments have "systems" people, who teach "real world" course that depend almost entirely on intuitive analysis. On the other side, they have theoreticians who are developing mathematical models without feeling obligated to show how those models apply to objects in the real world.

The "systems" people give courses in which they present programs without precisely specifying the problem, explain and analyse those programs using ad hoc procedures and arguments, and generally give the impression that when a program appears to work, the programmer's job is done. Often they treat currently available languages and systems as if they were a natural phenomenon, comparing operating systems, etc. the way that some biologists would compare species of birds.

Many theoreticians, on the other hand, give courses in which they create a world that fits their theories. For example, we find specialists in denotational semantics who, unable to explain side-effects with their theory, declare side-effects to be a bad idea. In fact, side-effects are unavoidable in practical computing systems and the inability to handle them is a serious fault of some theoretical models, not a sign that we should avoid side-effects.

The split in computer science departments has meant that we have graduates who know some "theory" and know how to program, but don't know how to use theory when programming. In other words, in our teaching and in our work we have created a gap between theory and practice. It is this senseless gap that has led us to write papers about technology transfer and formal methods. Our students learn ad hoc methods and practice them; later, saying that we are doing technology transfer, we try to correct this error. Unfortunately, the attitude that mathematics is something extra means that people are not prepared to learn any new mathematics. Someone who wanted to learn Z or VDM, would find it easy if they had learned classical logic and knew how to apply it. Unfortunately, even if they have taken a logic course, they have put it in a compartment labelled "other stuff" in their minds while they study what they see as "real" programming. It doesn't help them to understand how a new, logic-based, technique might be useful.

The assumption that we must separate "fundamentals" and applications is so deep that people sometimes argue over the order in which they should be presented. I recently attended a series of lectures where the speaker addressed the issue of which should we teach first, fundamentals or program design. His position was that we should be teaching fundamentals first and not allowing students to program in the early years of their education. His opponents wanted to teach programming first and leave "fundamentals" for later, more advanced courses. I disagree with both sides. Every programming course should show students precise specifications, every lecture should demonstrate program de-

velopments in which mathematics plays an integral role, and every course on mathematics should include discussion of how to apply it.

#### 4. WHY "INTEGRATED" COURSES ARE NOT POPULAR

A major barrier to integrated courses, courses in which "formal methods" and programming are inseparable, is the immaturity of the best known methods. The purveyors of these methods seem to have confused existence proofs with demonstrations of practicality. Several surveys of the state of the art in formal methods have given glowing descriptions of the success of massive efforts by highly skilled people and implied that these experiences prove that formal methods are practical. This is similar to watching a successful Mars landing and concluding that your next vacation will be at a crowded hotel along a Martian canal.

In fact, close examinations of these "successful" applications of formal methods reveal them to be evidence of tremendous determination on the part of those who did the work. The formal documents themselves are carefully done, but very detailed and incredibly tedious. The amount of writing and notation seems totally out of proportion to the complexity of the tasks. We find page after page of complex formulae that often can only be understood as a whole or must be checked by a machine. I have good reason to believe that few people have been able to read and understand those documents.

I myself have written several articles (e.g., [Parnas et al., 1991; Parnas, 1994b]) describing successful use of mathematical methods to inspect safety-critical programs for a nuclear plant. However, close examination of such experiences reminds us that there are two sides to every story. Our methods required an amount of effort that could only be justified because the potential cost of a software error was very high. Although the methods can be made to work, they are not methods that you would want to apply in every day work. This is not to be interpreted as criticism of our methods when compared to others; we chose these methods after investigation revealed that others were even worse.

The perception that the use of formal methods in software development lends a lot of "overhead" to programming is in sharp contrast to the situation in classical engineering. For example, the use of Kirchoff's laws and differential equations to design circuits is clearly a lot more effective than the trial-and-error, intuitive techniques that one would have to use without such mathematical methods. For ex-

ample, I was once working in a radio station when an important device quit working. In a hurry to get back to my real work, I tried to use intuitive shortcuts to figure out what had gone wrong. Each of my guesses proved wrong and hours went by without results. Finally, in frustration, I decided to apply what I had learned in my circuits classes. Within an hour I had found, and corrected, the problem. In other words, "Try it you'll like it!" is a statement that applies well to the mathematics taught in good engineering curricula. In contrast, those who do try to use popular formal methods in programming are often discouraged from further use by the experience.

A key sign of the immaturity of the methods is what happens after the "success". Often, the organisation that completed the effort does not continue to use the successful methods; the level of effort was just too great to allow deadlines to be met. In the case of the work described in (Parnas et al., 1991; Parnas, 1994b), the methods continue in use, but the software itself is being redesigned in the hope of reducing the amount of effort needed to perform future inspections. However, I have found other cases where the methods are dropped completely after the initial effort.

#### 5. WHAT'S WRONG WITH POPULAR "FORMAL METHODS"?

One problem with most formal methods is that their application does not yield concise but accurate summaries of program behaviour. In other engineering areas, we are able to use mathematics to summarise the properties of complex devices so that they can be treated as a black box when they are used as components in other systems. With many formal methods the formal description contains more detail than the actual product. Many methods are based on the old idea of placing assertions at key points in a program. These assertions tell us about the state at certain points in the program text, but do not summarise the effects of programs. Alternatives are discussed in (Parnas, 1994a).

A fundamental issue that will have to be solved before "formal methods" are used in practice is how we will distinguish between model, description, and specification. This is overlooked or ignored in most "formal methods" papers. The word "specification" or "specification language" is used whenever people use notation different from practical programming language. It seems "specification" serves as a euphemism for "unreal" in the phrase "specification language."

The purveyors of VDM, Z, Lotos, or Estell, have focused on their notation, but ignored the question of what information belongs in each document or "work product" produced during software development (Parnas and Madey, 1995). Facing such issues is essential to producing an integrated method, one in which mathematics is an inseparable component of a software development process. There is a definite need for improved notations, but the first question that should be answered is the question of document content.

In the area of "formal methods," there is often an unreasonable focus on what is called correctness. In other areas, engineers rarely prove theorems; they use mathematics to compute important properties of their proposed designs. They also use mathematics to describe their designs in a way that makes mathematical analysis possible. The difference is subtle, but it is the key to practicality. If I want to prove that a circuit is correct, I start with the daunting task of trying to write down everything that I expect of that circuit, including some things that are obviously true but hard to express. If I try to determine key attributes of the circuit, e.g., the maximum voltage across the terminals of a particular component or the resonant frequencies, I have a concrete, limited, and achievable, goal.

#### 6. IF AT FIRST YOU DON'T SUCCEED, ... BLAME OTHERS?

Many researchers have turned their effort from improving their methods to technology transfer. This implies that the technology is fine; the problem is that the practitioners don't understand it or aren't willing to learn it. The technology is not fine! Those who are now working on "technology transfer" would be well advised to return to their research and improve their methods. When they can demonstrate methods that they can use in their own routine

programming, technology transfer will take care of itself.

An old Dutch expression, "selling the pelt before the bear has been shot," accurately describes the activities of many people who are now focusing on technology transfer. Many spend their time berating practitioners for not applying their method. We all need to disseminate our ideas, but most of our time should be spent applying and improving our methods, not selling them. The best way to sell a mouse trap is to display trapped mice and be able to state the cost per rodent. Trapping real mice also shows you how to improve your traps.

#### 7. CONCLUSIONS

We will succeed in our efforts to reduce the cost and improve the quality of software through the use of mathematics only if we go back and re-examine our basic educational programs. We cannot educate people to see programming and mathematics as two separate fields, and later try to "paper over the gap" by means of technology transfer courses. We can't sell methods that we don't use ourselves. Technology transfer will fail, but research and education will not.

#### REFERENCES

- Parnas, D. L., Mathematical Descriptions and Specification of Software. Proceedings of IFIP World Congress 1994a, Volume I, August 1994, pp. 354-359.
- Parnas, D. L., Inspection of Safety Critical Software using Function Tables. Proceedings of IFIP World Congress 1994, Volume III, August 1994b, pp. 270-277.
- Parnas, D. L., Asmis, G. J. K., Madey, J., Assessment of Safety-Critical Software in Nuclear Power Plants. Nuclear Safety, Vol. 32, No. 2, (1991), pp. 189-198.
- Parnas, D. L., Madey, J., Functional Documentation for Computer Systems Engineering. Science of Computer Programming, Vol. 25, No. 1, (1995), pp. 41-61.