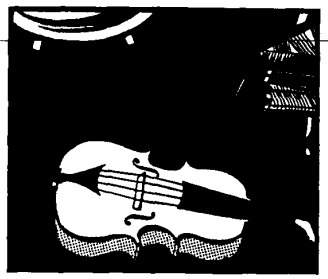


Seven Myths of Formal Methods

Anthony Hall, *Praxis Systems*

Formal methods are difficult, expensive, and not widely useful, detractors say. Using a case study and other real-world examples, this article challenges such common myths.

Formal methods are controversial. Their advocates claim they can revolutionize development. Their detractors think they are impossibly difficult. Meanwhile, for most people, formal methods are so unfamiliar that it is difficult to judge the competing claims. There is not much published evidence to support one side or the other, and a lot of what is said about formal methods is based on assertions, not on facts. Thus, some of the beliefs about formal methods have been exaggerated and have acquired almost the status of myths.

Praxis is a software-engineering company where we use formal methods for real projects: We write real specifications, not just exercises, and we develop real software from them. As a result of this experience, many of us are enthusiasts for formal methods. We have found that they offer real benefits; at the same time, we have found that many things that people believe about formal methods are not true.

This article takes a practical look at formal methods, presents some of the myths — favorable and unfavorable — and explains what we have found to be the truth behind them. As an example throughout this article of the use of formal methods, I draw particularly from our experience on a CASE project, which is described in the box on p. 13

The CASE project was certainly not the kind of project that most people associate with the use of formal methods, and we did not do a completely formal development involving proofs and program verification. Nevertheless, we found that we gained enormous benefit from using the Z specification notation,[1] which is one of several formal-methods notations.

The seven most prevalent formal-methods myths are variants of the following:

1. *Formal methods can guarantee that software is perfect.* The most important myth is that formal methods are somehow all-powerful — if only we mortals could apply

0740-7459/90/0900/0011/$01.00 © 1990 IEEE

them. This is a pernicious myth, because it leads to both unrealistic expectations and the idea that formal methods are somehow all-or-nothing. The reality is that no such guarantee can be given — but the usefulness of formal methods does not depend on such absolute perfection.

2. *They work by proving that programs are correct.* In the US, a lot of the work in formal methods has concentrated on program verification. This has made formal methods seem very hard and not very relevant to real life. However, you can achieve a lot without any formal proofs at all.

3. *Only highly critical systems benefit from their use.* This belief is based on the perceived difficulty of using formal methods. The truth is that critical systems do demand the most thorough use of formal methods, but any system benefits generally from using at least some formal techniques.

4. *They involve complex mathematics.* Formal methods are based on mathematics, and many people believe that this makes them too difficult for practicing software engineers. This myth is in turn based on a view that mathematics is intrinsically difficult. At Praxis, we have found that the mathematics of specification, at least, is easily learned and used.

5. *They increase the cost of development.* It used to be said that, although the use of formal methods was very expensive, it was worthwhile because of the lower maintenance costs for the resulting software. But this is a difficult argument to sell to hard-pressed project managers, whose budget is for development, not maintenance. In fact, we have some evidence that *development* can be cheaper when you use formal specification.

6. *They are incomprehensible to clients.* A formal specification is full of mathematical symbols, which render it incomprehensible to anyone unfamiliar with the terminology. Therefore, it is supposed, a formal specification is useless for nonmathematical clients. However, mathematics is not the only part of a formal specification — it supports many other ways of expressing the specification that give the client a better understanding early on in the project.

7. *Nobody uses them for real projects.* Formal methods are often associated with academic departments and research organizations. It is thought that only such organizations have the expertise necessary to use them and that they are only suitable for the idealized applications that such groups would carry out. But our experience in the CASE project, and the experience of other industrial users, is turning this point of view into a myth — or at least into history.

Myth 1

Formal methods can guarantee that software is perfect.

The fact is that formal methods are fallible.

It ought to be too obvious to need saying, but nothing can achieve perfection. Unfortunately, sometimes proponents of formal methods claim they offer an absolute guarantee that cannot be achieved any other way.

It ought to be too obvious to need saying, but nothing can achieve perfection. Unfortunately, it sometimes seems that proponents of formal methods claim they offer an absolute guarantee that cannot be achieved any other way. If you take this position then any problem with formally developed software is a refutation of formal methods' usefulness. Formal methods have been strongly criticized precisely on this absolutist basis.

It is important to understand formal methods' intrinsic limitations. Their fallibility is the most fundamental limitation, and it arises from two facts: Some things can never be proved and we can make mistakes in the proofs of those things we can prove.

Limits on proofs. A proof is a demonstration that one formal statement follows from another. The real world is not a formal system. A proof, therefore, does not show that, in the real world, things will happen as you expect. So you can never be sure that your specifications are "correct," however much you prove about them.

This should not deter you. All engineering is concerned with making models of the real world and using those models to design artifacts. Models based on mathematics are ideal because you can establish the models' properties by reasoning and because you can manipulate the models during design. The designer of a crane, for example, abstracts the real crane into a structure of idealized components with known properties like mass and load-bearing capacity. He uses this model to design and predict the properties of the real crane. There is no way he can prove that the real crane will behave as he predicted.

But, on the whole, the correspondence between the mathematical models used in structural engineering and the real world is well-enough understood that we trust such mathematical models. The more mature the engineering discipline, the more likely we are to trust the models it uses. There have certainly been enough engineering disasters to convince anyone that the correspondence is not perfect, but nobody would suggest that crane builders should abandon mathematics.

In software, the limits of our modeling techniques are also reasonably well understood. First, models cover only some aspects of a program's behavior. Second, the correspondence between the formal description and the real world is limited.

There are good mathematical models for the behavior of sequential programs. Models for concurrent behavior are also available but less easy to use. Some people say we cannot model timing constraints formally; this is not strictly true, but it is true that we do not know how to use such models to help us develop software that meets the constraints. Finally, we cannot yet model nonfunctional properties like performance, reliability, maintainability, and availability.

The correspondence between our formal models of programs and the actual behavior of real systems is limited by three factors: the behavior of the programming language, the operating system, and the underlying hardware. For safety-critical systems, these limitations are crucially im-

portant and we cannot assume that a program is correct just because it has been proved.

Mistakes may be made. Even within our formalism, we can make mistakes in doing proofs, just as we can make mistakes in writing programs. Indeed, published formal specifications have errors in them.

In spite of these apparent problems, formal methods *do* work. There are two reasons. One is that there are some ways in which formal methods offer qualitatively different and better guarantees than any other method. The other is that even though formal methods still let you make mistakes, they are much better at exposing these mistakes.

Demonstrating correctness. There is an often quoted remark that "Program testing can be used to show the presence of bugs, but never to show their absence!"[2] This seems to imply that something else — proving — can show the absence of bugs. There are two senses in which this is true (although in both cases the possibility of errors in the reasoning process means that the demonstration is not absolutely infallible):

- Some properties can be established *only* by formal reasoning. Many requirements are couched as universal statements, like "The program will always log user actions" and "The system will never lose a message." Such statements can in principle not be established by testing or simulation, but they can be established by reasoning about the specification.
- Some steps *can* be demonstrably correct. For example, the relation between a program and its specification is a formal one and can be proved to be correct. So you can nearly guarantee that a program meets a specification, even though this does not mean that the program is perfect. (The guarantee is only "near" because of the limits of the mathematical model in capturing the real world; even if the guarantee were absolute, it would not mean the program was perfect because the specification might be wrong.)

Finding errors. Although they eliminate only certain classes of errors — and then not with absolute certainty — formal methods do make it much easier to find all sorts of errors. In an informal specification, it is hard to tell what is an error, because it is not clear what is being said. When challenged, people try to defend their informal specification by reinterpreting it to meet the criticism. With a formal specification, we have found that errors are much more easily found — and, once they are found, everyone is more ready to agree that they are errors.

In this sense, formal methods are a scientific approach to development, since they offer specifications that can be refuted. (In informal software development, the specification is usually only refuted by testing. By this stage, it has of course been made formal — by translation into a programming language — but it is no longer easily comprehended by people.)

In the CASE and other projects using formal methods at Praxis, we have found that the ability to expose errors is one of these methods' key benefits. Even though we have undertaken very few proofs or completely formal development steps, we have found that inspections of formal specifications reveal more errors than those of informal specifications, and it is more effective to inspect designs or programs against formal specifications than against other kinds of design documentation. IBM has reported similar experiences.[3]

The CASE project

The CASE project we applied formal specifications to is a software-engineering tool set to support project teams using SSADM, a structured systems-analysis and -design method. Each team member has a workstation, and the workstations are networked to a central project machine. The infrastructure of the CASE project provides

- a multiuser distributed project-management and configuration-management system controlling all development information and tasks and
- a set of basic classes (like diagram, table, and matrix) from which tools for structured analysis can be developed by specialization.

The infrastructure is implemented on top of Sun Unix. It is coded in Objective C.

The specification is a document of about 340 pages written in Z with English comments. It contains about 550 schemas defining about 280 operations.

Development from this specification proceeded by

- writing a concrete specification of the interfaces in Objective C,
- writing, for some parts of the system, informal design documents,
- coding other parts directly from the Z specification,
- writing some Z specifications of lower level modules, and
- coding from the informal designs or lower level specifications.

We did no proof or mathematical program construction.

We used our normal company standards for project planning, integration and testing, configuration management, and so on.

We coded about 58,000 lines of Objective C, of which about 37,000 lines were deliverable software.

The project lasted nearly 90 weeks and used about 450 man-weeks of effort, of which about two were devoted to the system specification.

Myth 2

Formal methods are all about program proving.

The fact is that formal methods are all about specifications.

I use the term "formal methods" to cover the use of mathematics in software development. The main activities I include are

- writing a formal specification,
- proving properties about the specification,
- constructing a program by mathematically manipulating the specification, and
- verifying a program by mathematical argument.

Thus, program verification is only one aspect of formal methods. In many ways, it is the most difficult. For non-safety-critical projects, program verification is far from the most important aspect of a formal development. Since the cost of removing errors increases dramatically as a project progresses, it is more important to pay thorough attention to the early phases.

System specification. From an economic point of view, therefore, the most important part of a formal development is the *system specification*. For many projects, this is the only part of the development that is formal. In any case, a formal specification of what a program is to do is a prerequisite for verifying that the program is correct.

A formal specification is a precise definition of what the software is intended to do. You can give any piece of software, from a single module to a whole system, a formal specification. On the CASE project, we used Z to write the formal specification of the whole system. Such system specifications are the most practical and valuable ways of using formal methods.

A formal system specification is comparable in scope to a conventional requirements analysis using dataflow or entity-relationship diagrams. It differs from conventional design specifications in that it is concerned only with the function of the system and makes no commitments to its structure.

To illustrate the notion of a formal specification, the box below shows an example that is a simplification of part of the CASE specification. It is written in Z. A Z specification is a mathematical model of the system to be built. It consists of two parts: a definition of the state of the system and a collection of definitions of operations on the system.

A specification is abstract in three senses:

- It uses data types, like sets and relations, that can model applications directly, rather than computer-oriented types like arrays. In the example, I use sets to represent the collections of tasks and documents in the system and a function to represent the relationship between them. These representations capture the es-

Example formal specification

The CASE project system contains a collection of documents and a collection of tasks. Each document is produced by a task; tasks may produce more than one document; all tasks produce at least one document.

To describe this in Z, we built a mathematical model. We did not say what "tasks" and "documents" are, so we just let these be represented by the names TASK and DOC at this stage. In Z notation, the text in the first part of a schema is the declaration, which describes the model's *components*; the text in the second part of the schema is the predicate, which describes the model's *properties*. Schemas are split by horizontal rules.

Defining tasks and documents. This part of the model is called TasksAndDocuments. The specification is

TasksAndDocuments

$documents : \mathbb{P}\ DOC$
$tasks : \mathbb{P}\ TASK$
$outputTask : DOC \rightarrowtail\!\!\!\!\!\!+\!\!\!\!\!\!\rightarrow TASK$

$\mathrm{dom}\ outputTask = documents$
$\mathrm{ran}\ outputTask = tasks$

In Z, the symbol for a set of things is **P**, which you can pronounce "set of." The first two lines of our model define the components documents, which is a set of DOCS, and tasks, which is a set of TASKS. This expresses the fact that "the system contains a collection of documents and a collection of tasks."

Next, you must say that "each document is produced by a task." We did this in two parts. First, we set up an association between documents and the tasks that produce them, which we called outputTask. This association is written as a function, for which the Z symbol is $\rightarrowtail\!\!\!\!\!\!+\!\!\!\!\!\!\rightarrow$, that tells us that a document can only be the output of one task.

Then you must say that each document is produced this way, so you say that the action associates all the documents you know about with tasks: That is done in the statement "dom outputTask = documents," because the expression "dom outputTask" means "all the documents that are associated with tasks by the function outputTask."

Similarly, the Z expression "ran outputTask" means "all the tasks that are associated with documents by the function outputTask." To express the requirement that all tasks produce at least one document, we said "ran outputTask = tasks."

The final part of our English specification is that "tasks may produce more than one document"; there is no need to say anything special about this in the mathematics, since the specification as it stands allows it. In formal specifications, anything not forbidden is allowed; if we had wanted to say that "tasks may not produce more than one document," we could easily have done so.

Removing documents. We then specified the operation to remove a document. A document can be removed only if it is known to the system. When it is removed, the document is no longer recorded as a task's output. If this causes a task to have no remaining outputs, the task is also removed. The specification is

RemoveDocument

$\Delta TasksAndDocuments$
$oldDoc? : DOC$

$oldDoc? \in documents$
$outputTask' = \{oldDoc?\} \lhd outputTask$

To say this in Z, you first say that you are defining an operation that changes the part of the state called TasksAndDocuments; that is the meaning of the line "ΔTasksAndDocuments."

Next, you declare that the operation has an input parameter, oldDoc?, of type DOC, which is the document to be removed.

Now you have to say what the operation actually does. First, for it to do anything, the document you are trying to get rid of must be one of the known documents. In Z, you say it must be a member of the set of known documents: "oldDoc? ∈ documents."

Finally, you define the effect. You can do this very simply: All you do is remove the document from the function outputTask. The way to do this in Z is to give an equation that tells you what the new value of outputTask, called outputTask′, will be. The symbol for removing elements from the domain of a function is ⩤, so the equation you want is "outputTask′ = {oldDoc?} ⩤ outputTask."

You can rely on the other properties of the state to ensure that, when you do this, the document will also disappear from the documents set, since you defined the documents set to be identical to the domain of outputTask. Furthermore, if this leaves a task with no outputs, that task too will disappear, since all tasks are defined to produce at least one document. If you want, you can prove that these changes will happen.

sence of what is required better than the corresponding implementation structures.

- It specifies *what* is to be done rather than *how* it is to be done. The definition of the operation RemoveDocument, for example, simply says that, after the operation, the relevant document has been removed. It needs to say nothing about how the removal is done, nor how any related task is found and removed.
- It specifies only whatever level of detail is necessary; you can simply leave unsaid things that are not important. In the example, we did not say what TASK and DOC actually were. This too is an implementation detail of no interest to the specifier or client.

This abstraction represents a proper separation of concerns between what the users want to define and what they are content to leave to the implementers. Such separation of concerns is important in controlling the development process, whatever life-cycle model you use. For example, in a development that uses prototyping to explore user requirements, it is important to separate the essential behavior of the prototypes from incidental details of the prototype implementation.

You remove the incompleteness of the specification in two ways. First, you record in other documents like statements of nonfunctional requirements those things that you would like to say at the specification stage but cannot because of your mathematical models' limitations. Second, you supply during the subsequent design and implementation steps the information that has been deliberately omitted.

Occasionally, these subsequent steps reveal problems with the specification that had been hidden by the abstraction. For example, it is possible to write specifications that cannot be implemented efficiently. In that case, you must revise the specification itself at the design stage.

A specification is central to a project in three ways:

- The actual process of constructing the specification is as important as its existence.
- Proofs of the specification's properties are at least as useful as proofs of correct implementation.
- You can construct implementations from the specification so they are correct.

Benefits. We found that writing the CASE specification helped us to clarify the requirements, discover latent errors and ambiguities, and make decisions about functionality at the right stages.

For example, we started off with elaborate requirements for documents to have different status values with complex transitions between them. A formalization of this let us simplify the model into a few distinct concepts. For example, we modeled the extent of machine-checking a document separately from how far it had been through a formal approval process. This made it easy to understand and verify with the user that our rules governing these status values were correct. Such clarification of requirements can lead to smaller and simpler systems — and to less rework in system test.[4]

Formal specifications let you say whatever you think is important at the specification stage. But, if you really are prepared to leave decisions until a later stage, you can do that, too.

It is hard to fudge a decision when writing formal specifications, so if there are errors or ambiguities in your thinking, they will be mercilessly revealed: You will find you cannot write a coherent specification or that, when you present the specification to the users, they will quickly tell you that you have got it wrong. Better now than when all the programming money has been spent!

Several times during the development of the CASE project, we discovered unexpected consequences of the specification. For example, early on we wrote a specification that allowed documents, but not tasks, to have versions. We rapidly discovered that we could not express this model formally. To get over this, we introduced the concept of a task version, which represented the running of a task with a particular collection of document versions. This concept turned out to represent a real-world object that was central to the way that the CASE tool set would be used, but we had not been able to see this usage clearly in an informal description of the system.

Formal specifications let you say whatever you think is important at the specification stage. At the same time, if you really are prepared to leave decisions until a later stage, you can do that, too.

Our example has a typical instance of such a decision. We defined, in the specification, precisely what happens when the last output of a task is removed: The task is removed as well. It is likely that an informal specification would not have made this clear, and the coder would have had to make a decision. But this clearly is a specification matter, since the effect is visible to the user. Omitting it from the formal specification, whether accidentally or deliberately, would be very obvious — there would be a component of the state whose value was undefined.

Specifications and proofs. Once you have a formal specification, you can prove things about the specification itself, as well as proving that a program satisfies it. These other properties may have to do with consistency of the specification or completeness of operation definitions. They may also be proofs that the specification (and thus the developed software) will meet certain key requirements. For safety and security, these may be certain kinds of integrity or other important requirements. In any case, because errors at this stage are more costly than implementation errors, proofs of these properties are correspondingly more important than proofs of implementations. Jim Woodcock[5] has shown reasoning applied to a practical specification (the CICS storage manager).

Implementing from formal specifications. When you do come to implement specifications formally, you do not do it by writing a program and then trying to prove that it meets the specification. This is infeasible for any but the smallest programs. Instead, you construct a correct program in small steps. Each step takes

the specification and produces something a little nearer to the final program. Each step is small enough that you can see exactly what needs to be proved to show that the step is correct — and, if you doubt the correctness, you can actually carry out the proof. This style of development is described in a textbook on the Vienna Development Method[6] and a book on constructing correct algorithms.[7] It has been used, for example, to implement hardware from a formal specification in Z.[8]

Each design step in such a development adds some detail that was omitted from the formal specification or makes some decision that was postponed. The implementers must

- provide efficient implementation structures to represent the application concepts,
- know or develop algorithms to carry out the required operations, and
- fill in details where these have deliberately been left to their judgment.

In the CASE project, we used formality only in writing the specification. We did not try any program proving at all. The kinds of design steps we made on the CASE project were to:

- Decide on a concrete language interface for the operations.
- Decide on a concrete data structure to represent some abstract structure in the specification; for example, an object class to represent the function outputTask. The designer was free to choose any suitable representation that had the required properties.
- Decide on some lower level operations needed to implement the top-level operations. For example, we identified a component called the kernel that provided low-level storage and distribution functions. We specified this component formally and implemented it from its Z specification.

Of course, these design steps required creativity: The specification did not overconstrain the designers, but it also did not do their job for them. We found that making such design decisions was in practice relatively straightforward and that, most important, it was easy to see if any proposed design met the specification.

A specification is a kind of contract between specifiers and implementers, and if the specification is formal, it is easy to interpret the contract and to decide if it has been satisfied.

Myth 3

Formal methods are only useful for safety-critical systems.

The fact is that formal specifications help with *any* system.

Probably the largest practical applications of formal methods have been in noncritical projects. Our CASE project, for example, was not at all safety- or security-critical. Formal methods should be used wherever the cost of failure is high. Systems whose cost of failure is high include those that are

- critical in some way,
- replicated many times,
- fixed into hardware, or
- dependent on quality for commercial reasons.

Almost any serious piece of software qualifies for at least one of these reasons. Our CASE project, for example, had to be a high-quality product to satisfy the client and its users.

Applying formal methods can benefit many areas, including fitness for purpose, maintainability, ease of construction, and better visibility.

Formality offers ways to ensure the right software is built. You can discuss the specification with the user and, in some cases, build prototypes on the basis of the specification to demonstrate just what is proposed. You can use formal reasoning to demonstrate some of the specification's consequences, giving you something on which to have a discussion with the user.

One of the main problems in maintaining software is knowing what it is supposed to do. Another is knowing what each part is supposed to do, and thus what must be preserved as the software is changed. Formal specifications are ideal for this purpose.

Our experience shows that it is easier to build a system from a formal specification than by using other methods. Even when we have not done development rigorously, we have found coding from a formal specification to be straightforward.

The application of formal methods can also make you more confident in the development process because at each stage it is clearer what has and has not been done. Monitoring is more reliable and thus development is less risky.

Starting from a formal specification, the development process can be very rigorous, if it is done in small steps with each step formally expressed and justified. It can also be less rigorous, if the steps are larger and justified only informally. You choose the degree of rigor to suit the application. If the system is critical, it must of course be developed completely formally.

However, many benefits of formal methods come from the specification stage. Thus, on a noncritical system, even if none of the rest of the development is formal, just writing a formal specification is a big improvement over other informal methods.

Myth 4

Formal methods require highly trained mathematicians.

The fact is that the mathematics for specification is easy.

Once it is recognized that the practice of formal methods is most concerned with writing specifications, the mathematical difficulties become much less significant. You can develop specifications themselves with very straightforward mathematics that any practicing engineer should know.

For example, in Z, the only branches of mathematics you need to write specifications are set theory and logic. The elements of both these are easily understood and nowadays are taught to teenagers.

Of course, before engineers can use formal methods, they must be trained — in this respect, formal methods are no different from other methods. Our experience is that such training is not difficult and that people with only high-school math training can write excellent formal specifications. Certainly anyone who can learn a programming language can learn a specification notation like Z.

The specification of a problem is shorter and much easier to understand than its expression in a programming language. Consider the operation RemoveDocument in the example in the box on p. 14: A definition of this operation in, say, pseudocode would be far longer and less comprehensible.

People have a fear of new symbols. But

mathematical symbols are introduced to make mathematics easier, not more difficult. People quickly become familiar with the new symbols. The difficulty in learning logic is not the symbols, any more than the difficulty in learning Russian is learning the Cyrillic alphabet.

Difficulties. This does not mean that everything about writing specifications is easy. When the notation has been learned, there are still difficulties. Some people are better at it than others, just as some people are better at programming than others.

The main difficulty is making the right connections between the real world and the mathematical formalism. It can be hard to choose the right things in the real world to model — to get the right level of abstraction. Some programmers put too much detail into their specifications and make them too complicated. You can also make the opposite error: writing specifications that are too abstract.

However, these are problems of any kind of specification, not problems introduced by formality. Many programmers find it difficult to write specifications in any notation, because it is difficult for them to get away from programming-language detail. When using formal specifications, studying good published case studies and getting advice from an experienced person can help you avoid these problems.

Training hints. We have found that there are three stages of training needed:

- Training in discrete mathematics, which needs to cover elementary set theory and formal logic. For those who have a mathematical background but are unfamiliar with these topics, a single day suffices to introduce the ideas. Even for the innumerate, less than a week's training is needed. There are many good textbooks on discrete mathematics.
- Training in the particular formal notation. A Z or VDM course typically takes one or two weeks, assuming that the participants have the necessary mathematical background. Textbooks are available for VDM[6] and Z.[1]
- Tutoring and consultation in real projects. After training, students can use formal methods, but they will still encounter difficulties. To get over these, we recommend workshops where you can tackle problems with the help of a tutor. It is also essential that every project using formal methods have access to at least one person with experience using the method. If necessary, you can ensure this by hiring some consultants during the early stages of the project: 10 man-days of effort, used wisely, may suffice.

A much higher level of mathematical skill is needed if you intend to go beyond formal specification and carry out a fully formal development that includes proofs. It is unrealistic to expect the majority of software engineers to be able to do proofs easily. Nor is it likely that machine assistance will be any help. Proof tools are still in a very primitive state — and, in any event, there are fundamental difficulties with machine assistance for proof.

A much higher level of mathematical skill is needed if you intend to go beyond formal specification and carry out a fully formal development that includes proofs.

Therefore, competent people who can cope with the necessary mathematical manipulations are the ones who must carry out safety-critical projects. Of course, the same is true of bridge building.

Myth 5

Formal methods increase the cost of development.

The fact is that writing a formal specification *decreases* the cost of development.

A completely formal development that includes proving each development step is very expensive — probably infeasibly so for all but the most critical applications. But because many benefits come just from writing formal specifications, it is important to know if this too is costly.

Lower development costs. It is notoriously difficult to compare the costs of developing software under different methods. There are no figures for the development costs for the same piece of software using both a well-established formal method and a comparable informal method. However, experience on the cost of projects that use formal specification is beginning to accumulate. None of this evidence supports the idea that development costs are higher if you use formal specifications; if anything, it suggests that they are *lower*.

Our own experience on the CASE project showed a productivity (measured in lines of code per day) measured from start of specification to acceptance that was much higher than our normal estimating figure. Because we implemented the CASE project in a productive language (Objective C), the productivity ratio in terms of useful function implemented per day would probably be even higher.

Rolls-Royce and Associates has reported[4] that on a safety-critical project where it used formal specification and planned testing, it achieved better productivity figures than when it did neither. (At first, the productivity was lower, but this was attributed to learning to use various non-user-friendly tools and was not connected with the formal method itself.) The cost of learning to use the formal method was not a significant problem, although IBM has highlighted this learning as an important one-time cost.[3] Rolls-Royce reported that the 7 percent of the time spent on specification avoided large costs at the back end of the project.

Life-cycle changes. Although using formal specification on a project does not cost more, it does change the shape of the project. More time is spent on the specification phase — in the CASE project, about 30 percent of the effort was spent before implementation started. Why? Because more of the work is being done at this stage than typical. But the implementation, integration, and testing phases are shorter.

The longer specification phase does cause a problem: It can be difficult to manage the specification process, because it is harder to see what progress is being made. Especially at the beginning, it can be hard to believe that any progress

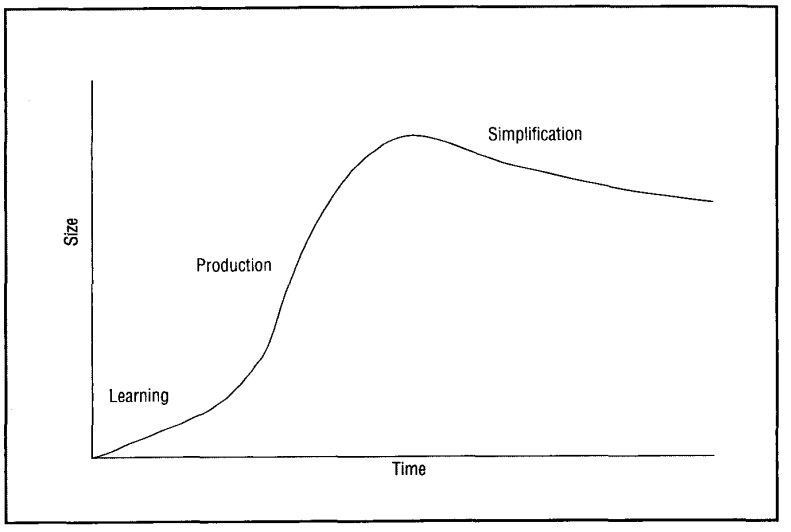


Figure 1. The life history of a specification.

is being made at all, since all sorts of ideas are being tried and thrown away — which is as it should be. Our experience of how the size of a specification grows is shown in Figure 1. (For the actual size of a real specification and the corresponding code, see the box on p. 14.)

At first, very little seems to happen. But after a time, people begin to understand the problem, and rapid progress is made. Then the growth slows down and, if things are going well, the specification starts to get smaller. This is the where the problem is really understood and where regularities and similarities are recognized, which leads to the specification's structure being tightened and improved. This polishing process can continue indefinitely, and a good project manager must know when to stop. He certainly should not stop while the specification is still growing — at that point, the problem is still not fully cracked.

It is important to record the plausible specifications that were tried and rejected, as well as the reasons for their rejection, not just the final specification. These records will help guide future projects, prevent the repetition of unfruitful work, and guide the maintainers.

It is also imperative to recognize that specifications are never perfect. When it comes to the implementation stage, you will find deficiencies in the specification. When this happens, you must modify the specification — under change control, of course. There is a strong temptation to correct the implementation but not the specification — this leads to rapidly increasing divergence between the specification and the actual software and means that the specification becomes useless for maintenance. The two must be kept in step. If you do this, the specification continues to be a valuable document throughout the software's life. Clearly, there is a cost in doing this, but it is not large: On the CASE project, it was less than 5 percent of the implementation phase's effort.

Myth 6

Formal methods are unacceptable to users.

The fact is that formal specifications help users understand what they are getting.

How? The specification captures what the user wants *before* it is built. But to realize this benefit, you must make the specification comprehensible to the user. There are three ways to do this:

- Paraphrase the specification in natural language.
- Demonstrate consequences of the specification.
- Animate the specification.

The first way is always essential. A mathematical specification must be accompanied by a natural-language description that explains what the specification means in real-world terms and why the specification says what it does.

You must allocate time and resources for the effort to write this accompanying text. This effort is worthwhile, since our experience has shown that documents produced from a formal specification can be more comprehensible, more accurate, shorter, and more useful than informal specifications.

A well-produced formal specification can have the mathematics taken out of it entirely — the result is a natural-language document that is a much better specification of the system than a conventional informal specification. You can also use formal specifications with diagrammatic notation — there is nothing to prevent the use of any notation that helps explain the system.

One way that formal specifications are more useful than any other method is that they may let you demonstrate by formal reasoning to the user that the specifications meet certain requirements. You can do this only if the requirements can themselves be expressed formally, but many properties like safety and security can be partially expressed formally. Even if there are no formally expressed requirements, you can draw out certain consequences of the specification and present them to the user. In the CASE project, for example, we deduced (although we did not formally prove) properties like "No version stored on the project machine is ever changed."

Formal specifications are sometimes thought of as antithetical to techniques like animation and prototyping. In fact, the approaches are complementary, and both have the goal of establishing user requirements more reliably. One way to use them together is to build prototypes to explore requirements issues and then to record the results in a formal specification as the basis of subsequent development. Sometimes, you can use prototypes to define areas that are not well expressed in formal specifications. On the CASE project, we used prototyping to explore details of the user interface and formal specifications for the system's actual functions.

You can animate some formal notations, giving you an immediate prototyping capability. However, the more powerful specification languages cannot be executed this way, and so a separate step, like implementation in Prolog, is required to animate the specification.

Myth 7

Formal methods are not used on real, large-scale software.

The fact is that formal methods are used daily on industrial projects.

Several organizations, not just Praxis, are using formal methods on industrial-scale projects. Many people know of applications in the security area, but the scope of formal methods is far wider. Examples of the kinds of project that are using formal methods include the following:

- Transaction processing. IBM's CICS is a large, 20-year-old transaction-processing system. It contains more than a half million lines of code. IBM is using Z to respecify key CICS interfaces to improve its maintainability. So far, Z specifications have been written for more than 100,000 lines of new or changed code.[3]
- Hardware. The use of formal methods is not confined to software. There are at least three examples of the notation Z being used to specify hardware. One is the Secure Multiprocessing of Information by Type Environment secure computer architecture. SMITE's order code has been specified in Z by the British company Plessey. The floating-point unit for the transputer was specified in Z, incidentally revealing errors in many other floating-point implementations.[8] Tektronix has been using Z to specify the functionality of oscilloscope families, as the article on p. 29 describes.
- Compilers. The Danish Datamatik Center has for many years been developing industrial compilers using formal methods.
- Software tools. Our CASE project system is only one, although the most complete, example of the use of formal specification in software tools. Other examples are the interface to the Portable Common Tools Environment,[9] a European standard for software engineering, and the specifications of database-based software-engineering environments.[10]
- Reactor control. Rolls-Royce and Associates used a combination of English and formal specification to specify nuclear-reactor control software.[4] It used animation to explore the specification with the responsible engineer.

Clearly, these projects represent a tiny fraction of all software development. However, they are real industrial-scale applications, and they report positive benefits from the use of formal methods.

Our own experience on the CASE project has been that formal methods can be very effective. But they are only one part of a project: The CASE project used formal specification in the framework of normal quality-assurance and project-management controls and with other good design, implementation, and testing techniques.

Formal methods offer no magic guarantees: Our CASE project was an ordinary project with its share of problems. But the project team believes that the formality of the specification was a major benefit throughout the project.

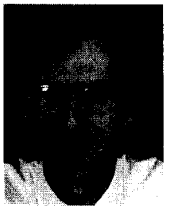
As a result of our experiences, we believe that formal methods must be better understood by developers at large. They are powerful tools, although by no means a panacea. The reasons for their effectiveness are not necessarily the reasons for which they were originally developed. Nor are the difficulties in their use the obvious ones of notation and mathematical sophistication.

Instead of perpetuating the seven myths, I offer seven facts to replace them:

1. Formal methods are very helpful at finding errors early on and can nearly eliminate certain classes of error.
2. They work largely by making you think very hard about the system you propose to build.
3. They are useful for almost any application.
4. They are based on mathematical specifications, which are much easier to understand than programs.
5. They can decrease the cost of development.
6. They can help clients understand what they are buying.
7. They are being used successfully on practical projects in industry. ❖

References

1. J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, Englewood Cliffs, N.J., 1989.
2. O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, Orlando, Fla., 1972, p. 6.
3. C.J. Nix and B.P. Collins, "The Use of Software Engineering, Including the Z Notation, in the Development of CICS," *Quality Assurance*, September 1988, pp. 103-110.
4. J.V. Hill, P. Robinson, and P.A. Stokes, "Safety-Critical Software in Control Systems," in *Computers and Safety*, Inst. Electrical Eng., Stevenage, Herts, England, UK, 1990, pp. 92-96.
5. J.C.P. Woodcock, "Calculating Properties of Z Specifications," *ACM SIGSoft Software-Eng. Notes*, July 1989, pp. 43-54.
6. C.B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, N.J., 1986.
7. D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.
8. G. Barrett, "Formal Methods Applied to a Floating-Point Number System," *IEEE Trans. Software Eng.*, May 1989, pp. 611-621.
9. C.A. Middleburg, "VVSL: A Language for Structured VDM Specifications," *Formal Aspects of Computing*, Jan.-March 1989, pp. 115-135.
10. A.N. Earl et al., "Specifying a Semantic Model for Use in an Integrated Project-Support Environment," in *Software-Engineering Environments*, I. Sommerville, ed., Peregrinus, London, 1986, pp. 202-219.



Anthony Hall is a principal consultant with Praxis Systems. He has worked on software-engineering methods, tools, and environments, as well as developing software applications. His main interest is the application of advanced software-engineering techniques like formal methods and object-oriented development to industrial projects.

Hall has an MA and a DPhil in chemistry from Oxford University and is a member of ACM and the British Computer Society.

Address questions about this article to the author at Praxis Systems, 20 Manvers St., Bath BA1 1PX, England, UK; UUCPnet jah@praxis.co.uk.