SPECIAL REPORT

The Exter

A small British firm shows that software bugs aren't inevitable

By Philip E. Ross

Peter Amey was an aeronautical engineer serving in the United Kingdom's Royal Air Force in the early 1980s when he found a serious flaw in an aircraft missile-control system being deployed at the time. It wasn't a defect in any of the thousands of mechanical and electronic parts that constituted the system's hardware. The problem was in the system's software. Amey found an erroneous piece of program code—a bug. Because of it, the unthinkable could happen: under rare circumstances, a missile could fire without anyone's having commanded it to do so.

Amey says his superiors, rather than commending his discovery, complained that it would delay the system's deployment. Like most project managers, they didn't like the idea of fixing errors at the end of the development process. After all, good design ought to keep errors out in the first place. Yet time and again, Amey knew, the software development process didn't prevent bugs; it merely put off dealing with them until the end. Did it have to be that way? Or could developers avoid bugs in the first place? He would find the answer to be "yes" when, years later, he joined Praxis High Integrity Systems.

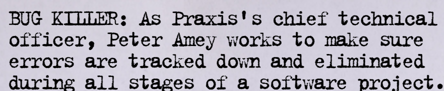
Praxis, headquartered in Bath, 2 hours from London by car, was founded in 1983 by a group of software experts who firmly believed they could put together a sound methodology to ruthlessly exterminate bugs during all stages of a software project.

At the time, the software world was in a malaise that it hasn't fully shaken even today [see "Why Software Fails," in this issue]. Software projects were getting larger and more complex, and as many as 70 percent of them, by some estimates, were running into trouble: going over budget, missing deadlines, or collapsing completely. Even projects considered successful were sometimes delivering software without all the features that had been promised or with too many errors—errors that, as in the missile-firing system, were sometimes extremely serious. The personal computer era, then just starting, only reinforced a development routine of "compile first, debug later."

minators

CHECKING CODE: At Praxis, Peter Amey [left] and Roderick Chapman use mathematical logic to make sure their programs do not contain errors.

ALL PHOTOS: PETER SEARLE



BUG KILLER: As Praxis's chief technical officer, Peter Amey works to make sure errors are tracked down and eliminated during all stages of a software project.

Praxis armed itself not only with an arsenal of the latest software engineering methods but also with something a little more unusual in the field: mathematical logic. The company is one of the foremost software houses to use mathematically based techniques, known as formal methods, to develop software.

Basically, formal methods require that programmers begin their work not by writing code but rather by stringing together special symbols that represent the program's logic. Like a mathematical theorem, these symbol strings can be checked to verify that they form logically correct statements. Once the programmer has checked that the program doesn't have logical flaws, it's a relatively simple matter to convert those symbols into programming code. It's a way to eliminate bugs even before you start writing the actual program.

Praxis doesn't claim it can make bug-free software, says Amey, now the company's chief technical officer. But he says the methodology pays off. Bugs are notoriously hard to count, and estimates of how common they are vary hugely. With an average of less than one error in every 10 000 lines of delivered code, however, Praxis claims a bug rate that is at least 50—and possibly as much as 1000—times better than the industry standard.

Praxis is still a small, lonely asteroid compared to the Jupiter-size companies that dominate the software universe—companies like Microsoft, Oracle, and SAP. The tiny British software house doesn't make products for the masses; it focuses on complex, custom systems that need to be highly reliable. Such mission-critical systems are used to control military systems, industrial processes, and financial applications, among other things.

Sometimes the software needs to work 99.999 percent of the time, like an air-traffic control program Praxis delivered some years ago. Sometimes it needs to be really, really secure, like the one Praxis recently developed for the National Security Agency, the supersecret U.S. signals intelligence and cryptographic agency, in Fort Meade, Md.

And though Praxis employs just 100 people, its name has become surprisingly well known. "They're very, very talented, with a very different approach," says John C. Knight, a professor of computer science at the University of Virginia and the editor in chief of *IEEE Transactions on Software Engineering*. Praxis's founders, he says, believed that building software wasn't as hard as people made it out to be. "They thought, it isn't rocket science, just very careful engineering."

Watts S. Humphrey, who once ran software development at IBM and is now a fellow at the Software Engineering Institute at Carnegie Mellon University, in Pittsburgh, also speaks highly of Praxis. He says the company's methodology incorporates things like quality control that should be more widely used in the field. In fact, Humphrey spent this past summer at Praxis headquarters to learn how they do things. He wants to use that knowledge to improve a complementary methodology he developed to help organizations better manage their software projects.

Praxis's approach, however, isn't perfect and isn't for everybody. Formal methods obviously are no silver bullet. For one thing, using formal methods can take more time and require new skills, all of which can mean higher up-front costs for a client. In fact, Praxis charges more—50 percent more in some cases—than the standard daily rate. To this its engineers will say: "You get what you pay for; our bug rate speaks for itself."

And although formal methods have been used to great effect in small and medium-size projects, no one has yet managed to apply them to large ones. There's some reason to think no one ever will, except perhaps in a limited fashion. Nevertheless, even though the company may not have all the answers to make software projects more successful, those working in the field can learn plenty of lessons from it, say advocates like Knight and Humphrey.

SOFTWARE WAS CONCEIVED AS a mathematical artifact in the early days of modern computing, when British mathematician Alan Turing formalized the concept of algorithm and computation by means of his now famous Turing Machine, which boils the idea of a computer down to an idealized device that steps though logical states.

But over time, software development gradually became more of a craft than a science. Forget the abstractions and the mathematical philosophers. Enter the realm of fearless, caffeinated programmers who can churn out hundreds of lines of code a day (often by hastily gluing together existing pieces of code). The problem is that for some projects, even tirelessness, dedication, and skill aren't good enough if the strategy is wrong.

Large, complex software systems usually involve so many modules that dealing with them all can overwhelm a team following an insufficiently structured approach. That's especially true of the mission-critical applications Praxis develops, as well as of large enterprise resource-planning systems, of the sort used by Fortune 500 companies, and complex data-driven software, such as the FBI's Virtual Case File project [see "Who Killed the Virtual Case File?" in this issue].

Even when you break such a big program down into small, seemingly manageable pieces, making a change to one turns out to affect 10 others, which may in turn affect scores or maybe even hundreds of other pieces. It may happen that making all the fixes will require more time and money than you have. If your system's correct operation depends on those changes, you'll have to either admit defeat or scramble to find a way to salvage what you've done so far, perhaps by giving up on some of the capabilities or features you'd hoped to have in the software.

As it turns out, complete failure—projects canceled before completion—is the fate of 18 percent of all information technology projects surveyed in a 2004 study by consultancy Standish Group International Inc., in West Yarmouth, Mass. Apparently that's the good news; the rate 10 years ago, according to Standish, was 31 percent.

Still, the overall picture is pretty bleak. Standish asserts that more than 50 percent of the thousands of projects it surveyed faced problems, from being turned over without significant features to going well beyond their deadlines or budgets. In the end, according to the Standish numbers, only 28 percent of projects could be considered successes by any rigorous definition.

Standish's numbers, however, are far from universally accepted in the computer industry. For contract software projects, more specifically, other analyses in recent years have put the success rate as low as 16 percent and as high as 62 percent. Nevertheless, even using those numbers as a guide, it's hard not to see the contract software business as anything but an enterprise all too often mired in mediocrity. As one study by consultant Capers Jones, in Marlborough, Mass., put it: "Large software systems...have one of the highest failure rates of any manufactured object in human history."

Today, ever more sophisticated tools are available to help companies manage all aspects of their software projects. These tools help conceptualize and design the system; manage all people, files, computers, and documents involved; keep track of all versions and changes made to the system and its modules; and automate a number of tests that can be used to find system errors.

Indeed, worldwide sales of such software development tools, according to Stamford, Conn.–based market research firm Gartner Inc., generate more than US $3 billion a year. Rational Software Corp., a company acquired by IBM in 2002 for $2.1 billion, is currently the market leader, followed by Microsoft, Computer Associates International, Compuware, Borland, and others, according to Gartner.

But the effect of widespread use of these tools on overall software quality hasn't been gauged in a detailed or rigorous way. Some would even argue that the sector is a little reminiscent of the market for diet products: it, too, is a billion-dollar industry, and yet, somehow, obesity as a public health problem hasn't gone away. And, just as the few successful diet strategies all seem to require a major change in lifestyle, perhaps, too, the software failure rates won't improve significantly without a basic and widespread shift in tactics.

Certainly, Praxis's experience supports that idea. Consider one of the company's recent projects, for Mondex International Inc., a financial services company founded in the UK that is now a subsidiary of MasterCard International Inc. First, a little background. Mondex had a product called an electronic purse, a credit card–like payment card that stored electronic cash. That is, it did not debit a bank account or draw on a line of credit; it stored the cash digitally in an embedded chip. Mondex wanted to make the card flexible enough to run a variety of applications that would keep track not only of electronic cash but also of discount coupons, loyalty reward points, and other items still unimagined.

The critical issue was to make sure that only cards with legitimate applications would work; any other card, even if programmed to pass as a Mondex card, would be deemed invalid. The solution Mondex chose was to use a special program, known as a certification authority, that would run on a central computer at the company's headquarters. The certification authority would generate unique digital certificates—long strings of numbers—to accompany all applications on the cards. That way, a card reader at, say, a store could validate a card's certificates by running them through a series of mathematical operations that would prove unequivocally that they came from Mondex.

Mondex hired Praxis to develop the certification authority, which was the most critical part of the whole system. After all, if the security of one card was broken, then just that one card could be forged. But compromising the certification authority would allow mass forgery of cards.

THE PRAXIS TEAM BEGAN WORKING on the solution in late 1998. The first step was to hammer out what, precisely, the Mondex system was supposed to do—in software jargon, the system's requirements. These are essentially English-language bullet points that detail everything the program will do but not how it will be done.

Getting the requirements right is perhaps the most critical part of Praxis's methodology. For that reason, Praxis engineers held many exhaustive meetings with the people from Mondex, during which they tried to imagine all possible scenarios of what could happen. As Praxis does for all its projects, it insisted that Mondex make available not only its IT people but everyone who would have any involvement with the product—salespeople, accountants, senior managers, and perhaps even the CEO. "We focus very hard on identifying all stakeholders, everybody that cares," says Roderick Chapman, a principal engineer at Praxis.

To make sure Praxis was on target with the system requirements, it devised a prototype program that simulated the graphical interface of the proposed system. This prototype had no real system underlying it; data and commands entered through the interface served only to check the requirements. In fact, Praxis made no further use of the prototype—the real graphical interface would be

SPARK MAKER: Roderick Chapman, a principal engineer at Praxis, says that Spark, the programming language the company created, helps avoid many serious bugs.

BUGPROOF CODE

Praxis High Integrity Systems uses mathematical logic to check that its programs are free from bugs. You can get the gist of how the company does that by following this simple example.

Suppose a Praxis programmer needs a piece of code to add two numbers, *a* and *b*, and multiply that sum by a third number, *c*. The first thing to do is to describe that calculation using Z, a formal specification language that spells out the program's logic. In the language of Z, that simple operation looks like this:

$$Calculation : Number \times Number \times Number \rightarrow Number$$
$$\forall\, a, b, c : Number \bullet$$
$$Calculation\ (a, b, c) = (a + b) * c$$

Next, the programmer converts this Z specification into Spark, a programming language created by Praxis. To allow the programmer to more easily spot bugs, Spark code has two parts. The first part is essentially a refinement of the Z specification:

```
function Calculation(A, B, C : in Number) return Number;
--# return (A + B) * C;
```

The second part of the Spark code is the executable portion that effectively makes the calculation. But note that the second part contains a bug: the expression (A + B) * C is mistakenly written as (A + B * C):

```
function Calculation(A, B, C : in Number) return Number
is
begin
return (A + B * C);
end Calculation;
```

The Praxis programmer would catch the bug by verifying—either by eye or through the use of special verification software—that the first part of the Spark code doesn't match the second part. The verification software also checks a number of other conditions to make sure the calculation won't cause errors, like a memory buffer overflow or a division by zero.

The program discussed in this example is extremely simple, with just a few lines of code, so any programmer could easily spot the bug without the help of mathematical methods. But Praxis constructs programs with tens of thousands of lines of code containing complex logical operations, and in such cases Z and Spark are invaluable tools for spotting—and killing—bugs. *—P.E.R.*

developed later, using much more rigorous methods. In following this approach, Praxis was complying with an edict from Frederick P. Brooks's 1975 classic study of software development, *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, 2nd edition, 1995):

> *In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter and build a redesigned version in which these problems are solved. The discard and redesign may be done in one lump, or it may be done piece-by-piece. But all large-system experience shows that it will be done....*
>
> *Hence plan to throw one away; you will, anyhow.*

Once Praxis's engineers had a general idea of what the system would do, they began to describe it in great detail, in pages and pages of specifications. For example, if a requirement said that every user's action on the system should produce an audit report, then the corresponding specification would flesh out what data should be logged, how the information should be formatted, and so on.

This is the first math-intensive phase, because the specifications are written mostly in a special language called Z (pronounced the British way: "zed"). It's not a programming language—it doesn't tell a computer how to do something—but it is a formal specification language that expresses notions in ways that can be subjected to proof. Its purpose is simple: to detect ambiguities and inconsistencies. This forces engineers to resolve the problems right then and there, before the problems are built into the system.

Z, which was principally designed at the University of Oxford, in England, in the late 1970s and early 1980s, is based on set theory and predicate logic. Once translated into Z, a program's validity can be reviewed by eye or put through theorem-proving software tools. The goal is to spot bugs as soon as possible [see sidebar, "Bugproof Code"].

The process is time-consuming. For the Mondex project, spec-writing took nearly a year, or about 25 percent of the entire development process. That was a long time to go without producing anything that looks like a payoff, concedes Andrew Calvert, Mondex's information technology liaison for the project. "Senior management would say: 'We are 20 percent into the project and we're getting nothing. Why aren't we seeing code? Why aren't we seeing implementation?' " he recalls. "I had to explain that we were investing much more than usual in the initial analysis, and that we wouldn't see anything until 50 percent of the way through." For comparison, in most projects, programmers start writing code before the quarter-way mark.

Only after Praxis's engineers are sure that they have logically correct specifications written in Z do they start turning the statements into actual computer code. The programming language they used in this case, called Spark, was also selected for its precision. Spark, based on Ada, a programming language created in the 1970s and backed by the U.S. Department of Defense, was designed by Praxis to eliminate all expressions, functions, and notations that can make a program behave unpredictably.

By contrast, many common programming languages suffer from ambiguity. Take, for example, the programming language C and the expression "i++ * i++," in which "*" denotes a multiplication and "++" means you should increment the variable "i" by 1. It's not an expression a programmer would normally use; yet it serves to illustrate the problem. Suppose "i" equals 7. What's the value of the expression? Answer: it is not possible to know. Different compilers—the special programs that transform source code into instructions that microprocessors can understand—would interpret the

expression in different ways. Some would do the multiplication before incrementing either "i," giving 49 as the answer. Others would increment the first "i" only and then do the multiplication, giving 56 as the answer. Yet others would do unexpected things.

Such a problem could not happen in Spark, says Praxis's Chapman, because all such ambiguous cases were considered—and eliminated—when the language was created. Coding with Spark thus helps Praxis achieve reduced bug rates. In fact, once Spark code has been written, Chapman says, it has the uncanny tendency to work the first time, just as you wanted. "Our defect rate with Spark is at least 10 times, sometimes 100 times lower than those created with other languages," he says.

"WE ARE **20 PERCENT** INTO THE PROJECT AND WE'RE GETTING NOTHING. WHY AREN'T WE SEEING **CODE?**"

Peter Amey explains that the two-step translation—from English to Z and from Z to Spark—lets engineers keep everything in mind. "You can't reason across the semantic gap between English and code," he says, "but the gap from English to an unambiguous mathematical language is smaller, as is the gap from that language to code."

What's more, Spark lets engineers analyze certain properties of a program—the way data flows through the program's variables, for example—without actually having to compile and run it. Such a technique, called static analysis, often lets them prevent two serious software errors: using uninitialized variables, which may inject spurious values into the program, and allocating data to a memory area that is too small, a problem known as buffer overflow.

In practice, though, not everything can be put through the mathematical wringer. Problems with the way different modules exchange data, for instance, by and large have to be solved the old-fashioned way: by thinking. Nor can Praxis completely eliminate classic trial-and-error testing, in which the programmers try to simulate every situation the software is likely to confront.

But what Praxis does do is make such simulation a last resort, instead of the main line of defense against bugs. (As famed computer scientist Edsger Dijkstra wrote, "Program testing can be used to show the presence of bugs, but never to show their absence!") For the Mondex project, such testing took up 34 percent of the contract time. That's in the lower end of the usual share, which typically ranges from 30 to 90 percent. Reduced efforts on testing mean huge savings that go a long way toward balancing the extra time spent on the initial analysis.

The system went live in 1999. Though it cost more up front, the contract called for Praxis to fix for free any problem—that is, any deviation from the specification—that came up in the first year of operation, a guarantee rarely offered in the world of contract software. That first year, just four defects triggered the clause. According to Chapman, three of the problems were so trivial that they took no more than a few hours to correct. Only one was functionally significant; it took two days to fix. With about 100 000 lines of code, that's an average of 0.04 faults per 1000 lines. Fault rates for projects not using formal methods, by some estimates, can vary from 2 to 10 per 1000 lines of code, and sometimes more.

For Mondex, fewer bugs meant saving money. Calvert estimates that Mondex will spend 20 to 25 percent less than the norm in maintenance costs over the lifetime of the project.

FORMAL METHODS WERE RELATIVELY NEW when Praxis started using them, and after some ups and downs, they have recently been gaining popularity. Among their leading proponents are John Rushby at SRI International, Menlo Park, Calif.; Constance Heitmeyer, at the U.S. Naval Research Laboratory's Center for High Assurance Computer Systems, Washington, D.C.; Jonathan Bowen at London South Bank University; the developers of Z at the University of Oxford and other institutions; and the supporters of other specification languages, such as B, VDM, Larch, Specware, and Promela.

In recent years, even Microsoft has used formal methods, applying them to develop small applications, such as a bug-finding tool used in-house and also a theorem-proving "driver verifier," which makes sure device drivers run properly under Windows.

But still, the perceived difficulty of formal tools repels the rank-and-file programmer. After all, coders don't want to solve logical problems with the help of set theory and predicate logic. They want to, well, code. "Few people, even among those who complete computer science degrees, are skilled in those branches of pure mathematics," says Bernard Cohen, a professor in the department of computing at City University, in London.

In every other branch of engineering, he insists, practitioners master difficult mathematical notations. "Ask any professional engineer if he could do the job without math, and you'll get a very rude reply," Cohen says. But in programming, he adds, the emphasis has often been to ship it and let the customer find the bugs.

Until formal methods become easier to use, Cohen says, Praxis and companies like it will continue to rely on clients' "self-selection"—only those users who are highly motivated to get rock-solid software will beat a path to their door. Those that need software to handle functions critical to life, limb, national security, or the survival of a company will self-select; so will those that are contractually obligated to meet software requirements set by some regulator. That's the case with many military contractors that now need to demonstrate their use of formal methodologies to government purchasers; the same goes for financial institutions. Mondex, for instance, required the approval of the Bank of England, in London, and formal methods were part of that approval.

Yet even if regulators were omnipresent, not all problems would be amenable to formal methods, at least not to those that are available now. First, there is the problem of scaling. The largest system Praxis has ever built had 200 000 lines of code. For comparison, Microsoft Windows XP has around 40 million, and some Linux versions have more than 200 million. And that's nothing compared with the monster programs that process tax returns for the U.S. Internal Revenue Service or manage a large telecom company's infrastructure. Such systems can total hundreds of millions of lines of code.

What does Praxis say about that? "The simple answer is, we've never gone that big," says Chapman. "We believe these methods should scale, but we have no evidence that they won't or that they will." So what if a client approaches Praxis with a really big project? Would the company handle it? "The key weapon is abstraction," he says. "If you can build abstractions well enough, you should be able to break things down into bits you can handle." That maxim guides every other discipline in engineering, not least the design of computer hardware. Why not apply it to software, too? ■

ABOUT THE AUTHOR

PHILIP E. ROSS (IEEE Member) wrote "Managing Care Through the Air" for the December 2004 issue of *IEEE Spectrum*. His work has also appeared in *Scientific American*, *Forbes*, and *Red Herring*.