Investigating the Influence of Formal Methods

Formal methods promise much, but can they deliver? In this project, results are inconclusive, but careful data gathering and analysis helped establish influences on product quality.

Shari Lawrence Pfleeger
Systems/Software Inc. and Howard University

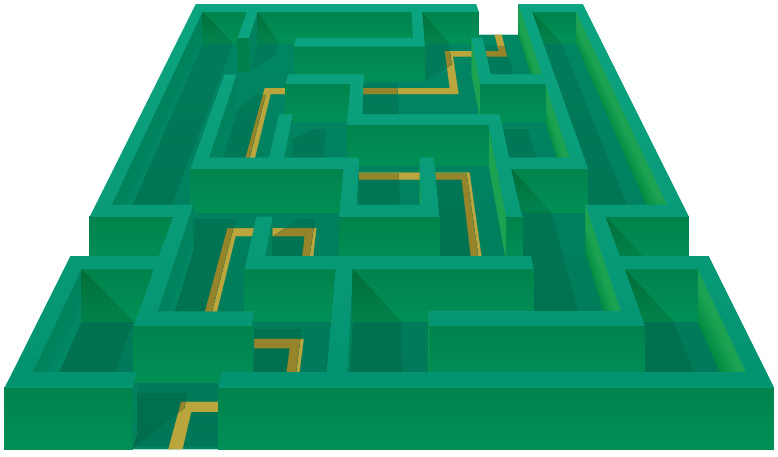
Les Hatton
Oakwood Consultancy

Practitioners and researchers continue to seek methods and tools for improving software development processes and products. Candidate technologies promise increased productivity, better quality, lower cost, or enhanced customer satisfaction. But we must test these methods and tools empirically and rigorously to determine any significant, quantifiable improvement. We tend to consider evaluation only after using the technology, which makes careful, quantitative analysis difficult if not impossible. However, when an evaluation is designed as part of overall project planning, and then carried out as software development progresses, the result can be a rich record of a tool's or technique's effectiveness.

In this study, we investigated the effects of using formal methods to develop an air-traffic-control information system. Because we are studying one project in isolation, we cannot draw conclusions about the suitability of formal methods for all projects. As we describe in the sidebar "Can Formal Methods Always Deliver?" the jury is still out on when and whether formal methods improve products. Nevertheless, the lessons we learned are instructive, not only in showing how formal methods influenced code quality on this project, but also in highlighting the limitations of retrospective studies and their use in planning follow-up investigations.

We describe what we did, as well as what we could have done had the study been carried out as the software system was being developed and tested. We also show how this preliminary investigation helps to suggest hypotheses for further studies. Thus, the lessons we learned can be applied not only to gauge the effects of formal methods but also in planning similar studies of other techniques and tools.

The procedure we used was not predetermined; the results of one analysis step largely determined where we went next. Indeed, research often involves following one trail and then another, uncovering relationships and unearthing facts, until the picture begins to make sense. However, we did learn many specific lessons, which we hope will enrich future investigations.



PROJECT DESCRIPTION

In the early 1990s, Praxis built an air-traffic-control information system for the UK Civil Aviation Authority using a variety of formal methods. The motivation for using particular formal methods and a general description of the system's function and architecture are in Anthony Hall's recent work.[1] The system, CDIS (Central Control Function Display Information System), provides controllers at the London Air Traffic Control Centre with key information for one of the busiest airspaces in the world. It is composed of nearly 200,000 lines of C code.

Use of formal methods: requirements

Praxis had used formal methods before, but not to the extent used in CDIS.[1] The team developed functional requirements using three techniques:

- an entity-relationship analysis to describe real-world objects, their properties, and their interre-

0018-9162/97/$10.00 © 1997 IEEE

Can Formal Methods Always Deliver?

Norman Fenton
Centre for Software Reliability, City University

Shari Lawrence Pfleeger
Systems/ Software Inc. and Howard University

Formal methods mean different things to different people, and there is no shortage of methods. Foundationally, each formal technique involves the use of mathematically precise specification and design notations, and each (in its purest form) is based on refinement and proof of correctness at each stage in the life cycle. However, all formal methods are not created equal, and it is misleading to lump them in a single category to decide if they will improve a product.

Some organizations are reluctant to move from their current practices to practices that include formal methods because there is often a radical change at the very beginning of the software life cycle: the capture and recording of customer requirements. In these cases, compelling evidence of the effectiveness of formal methods, such as the case study described in the main text, can make all the difference.

Unfortunately, past evaluations of the use of formal methods are inconclusive. The few serious industrial uses of formal methods focused on formal specification alone, with no widespread attempt at formal deduction, refinement, or proof. In fact, the Paris metro is the only documented case in which formal methods were used throughout development. Susan Gerhart, Dan Craigen, and Ted Ralston extensively survey the use of formal methods in industrial environments,[1] reporting that

> There is no simple answer to the question: Do formal methods pay off? Our cases provide a wealth of data but only scratch the surface of information available to address these questions. All cases involve so many interwoven factors that it is impossible to allocate pay off from formal methods versus other factors, such as quality of people or effects of other methodologies. Even where data was collected, it was difficult to interpret the results across the background of the organization and the various factors surrounding the application.

There is even some evidence in Peter Naur's work[2] suggesting that formal methods do not necessarily assure high quality. For example, Naur reports that the use of formal notations does not lead inevitably to improving the quality of specifications, even when used by the most mathematically sophisticated minds. In his experiment, the use of a formal notation often led to a greater number of defects, rather than fewer. Thus, we need careful analyses of the effects of formal methods to understand what contextual and methodological characteristics affect the end results.

Meanwhile, anecdotal evidence of the positive effects of formal methods continues to grow. Susan Gerhart, Dan Craigen, and Ted Ralston described several instances of their use for safety-critical systems in early 1994.[3] And Iain Houston and Steven King[4] reported on a joint project between IBM Hursley and the Programming Research Group at Oxford University. A serious attempt was made to quantify the benefits of using Z on the CICS respecification project, and a proceedings paper provides sanitized graphs and general information. As a result, CICS is widely believed to provide the best quantitative evidence to support the efficacy of formal methods (an observation also confirmed in the Gerhart study). However, the public announcements of success have never been accompanied by a complete set of data and analysis, so independent assessment is difficult.

As anecdotal support for formal methods has grown, practitioners have been more willing to use formal methods on projects that involve safety-critical software. For example, John McDermid[5] asserts that "these mathematical approaches provide us with the best available approach to the development of high-integrity safety-critical systems." Formal methods are being incorporated into standards and imposed on developers. For instance, the interim UK defense standard for such systems, DefStd 00-55, mandates the use of formal methods. Jonathan Bowen and Michael Hinchey[6] report that other countries are following suit, with some countries requiring formal methods and others strongly recommending them.

Such standards formulation without a solid basis of empirical evidence can be dangerous and costly. As Norman Fenton, Shari Lawrence Pfleeger, and Robert Glass point out,[7] there is still no hard evidence to show that

- Formal methods have been used cost-effectively on a realistic safety-critical system development.
- The use of formal methods can deliver reliability more cost-effectively than, say, traditional structured methods with enhanced testing.
- Either developers or users can ever be trained in sufficient numbers to use formal methods properly.

Moreover, we must understand how to choose among the many competing formal methods, which may not be equally effective in a given situation.

References

1. S. Gerhart, D. Craigen, and T. Ralston, "Observation on Industrial Practice Using Formal Methods," *Proc. 15th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 24-33.
2. P. Naur, "Understanding Turing's Universal Machine—Personal Style in Program Description," *Computer J.*, No. 4, 1993, pp. 351-371.
3. S. Gerhart, D. Craigen, and T. Ralston, "Experience with Formal Methods in Critical Systems," *IEEE Software*, Jan. 1994, pp. 21-28.
4. I. Houston and S. King, "CICS Project Report: Experiences and Results from the Use of Z," *Proc. VDM '91: Formal Development Methods*, Springer-Verlag, New York, 1991.
5. J. McDermid, "Safety-Critical Software: A Vignette," *IEE Software Eng. J.*, No. 1, 1993, pp. 2-3.
6. J. Bowen and M. Hinchey, "Formal Methods and Safety-Critical Standards," *Computer*, Aug. 1994, pp. 68-71.
7. N. Fenton, S. Pfleeger, and R.. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, July 1994, pp. 86-95.

lationships (such as arrivals, departures, workstations, displays);
- a real-time extension of Yourdon-Constantine structured analysis to define the processing requirements; and
- a formal specification language to define the data and operations.

In addition, to ensure that the development team understood crucial parts of the system before development began, Praxis chose to express the requirements for critical system elements using the Vienna Development Method. To enhance their understanding even more, the development team used a different formal method to document distinct points of view: required functionality, using a formal language similar to VDM; user interface, using pictures, text and state-transition diagrams (considered a formal method); and concurrency, using Milner's calculus of communicating sequential processes (CCS).

Use of formal methods: design

The system design was produced by separate teams of varying sizes, a fact that surfaced as we tried to understand differences in design quality. The design was expressed in four parts, connected by a design overview document:

- *Application code*. Ten contributing developers wrote VDM specifications, created as refinements of the core specification.
- *Concurrency.* One developer used finite state machines (FSM) to define concurrency and invoke the application code.
- *Local area network*. Two developers used a mix of VDM and CCS. Because the LAN software was particularly difficult to design, they also did some formal proofs to uncover faults in the design.
- *User interface code.* Four developers used pseudocode to describe the user interface.

Thus, we can categorize any code module in CDIS as being influenced by one of four design types: VDM, FSM, VDM/CCS, or informal.

DATA ANALYSIS

The CDIS development team had kept careful records of all faults reported during in-house system testing as well as after delivery. Comments from the Civil Aviation Authority and CDIS users were very positive, and the system appeared to be stable and reliable. But Praxis wanted to know more about how the use of formal methods affected the quality of the code. In concert with researchers at the Centre for Software Reliability, Praxis opened their records for scrutiny, and we analyzed the data to try to determine the effects of formal methods. To do so, we took several steps to understand the meaning of the data and the relationships among data elements. Our goal was to find evidence of how formal methods affected code quality, if indeed they did at all.

Praxis opened their records for scrutiny, and we analyzed the data to try to determine the effects of formal methods.

Step 1: Understand the data

Analysts can express software quality in several ways. Usually, they first distinguish faults (errors that the developer sees) from failures (errors that the user sees). They then express quality in terms of faults and failures, often normalized by a measure of size, such as lines of code. Studies have shown that some faults never lead to failures (for example, when the faulty code is never executed); for this reason, we define reliability in terms of time between failures, rather than faults.

Figure 1 on the next page shows a sample of the fault reports Praxis used in CDIS development. As a problem was identified, the team assigned a fault report number, as well as a category to describe the likely source of the problem: specification, design, or code. They also briefly described the problem, adding suggestions about which code modules, specification documents, or design components might be the source. According to categories defined by the Civil Aviation Authority, they also assigned a severity designation of 1, 2, or 3:

- *Category 1. Operational system critical.* Includes corrupted or lost data at any stage of processing or presentation, processing unable to meet response times or capacity constraints, halt or interruption of service, or failure of any part of the system to meet stated reliability requirements (in terms of mean time to failure and mean time to repair).
- *Category 2. System inadequate.* Includes noncompliance or omission of air-traffic-control operational function, omission of function as detected in a physical configuration audit, or any Category 1 failure occurring during acceptance testing or training.
- *Category 3. System unsatisfactory.* Includes noncompliance or omission of non-air traffic-control support and maintenance functions, nonadherence to standards, layout or format errors, or inconsistency or omission of documentation.

After fixing the problem, the responsible party listed all documents and modules that actually changed and the new severity designation, if it changed. (On closer inspection, some failures were often more or less severe than reported.) The team added a designation category of 0 to indicate that the perceived problem was not a problem at all or had been reported previously.

As the figure shows, the Praxis fault reports are really failure reports; they describe problems experi-

Figure 1. Sample CDIS fault report.

CDIS FAULT REPORT **S.P0204.6.10.3016**

ORIGINATOR:	Joe Bloggs
BRIEF TITLE:	Exception 1 in dps_c.c line 620 raised by NAS
FULL DESCRIPTION	Started NAS endurance and allowed it to run for a few minutes. Disabled the active NAS link (emulator switched to standby link), then re-enabled the disabled link and CDIS exceptioned as above. (I think the re-enabling is a red herring.) (during database load)

ASSIGNED EVALUTION TO: DATE:

CATEGORISATION: 0 (1) 2 3 Design Spec Docn
SEND COPIES FOR INFORMATION TO:
EVALUATOR: DATE: 8/7/92

CONFIGURATION ID	ASSIGNED TO	PART
dpo_s.c		

COMMENTS: dpo_s.c appears to try to use an invalid CID, instead of rejecting the message. AWJ

ITEMS CHANGED

CONFIGURATION ID	IMPLEMENTOR/DATE	REVIEWER/DATE	BUILD/ISSUE NUM	INTEGRATOR/DATE
dpo_s.c v.10	AWJ 8/7/92	MAR 8/7/92	6.120	RA 8-7-92

COMMENTS:

CLOSED

FAULT CONTROLLER: DATE: 9/7/92

We analyzed some 3,000 fault reports generated from 1990 to June 1992.

enced by the testers or the users. The team took great care to capture data on these failures and their fixes. Because of time constraints, they did little causal analysis before delivery, but they continued to note and track problems after delivery, finding less than one problem per thousand lines of delivered code.

We analyzed some 3,000 fault reports generated from 1990 to June 1992 (delivery), considering only those designated with 1, 2, or 3. (We eliminated 910 fault reports designated 0.) We then partitioned the data so that we could separately analyze problems arising from code and those arising from specification or design problems. (The team typically did not count predelivery changes that affected test code, although they recorded some such faults accidentally.) We also had to consider changes affecting configuration management files and documents (because of requirements in the severity categories and other contract obligations).

The fault report lists only the modules changed to fix a problem. Rarely was a root cause clearly identified. Because of this, we decided to measure the effect of a problem by counting the number of modules changed. Similarly, because we did not know the size of each module (we knew only total lines of code for each design type), we normalized our data by the number of modules in each category that was being compared. We also classified each module and document by the type of design that influenced it: VDM, VDM/CCS, FSM, or informal.

Step 2: Look for differences in the number of changes

We sought answers to two questions:

- Did formal methods quantitatively affect code quality?
- Was one formal method superior to another?

Because the fault reports told us which modules were changed, we began by looking for differences in the number of changes; if one design type's modules had changed fewer times than the others', we might have clear evidence of that design type's superiority.

Table 1 presents general data on the size and quality of the code developed from each design type. The first row shows the number of delivered lines of code from each type. The second row quantifies the number of code changes resulting from fault reports that affected code of each type. If a single fault report resulted in a change to more than one module, we counted each module change separately, so the total

code changes are more than the total fault reports. The third row divides the second row by the first row to get a normalized measure of the relative number of changes in each design category. When taken with the formal types as an aggregate, the changes to informally designed delivered modules are not significantly higher or lower than changes to formally designed modules (21.0 vs. 19.6).

The last three rows look at the same data in terms of number of modules. Fewer VDM/CCS modules changed overall. Code developed using VDM alone required the most module changes. We performed a similar analysis for the extra code used in development and testing but not delivered to the customer, and then code for the entire system (delivered code and extra code). The results did not differ significantly from those in Table 1.

If you measure quality strictly by the changes needed to correct modules, our results do not clearly indicate that formal design methods produced higher quality code than informal ones. However, we noticed a different relationship: For the formally designed modules, the fewer the developers, the fewer the faults. Unfortunately, because the project team had been disbanded, and many team members were assigned to new projects outside the local area, we could not investigate the relationship between teams and quality. To do so, we would have had to gather information about team member experience (with languages, methods, tools and application domains) and training, as well as about the organization of and communication among team members. Such information would have helped us decide whether team size (or something else) affected code quality.

To determine if the design method affected the results for modules that required an extreme number of changes, we narrowed our analysis first to delivered modules that required more than five changes each, and then more restrictively to those that required more than 10 changes each, as a result of fault reports. Table 2 presents the data for this case.

Again, VDM/CCS has the highest quality compared with the other design methods. Informally designed modules required fewer changes than those designed using formal methods, but as with Table 1, the overall difference between informal and formal methods is insignificant. Further analysis normalizing the modules by size yields similar results.

We also had data to tell us which documents (including specification, design, and code-related ones) had been changed because of a problem report. Each document could be related to a design type, so we analyzed the relationship between document changes and design type. Again, we found no conclusive evidence that any design type was superior.

Step 3: Look for trends

Because we found no compelling evidence by examining the problem data statically, we decided to look at problems reported over time. We grouped fault reports by quarter so that we could examine the proportion of changes made by design type during that time. As before, we counted each code change that resulted from a fault report, not simply the number of fault reports. Thus, the total number of changes represented in the graph is far more than the number of fault reports. These are changes to what would be delivered code, not to all code. (Some faults may have required changes only in specification and design but not in the code itself.) We did not include changes to

Table 1. Relative sizes and changes reported for each design type in delivered code.

	FSM	VDM	VDM/CCS	Total formal	Informal
Total lines of delivered code	19,064	61,061	22,201	102,326	78,278
Number of changes in delivered code caused by fault reports	260	1,539	202	2,001	1,644
Code changes per thousand lines of code	13.6	25.2	9.1	19.6	21.0
Number of modules with this design type	67	352	82	501	469
Total number of modules changed	52	284	57	393	335
Percentage of delivered modules changed	78	81	70	78	71

Table 2. Code changes by design type for modules requiring many changes.

	FSM	VDM	VDM/CCS	Total formal	Informal
Total number of modules changed	58	284	58	400	556
Number of modules with more than five changes	11	89	11	111	108
Percentage of modules changed	16	25	13	22	19
Number of modules with more than 10 changes	8	35	3	46	31
Percentage of modules changed	12	19	4	9	7

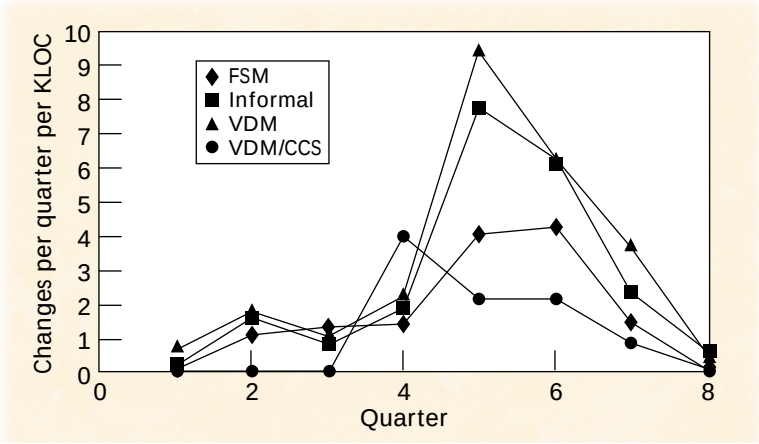


Figure 2. Changes reported by quarter, normalized by size of code.

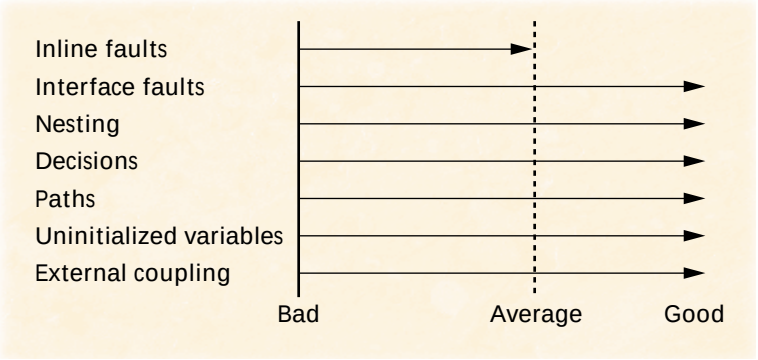


Figure 3. Summary graph of software quality.

any extra code needed for configuration management or testing. Again, to account for differences in size, we normalized the counts by the number of lines of code in each design type.

Figure 2 shows the results of the trend analysis over two years. Because the graph represents all testing done before delivery, the spikes at or after quarter 4 probably represent changes that occurred at the onset of system testing. These late changes well into the development cycle are not desirable. The spikes seem to be moderate for code designed using VDM/CCS and FSM. The spikes are very large and relatively late for the informally designed and VDM modules.

However, these differences may have nothing to do with the design method, but rather may reflect the organization of the development team. The FSM and VDM/CCS code were developed by fewer people (one and two, respectively) than the other two kinds. The larger spikes may reflect the more complex communication problems that occur when larger groups develop code together.

Moreover, the spike for the VDM/CCS code occurs earlier than for the others, and the number of changes drops soon thereafter. This suggested to us that faults are found and fixed earlier in that type of code and that something other than system testing may be responsible. We wanted to investigate further to determine the cause of the spikes, but we had no additional data about testing and other project activities to enlighten us. We needed to take a different approach.

Step 4: Conduct a code audit

So far, we had focused on evidence of the *behavior* of the code, but we had not examined the code itself. Because we had access to tools that reveal code structure, we supplemented our previous analysis with a static, structural look at the delivered code. CDIS code is written almost entirely in C, so we used QAC to detect reliance on unsafe features of C as documented in Les Hatton's guidelines for developing safety-critical systems.[2]

The code was supplied to us in three separately compilable pieces (not corresponding to the four-part design described above), and we audited each piece separately. The audit had two parts:

- Analyze each module for potential faults remaining.
- Calculate several structure and dependence measures to compare the modules with the database from Programming Research, which contains the results of audits for millions of lines of C in a variety of application domains, including safety-related systems.

The first part of the audit produced a listing of potential problems, organized in six categories:

- Syntax and constraint violations of a conforming C compiler.
- In-line and interface faults. Inconsistencies in the way the language elements have been combined.
- Reliance on imprecisely defined features of C. Behavior is either unspecified (legal but not defined), undefined (illegal but not defined), implementation-defined (well-defined but dependent on a particular compiler), or location-specific.
- Potential reliance on uninitialized variables.
- Reliance on implicit narrowing conversions and implicit conversions between signed and unsigned behavior.
- Clutter. Unused variables and unreachable code.

The results of the first part of the audit were not immediately useful, but we used them in a later step of our investigation.

The second part of the audit evaluated component and system complexity. Using software measurements such as cyclomatic number, static path count,[3] and maximum depth of nesting, the tool compared the CDIS code with assembled population distributions for large amounts of code from many application areas;

Table 3. Faults discovered during unit testing.

	FSM	VDM	VDM/CCS	Total formal	Informal
Number of faults discovered	43	184	11	238	487
Number of modules with this design type	77	352	83	512	692
Number of faults normalized by the number of modules	0.56	0.52	0.13	0.46	0.70

the purpose was to detect anomalous distributions.

The key result from the audit was that CDIS has an unusually low proportion of components with high complexity, compared with the population at large. In fact, the CDIS code is one of the simplest large packages yet encountered in terms of component complexity. From a developer's point of view, CDIS code is "higher quality"—easier to understand and easier to test—than most systems.

The audit software also produced a graph for each piece of code that displayed in-line fault behavior, plus the results of the complexity analysis. Surprisingly, the structural profiles were similar. Figure 3 shows the typical profile. For almost all criteria in the category, the code is significantly better than average.

These audit profiles painted a picture of modules with a very simple design and loose coupling. Because the three code pieces exhibited the same characteristics, and because the design techniques were different for each, the simplicity cannot be attributed to a particular design method, formal or informal. Instead, the simplicity seems likely to be a direct legacy of a formal specification.

Thus, once again the evidence was inconclusive. We didn't know whether the simplicity resulted from the use of a formal method for specifying the system or from a more thorough than usual analysis of the specification (an indirect result of the formal method). Determining this would require further investigation.

We knew from the static code audit that the overall code seemed to be high quality, but the audit told us little about the effects of the different design techniques. Our next step would be to reveal more about the design by looking at the results of unit testing.

Step 5: Examine the results of unit testing

The quality of delivered code depends on many things, including thorough testing. Each testing activity is effective when it makes problems visible, enabling developers to remove faults before delivery. And the earlier the detection and correction, the easier the next development step. Thus, we decided to examine data about the effectiveness of unit testing, to see whether some parts of CDIS were easier to test than others. Contractually, Praxis was committed to 100 percent statement coverage (with some exceptions granted by the Civil Aviation Authority), meaning that every statement in the code had to be exercised in at least one test script.

The team had tracked statement coverage using Software Research's TCAT tracking tool, which told them the percentage of statements covered by a set of tests. They had unit-tested the VDM and FSM code using Softest, a test harness (a tool that executes and controls testing) from IPL, but they had done no integration testing to locate faults in collections of modules invoked together. They tested the VDM/CCS code (LAN) end to end, rather than relying on conventional unit testing, so some faults that would ordinarily have persisted through unit testing were removed earlier from that code. (This testing technique may explain the early spike in Figure 2.) They also tested the informal code (user interface) in a test harness, permitting a degree of integration and thus also detecting some faults early.

Praxis analyzed its own fault reports (separate from our analysis) and gave us a summary showing the types of faults discovered during development but before system testing. Of all the predelivery faults reported, 340 occurred during code review, 725 in unit testing, and 2,200 during system and acceptance testing (faults designated with 1, 2, or 3). The faults discovered during unit testing, shown in Table 3, occurred in informally designed modules more often than in formally designed ones. This relationship persisted even when we normalized the number of faults, dividing it by the number of modules with that design type (FSM, VDM, and so on). This suggests that formal design may have helped minimize errors or aided fault discovery early in development.

The thoroughness of predelivery testing is borne out by the differences in failures reported before and after release. The delivered code is approximately 10 times less fault-prone after system testing. Only 273 problems were reported between delivery (1992) and June 1994 (the end of our data set). Of these, 147 were attributable to actual code faults.

In our experience, this distribution of faults across review, unit testing, and system testing was unusual. Statistics in the literature suggest that code review is far more effective than unit testing, so more faults should have been found in review than in unit testing.[4] This aberration may have had several causes, including the possibility that the use of formal methods made problems more visible during unit testing. The data we had did not allow us to determine the

Because we had access to tools that reveal code structure, we supplemented our previous analysis with a static, structural look at the delivered code.

Table 4. Changes to delivered code as a result of postdelivery problems.

	FSM	VDM	VDM/CCS	Total formal	Informal
Number of changes	6	44	9	59	126
Number of lines of code with this design type	19,064	61,061	22,201	102,326	78,278
Number of changes normalized by KLOC	0.31	0.72	0.41	0.58	1.61
Number of modules with this design type	67	352	82	501	469
Number of changes normalized by the number of modules	0.09	0.13	0.11	0.12	0.27

Table 5. Postdelivery problems discovered in each problem category.

Category	Number of problems
1	6
2	6
3	46
Specification	1
Design	1
Testing	29
Documentation	11
None assigned	47

cause more definitively. We needed stronger evidence still about the design types. The next logical step was to compare postdelivery failures and their relationship to formal methods.

Step 6: Evaluate postdelivery changes

A look at the postdelivery failures and their relationship to formal methods gave us a different picture of the effects of the design types. The 147 problems (faults 1, 2, or 3) reported during the analysis led to 549 changes to modules or documents; of these, 21 changes were to formally derived documents and 37 to informally derived ones.

Table 4 shows the picture for delivered code. Of the 185 changes to delivered code, six were to FSM-designed modules, nine to VDM/CCS-designed modules, 44 to VDM-designed modules, and 126 to informally designed modules. The remaining changes were to informally designed or specified user documents, test documents, or test modules.

In other words, far fewer changes were required *after delivery* to formally designed parts than to informally designed parts, making the formal code more reliable than the informal code.

We can paint a similar picture of high quality when viewing the same postdelivery problems by severity categories and root causes. As Table 5 shows, only six problems had the designation of 1, and only one specification problem and one design problem have arisen since delivery; the remaining problems were minor.

Reliability is usually measured in terms of mean time between failures, but a reasonable surrogate measure is the number of reported failures per thousand lines of delivered code. If we divide the 147 problems reported by the size of the delivered system, the CDIS code exhibits a remarkably low 0.81 failure per KLOC. Table 6 compares this failure rate with others reported in the literature.[5] The table also shows the general trend of higher reliability in systems that used formal methods. (We view this comparison cautiously, since we do not know how each study defined and counted faults, failures, and lines of code.)

Thus, the postdelivery analysis shows us that the CDIS code compares favorably with others reported, confirming the favorable profile presented by the static code audit. In fact, when we compared a list of the 119 files that contained statically detectable potential problems after release (from the code audit) with a list of the 25 nonheader files that had the most fault reports, we found many of the same file names. In other words, the data strongly suggests that a significant number of the problems revealed during predelivery testing might have been identified and fixed before compilation if an audit had been done earlier.

One of our goals was to determine if one formal method is superior to another. At first, we looked for differences in the number of faults resulting from the use of each method, as reflected by the number of changes required. In looking at testing effectiveness, we began to look at each method as an enabler of other activities. For this reason, we next compared faults discovered predelivery and postdelivery. The differences were striking. In CDIS, system testing was much more important than unit testing in revealing errors; only one fifth of all faults were revealed by unit testing, and most of these were in the user interface code (unit testing and integration were different for different code parts). Praxis feels that either the testing technique or the design method may be responsible for differences in the fault profile.

However, we were struck by the uniformity of the code depicted by the static analysis, regardless of the design or test method. We suspected that the code structure reflected the use of a formal specification, so the specification technique was responsible for the quality, independent of design type or test technique. To us, this difference in fault profile illustrated how formal methods had two effects: one direct and one indirect. The direct effect is that the working system was closer to the requirements, as shown by the spread

Table 6. Postdelivery problem rates reported in the literature.

Source	Language	Failures per KLOC	Formal methods used?
Siemens operating system	Assembly	6-15	No
NAG scientific libraries	Fortran	3.00	No
CDIS air-traffic-control support	C	0.81	Yes
Lloyd's language parser	C	1.40	Yes
IBM Cleanroom development	Various	3.40	Partly
IBM normal development	Various	30.0	No
Satellite planning study	Fortran	6-16	No
Unisys communications software	Ada	2-9	No

of postdelivery failures (Table 5). The indirect effect is that the system was highly testable.

What we found intriguing is that the informally designed code was just as testable (in terms of structure), as suggested by the structural metrics generated by the static code analysis: small, independent modules with low depth of nesting and relatively low cyclomatic number.

We could not tell with certainty from the data available whether this effect was cultural (a result of the general emphasis on quality and repeatability at Praxis), the result of differences in testing techniques, or the result of some other technology or attitude. We could resolve this difference in interpretation only by further investigating the effects of testing, culture, and attitude on the fault-discovery profile.

LESSONS LEARNED

We began our investigation seeking to learn how formal methods affected code quality. Our investigation revealed quantitative evidence of high code quality, but raised many questions about which factors are relevant and how activities and characteristics are related. At the same time, we learned many lessons about how we can use retrospective studies to help us improve our empirical investigations.

Lessons about formal methods

On the one hand, we found no compelling quantitative evidence that formal design techniques *alone* produced code of higher quality than informal design techniques. The predelivery fault profile showed no difference between formally designed and informally designed code. On the other hand, the unit testing data showed fewer errors in formally designed code, and postdelivery failures were significantly less for formally designed code. Thus we can conclude that formal design, combined with other techniques, yielded highly reliable code.

Because the high-quality audit profile was uniform and independent of design type, the formal *specification* is what most likely led to relatively simple and independent components that in turn led to straightforward unit testing.

However, we also learned that formal specification and design are effective under some but not necessarily all circumstances. Their effectiveness may be improved by supplementing them with other approaches, so that in concert they address most of the likely problems of software development. Moreover, formal methods may be more effective in acting as a catalyst for other techniques, especially testing, by virtue of producing testable components or by providing a structure on which to base comprehensive system testing.

Lessons about empirical investigation

Any empirical investigation begins with a look back, to aid in understanding the key variables and how they may be related. The insights gained from looking back allow those conducting future investigations to see what variables should be controlled to tell what is responsible for the effects noted.

The analysis we used might have been more effective had we had additional or different data. This would have allowed us to narrow the cause-and-effect relationship and determine with more certainty the contribution (or lack thereof) of formal methods.

Understand the data. The CDIS team did an admirable job of capturing data about the problems that arose during development and testing; most organizations collect far less data and do far less analysis. Using their data, we have seen that the quality of the system may have been affected by the specification and design methods, the size of the development teams, the communication among groups, or the type of unit testing.

What we didn't know was the experience of the developers (with the language, the tools, the specification/design/testing techniques, the development platform, the application, and each other). Any future efforts should collect data related to these characteristics.

We also found some confusion between fault and failure. Fault reports should separate the problem description into its component issues to clearly and explicitly describe the location, timing, mode, effect, and mechanism. This clarification will allow analysts to evaluate not only which modules were affected and when, but also how the fault could have been discovered and fixed earlier in the process. This would effectively change data analysis from reactive (determining current quality) to proactive (preventing faults or discovering them early). Cost information, appended to the problem reports, would let analysts determine the effect of a fault on the project's budget and schedule. Assessing the costs and benefits of prevention techniques can help managers decide whether additional preventive measures (such as code reviews or cleanroom development) are financially justified.

Capture business and technical effects. As we noted earlier, Praxis did not account for the size of a mod-

Thus we can conclude that formal design, combined with other techniques, yielded highly reliable code.

Our study is far from conclusive, however. Further investigation is needed to determine the role of team size, specification thoroughness, and a host of other variables.

ule when talking about changes or fixes. Indeed, how do you compare a one-line change to a 1,000-line module with a 10-line change to a 10-line module? Developers who want to evaluate the effects of a fault relative to the number of lines of code changed must capture data in fault reports that reflect the size of the change. They should also decide in advance which modules to track when considering the effects of change. For example, are changes to header files significant, or should they not be tracked?

Look for trends. We had information about when a problem was reported and when it was fixed, but we had no information about how much time it took to find and fix the problem. We could have more effectively analyzed the change's effect, say, on the cost and schedule if we had had this information.

Examine structure. Because of the constraints of the QAC tool, we could not audit the code by design type. It would have been useful to compare the structure and potential problems of one design type with another.

Evaluate the effect of intermediate activities. We had little data about the particulars of unit, integration, system, or acceptance testing. We might have been able to come to stronger conclusions if we had been able to look at the data generated by the testing tools, as well as the test scripts and data sets. Test coverage measures may have given us insight into relationships between testing activities and faults discovered.

Separate predelivery from postdelivery. Reliability improvements come only when you eliminate the small proportion of faults that leads to the most common failures.[6] Thus, a fault report should be filed when a fault is found *or* when a failure occurs; the two notions should be explicit (even have separate reporting forms), so that reliability assessment can be separated from maintainability evaluation.

Formal methods have long been discussed in the software engineering community. Common wisdom suggests that using formal methods, both in specification and design, will result in highly reliable code. The need for high reliability in mission-critical applications has encouraged organizations to consider formal methods or adopt them as standard practice, even without compelling evidence of their effectiveness. Our results suggest that using formal specification can lead to code that is relatively simple and easy to test. This, coupled with thorough testing, can produce highly reliable code.

Our study is far from conclusive, however, since many other factors may have contributed to the high quality. Further investigation is needed to determine the role of team size, thoroughness of specification, and a host of other variables. For example, a study that determined how team size affects code quality or that compares a thorough specification with a formal one would provide valuable additional information to the growing body of evidence about formal methods.

We hope that, in reading our study results, practitioners will consider doing three things:

- In project planning, include data definition and capture activities that will let you understand which factors contribute to task effectiveness and product quality.
- Work with researchers to identify trends and relationships so that you can understand how what you do affects what you build.
- Be skeptical. Look for quantitative evidence that standards and recommended practices will really improve your products and processes.

By applying these techniques to future projects that incorporate formal methods, the software engineering community will gain a better understanding of how these methods influence both product and process. ❖

Acknowledgments

This case study was performed as part of the SMARTIE project (Standards and Methods Assessment using Rigorous Techniques in Industrial Environments), supported by the British Department of Trade and Industry and the Science and Engineering Research Council, project 2160. Additional support was provided by the PDCS2 project (Predictably Dependable Computing Systems), funded by the European Commission under its ESPRIT initiative. We gratefully acknowledge the assistance of the other participants in this SMARTIE case study: Stella Page and Norman Fenton at the Centre for Software Reliability, Alun Jones, Anthony Hall, and Martyn Thomas at Praxis plc, and Andrew Ritchie at Programming Research Ltd.

References

1. J. Anthony Hall, "Using Formal Methods to Develop an ATC Information System," *IEEE Software*, Mar. 1996, pp. 66-76.
2. Les Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill, New York, Dec. 1994.
3. B. Nejmeh, "NPATH: A Measure of Execution Path Complexity and Its Applications," *Comm. ACM*, Feb. 1988, pp. 188-200.
4. W. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989.
5. L. Hatton, "Programming Languages and Safety-Related Systems," *Proc. Safety-Critical Systems Symp.*, Springer-Verlag, New York, 1995, pp. 48-64.

6. E. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, No. 1, 1984, pp. 2-14.

Shari Lawrence Pfleeger is president of Systems/Software Inc., where she consults with industry and government on issues involving software engineering and technology evaluation. She is also professor and director of Howard University's Center for Research in Evaluating Software Technology. This work was performed when Pfleeger was a visiting professorial research fellow at City University's Centre for Software Reliability; her work there included evaluating the extent and effect of standards and writing guidelines for software engineers on how to perform experiments and case studies. Pfleeger has written several books in computer science and software engineering, including with Norman Fenton Software Metrics: A Rigorous and Practical Approach *(International Thomson Press, 1997), as well as dozens of research papers in computer science and mathematics. Pfleeger received a PhD in information technology and engineering from George Mason University. She is a member of the IEEE, the IEEE Computer Society, and the ACM. She is associate editor-in-chief of* IEEE Software *and an adviser to* IEEE Spectrum.

Les Hatton is an independent consultant in high-integrity systems, specializing in investigating safety-critical systems. Previously, he was director of research and engineering for Programming Research Ltd. After 15 years as a geophysicist, he now specializes in software safety. He is the author of Safer C: Developing Software for High-Integrity and Safety-Critical Systems *(McGraw-Hill, 1995) and is now completing* Software Failure: Avoiding the Avoidable and Living with the Rest, *to be published this year.*

He received an ALCM in guitar from the London College of Music, bachelor's and master's degrees in mathematics from King's College, Cambridge, and a PhD in computational fluid dynamics from Manchester University. He is on the British Standards Institute C++ committee.

Contact Pfleeger at Systems/Software Inc., 4519 Davenport St. NW, Washington, DC 20016-4415; s.pfleeger@ieee.org.