Do Formal Methods Improve Code Quality?

Shari Lawrence Pfleeger
Centre for Software Reliability
Northampton Square
London EC1V 0HB
England
phone: +44 171 477-8426
fax: +44 171 477-8585
email: shari@csr.city.ac.uk

Les Hatton
Programming Research Ltd.
Glenbrook House, 1/11 Molesey Road
Hersham, Surrey KT12 4RH
England
+44 932 888080
+44 932 888081
les_hatton@prqa.co.uk

Abstract
Formal methods are advocated on many projects, in the hope that their use will improve the quality of the resulting software. However, to date there has been little quantitative evidence of their effectiveness, especially for safety-critical applications. We examined the code and development records for a large air traffic control support system to see if the use of formal methods made a measurable difference. In this paper, we show that formal specification, in concert with thorough unit testing and careful reviews, can lead to high-quality code. But we also show that improved measurement and record-keeping could have made analysis easier and the results more clear-cut.

As practitioners and researchers, we continue to look for methods and tools that help us to improve our software development processes and products. Articles in *IEEE Software* and elsewhere focus on candidate technologies that promise increased productivity, better quality, lower cost or enhanced customer satisfaction. But we are reminded that these methods and tools should be tested empirically and rigorously to determine if they make a quantifiable difference to software we produce.[1,2] Often, such evaluation is not considered until after the technology has been used, making careful, quantitative analysis difficult if not impossible. However, when evaluation planning is incorporated in overall project planning, the result can be a rich record of the effectiveness of the tool or technique.

In this article, we present the results of such an evaluation. We show how use of formal methods was monitored as the project progressed, and analyzed once the project was complete. The lessons reported apply not only to those who are interested in the effects of formal methods but also to those who want to plan similar studies of other techniques and tools. Thus, this exploration of the effects of formal methods resembles a detective's investigation; we report on the motive for the study, the means of finding and analyzing the evidence, and the opportunities for learning about the effectiveness of formal methods.

The motive: A formal look at formal methods

Formal methods mean many things to many people, and there are many types of formal methods proffered for use in software development. Each formal technique involves the use of mathematically precise specification and design notations. In its purest form, formal development is based on refinement and proof of correctness at each stage in the life cycle.

It is widely accepted that a formal method is one that has a formal notation, mathematical semantics, and formal deductive system. Given those requirements, no existing method is truly a formal method. But there are many that are close. Some have mathematical semantics (sometimes partial) but almost no deductive system, such as Z and Statecharts. These Marie-Claude Gaudel calls conceptual techniques. Others have logic but almost no semantics, such as VDM and Unity; these she labels deductive techniques.[3] Still others are defined by an evaluation mechanism (operational semantics or evaluation rules) and are executable specifications. However, executability is not the main interest of formal methods, and in some cases gets in the way of their use. Gaudel also distinguishes between state-oriented (e.g. Z and VDM) and behavior-oriented (e.g. Lotos, Unity, RSL) techniques. So it is clear that all formal methods are not created equal, and it is misleading to lump all such methods together to decide if formal methods make a positive difference to a software project.

For some organizations, the changes in software development practices needed to support such techniques can be revolutionary. That is, there is not always a simple migration path from current practice to inclusion of formal methods, because the effective use of formal methods can require a radical change right at the beginning of the traditional software life-cycle: how we capture and record customer requirements. Thus, the stakes in this area can be particularly high. For this reason, compelling evidence of the effectiveness of formal methods is highly desirable.

Unfortunately, past evaluations of the use of formal methods were 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 only documented case where formal methods were used throughout development was that of the Paris metro. Susan Gerhart, Dan Craigen and Ted Ralston performed an extensive survey of the use of formal methods in industrial environments[4], 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."

As Shari Lawrence Pfleeger, Norman Fenton and Stella Page note, there is even some evidence that formal methods in certain contexts can be counter-productive.[10] For example, Peter Naur[5] 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.[6] And Iain Houston and Steven King report on a joint project between IBM Hursley and the Programming Research Group at Oxford University.[7] A serious attempt was made to quantify the benefits of using Z on the CICS re-specification 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 where the software is safety-critical. For example, John McDermid[8] 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, makes mandatory the use of formal methods. Jonathan Bowen and Michael Hinchey report that other countries are following suit, with some countries requiring formal methods and others strongly recommending them.[9]

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[2], 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 make proper use of formal methods

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

The means: An air traffic control support system

As Anthony Hall describes in his paper,[13] Praxis built an air traffic control information system for the UK Civil Aviation Authority in the early 1990s using a variety of formal methods. The Central Control Function Display Information System (CDIS) provides controllers at the London Air Traffic Control Centre with key information, allowing them to manipulate the approach sequence of aircraft for the one of the busiest airspaces in the world. Hall describes the system function and architecture, making it clear that formal methods were an appealing technique for ensuring the quality of the CDIS software.

Praxis had used formal methods before, but not to the extent used in CDIS. Several different formal methods were involved in the development. The functional requirements were developed using three techniques:

- an entity-relationship analysis to describe the real-world objects, their properties and interrelationships (such as arrivals, departures, workstations, plasma display screens)
- a real-time extension of Yourdon's structured analysis to define the processing requirements
- a formal specification language, to define the data and operations

For reasons of clarity described in detail by Hall, Praxis decided to do a complete, top-level formal specification of critical system elements using VDM. The development team found that this use of formal notation to capture essential CDIS operations improved their understanding of the requirements.

This success led the team to produce the system specification in three parts:

- a formal core specification, written in a VDM-like language
- a set of user interface definitions, using Backus Normal Form syntax descriptions and finite state machine protocol descriptions
- a concurrency specification, using Tony Hoare's communicating sequential processes technique, supplemented with data flow diagrams

Finally, the CDIS design was organized as five components:

- the *design overview*, describing the overall system architecture and design rationale
- the *functional design*, describing the main application modules and derived primarily from the core specification
- the *process design*, describing the processes, tasks, interprocess communication and data sharing, and derived from the concurrency specification and non-functional requirements
- the *user interface design*, derived from the user interface definitions
- the *infrastructure*, including lower-layer application "glue" as well as local area network (LAN) design

The LAN proved to be a very difficult area of design, and its components were eventually expressed using Robin Milner's Calculus of Communicating Systems (CCS).

As Hall points out, "While the three kinds of specification were three different *views* of the same thing, the four different designs (excluding the overview) were designs for different *parts* of the software." Thus, formal methods were used to produce all of the specification and were involved in three places in the design: VDM for the application modules, finite state machines for the processes, and VDM with CCS for the LAN. For this reason, we can categorize any code in CDIS as being influenced by one of four design types: VDM, FSM, VDM/CCS or informally-designed.

Of the almost 200,000 lines of code delivered to the Civil Aviation Authority, the VDM/CCS-derived code (that is, the local area network software) was designed and implemented by a team of two developers. Most of the FSM work (including the links to external systems) was designed by one person. The graphical user interface code, which comprised most of the informally-developed programs, was developed by a team of four people. Finally, the VDM-only designs have as many as ten different authors, though several took responsibility for small areas of the overall system; the number of people who wrote the code from these designs was greater.

The opportunity: A SMARTIE case study

At the same time that CDIS was being developed, the British Department of Trade and Industry and the Science and Engineering Research Council funded a project to investigate the effectiveness of software engineering standards. Called SMARTIE (Standards and Methods Assessment using Rigorous Techniques in Industrial Environments) and led by the Centre for Software Reliability at City University, the collaborative academic-industrial partnership defined a framework for assessing standards and performed several case studies to investigate the effectiveness of particular standards in actual development environments. Shari Lawrence Pfleeger, Norman Fenton and Stella Page reported on the results of the initial SMARTIE work in *IEEE Computer*.[10]

This initial SMARTIE work involved documentation and reporting standards focusing on reliability and maintainability. But the SMARTIE researchers were eager to examine the effect of formal methods on software quality, since several emerging national and international standards are proposed to make their use mandatory. Praxis offered the SMARTIE researchers the opportunity to examine CDIS development data to determine if the use of formal methods had a positive effect on the resulting code. The CDIS development team had kept careful records of all faults reported during in-house system testing, as well as after fielding the system. Comments from the Civil Aviation Authority and users of the system were very positive, and the next step was to determine whether the perceptions of the users were supported quantitatively.

Figure 1 shows a sample of the fault reports used by Praxis on the CDIS development. The format of the fault report changed twice during the early weeks of implementation, as the team grew to understand better what kind of fault tracking information it needed. As a problem was identified, a fault report number was assigned, as well as a category to describe the likely source of the problem: specification, design or code. A brief description of the problem was written on the fault report form, plus suggestions about which code modules, specification documents or design components might be the source. Following contractual obligations, a severity designation was assigned at the same time, with severity categories defined by the Civil Aviation Authority according to the following criteria:

Category 1: Operational system critical

- Corruption or loss of data in any stage of processing or presentation, including database
- Inability of any processor, peripheral device, network or software to meet response times or capacity constraints

- Unintentional failure, halt or interruption in the operational service of the system or the network for whatever reason
- Failure of any processor or hardware item or any common failure mode point within the quoted mean time between failures that is not returned to service within its mean time to repair

Category 2: System inadequate

- Non-compliance with or omission from the air traffic control operational functions as defined in the functional or system specifications
- Omission in the hardware, software or documentation (including testing records and configuration management records) as detected in a physical configuration audit
- Any Category 1 item that occurs during acceptance testing of the development and training system configurations

Category 3: System unsatisfactory

- Non-compliance with or omission from the non-air traffic control support and maintenance functions as defined by the functional or system specifications applicable to the operational, development and training systems
- Non-compliance with standards for deliverable items including documentation
- Layout or format errors in data presentation that do not affect the operational integrity of the system
- Inconsistency or omission in documentation

When the problem was fixed, the responsible party would list on the form all documents and modules actually changed; sometimes the severity designation was changed, too, to indicate the true nature of the problem. Indeed, the designation could be changed to 0, meaning that the perceived problem either was not a problem at all or had been reported previously.

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 FOR EVALUATION 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**

Figure 1. Sample CDIS fault report

There were 910 fault reports designated "0" by the developers. The SMARTIE researchers restricted themselves only to those faults rated non-zero and thereby to those documents and code modules that were actually the source of problems. Moreover, the data was separated so that faults arising from code could be analyzed separately from faults arising from specification or design problems. However, since the Civil Aviation Authority's severity categories and related contractual obligations considered faults that affected documentation and configuration management, we counted as faults not only the changes made to delivered code but also changes affecting test code, configuration management files, and documents.

The detective work

Praxis had taken great care to capture fault and fix data, and their efforts in recording were restricted only by the time available; as with most such projects, there were tight time schedules to consider. Thus, Praxis had done very little analysis of the faults before delivery. On the other hand, they continued to note and track problems after delivery, finding approximately one problem per thousand lines of delivered code.

The SMARTIE team had three basic questions to answer:

1. Did formally-designed code have higher quality than informally-designed code?
2. Was one formal method superior to another?

3. How could data collection and analysis be improved to make quality questions easier to answer?

To begin our investigation, we captured information about each of the over-three thousand fault reports that were generated from the end of 1990 to the middle of 1992, when the software was delivered to the Civil Aviation Authority. Next, we classified each module and document by the type of design that generated it: VDM, VDM/CCS, FSM or informal methods. Then, we generated some summary numbers to get a general idea of how design type affected the number of fault reports that were issued. The number of lines of code reported is a count of non-blank, non-comment C lines as produced by a static code analysis tool.

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

Design Type	*Total Lines of Delivered Code*	*Number of Fault Report-generated Code Changes in Delivered Code*	*Code Changes per KLOC*	*Number of Modules Having This Design Type*	*Total Number of Delivered Modules Changed*	*Percent Delivered Modules Changed*
FSM	19064	260	13.6	67	52	78%
VDM	61061	1539	25.2	352	284	81%
VDM/CCS	22201	202	9.1	82	57	70%
Formal	102326	2001	19.6	501	393	78%
Informal	78278	1644	21.0	469	335	71%

Table 1a presents general data on the size and quality of the code developed from each design type. The first column shows the number of delivered lines of code derived from each of the four types. The second column quantifies the number of code changes resulting from fault reports that affected code of each type. That is, if a single fault report resulted in a change to each of several modules, then each module change was considered a fault and counted separately. Consequently, the total number of faults (that is, code changes) exceeds the total number of fault reports. Then, the third column divides the second column by the number of thousands of lines of code in that design type to yield a normalized measure of the relative number of faults in each design category. Although VDM/CCS-designed code has fewer changes than the others, the changes to informally-designed delivered modules are not substantially different from those of the aggregated formally-designed ones.

The last three columns look at the same data in terms of number of modules. Here, VDM/CCS still produces the best results; fewer VDM/CCS modules were changed overall, but code developed using VDM alone required the most module changes.

Table 1b presents a similar analysis for the extra code that was used in development and testing but not delivered to the customer, and Table 1c is a summary for the entire system (delivered plus extra code). Since lines of code counts were not available for some of the extra code, the tables report only module counts.

Table 1b. Relative sizes and faults reported for each design type in extra code

Design Type	*Number of Fault Report-generated Code Changes in Extra Code*	*Number of Modules Having This Design Type*	*Total Number of Extra Modules Changed*	*Percent Extra Modules Changed*
FSM	14	10	6	60%
VDM	0	0	0	0%
VDM/CCS	1	1	1	100%
Formal	15	11	7	64%
Informal	425	223	221	99%

Table 1c. Relative sizes and faults reported for each design type in all code

Design Type	*Number of Fault Report-generated Code Changes*	*Number of Modules Having This Design Type*	*Total Number of Modules Changed*	*Percent Modules Changed*
FSM	274	77	58	75%
VDM	1539	352	284	81%
VDM/CCS	203	83	58	70%
Formal	2016	512	400	78%
Informal	2069	692	556	80%

If quality is measured by the number of changes needed to correct modules, then the three tables show no clear indication that formal methods produced higher-quality code than informal methods.

However, it may have been the case that, for modules require an extreme number of changes, the type of design method made a difference. Hence, we narrowed our analysis to those delivered modules requiring more than five and more than ten changes as a result of fault reports. Table 2 presents the data for this case.

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

Design Type	*Total Number of Modules Changed*	*Number of Modules with Over 5 Changes Per Module*	*Percent of Modules Changed*	*Number of Modules with Over 10 Changes Per Module*	*Percent of Modules Changed*
FSM	58	11	16%	8	12%
VDM	284	89	25%	35	19%

VDM/CCS	58	11	13%	3	4%
Formal	400	111	22%	46	9%
Informal	556	108	19%	31	7%

Again, VDM/CCS is the highest quality compared with other methods But in this light, informally-designed modules required fewer changes than those designed using formal methods.

We analyzed the documents in a similar manner, doing a causal analysis to determine which changes occurred because of specification problems, design problems and code problems. Each document could be assigned a type, based on the method used to generate most of its contents. This type designator was supplied by Praxis staff, and some of the documents were of uncertain type. Table 3 summarizes the number of documents in each type, as well as the number of faults (that is, changes to documents as a result of a fault report) in each type. As with the code, one fault report can result in more than one change. A change is considered to be a specification change if a specification document was changed, a design change if a design document was changed but no specification was changed, and a code change if a code-related document was changed but no specification or design documents were changed. If more than one document in a type is changed as the result of a fault report, then the change is counted only once in the causal analysis; hence, the total in the three categories (1103) is less than the total number of document changes (1155).

Table 3. Document fault analysis

Document ID	*Blank*	*?*	*Formal*	*Informal*	*Informal?*
Number of documents	6	2	30	24	2
Number of changes	5	60	894	194	2
Specification change	1	1	228	72	0
Design change	7	56	641	85	2
Code change	0	0	9	1	0

None of these tables provides compelling evidence that formal methods are better than informal, in terms of the number of faults located in each design type. But a trend analysis can reveal more information than a static snapshot. Consequently, we grouped fault reports by quarter so that we could examine the proportion of faults reported by design type in each quarter. As before, we counted as a fault each code change that resulted from a fault report, not simply the number of fault reports. Thus, the total number of faults represented by the graph is far more than the number of fault reports. Note too that these are faults resulting in changes to what was to become delivered code, not all faults. (That is, some faults may have required changes only in specification and design but not in the code itself. And we did not include changes to extra code needed for configuration management or testing.)

Figure 2 shows the results of the trend analysis. This graph shows a very large, late spike for the informally- and VDM-designed modules that does not appear for the other design types. Since the graph represents all testing done before delivery, the spikes may represent the

thoroughness of testing, rather than late surprises in the code. These late changes well into the development cycle are not a desirable property and seem to have been avoided in code that was designed using VDM/CCS and FSM. However, the FSM and VDM/CCS code was a much smaller proportion of the system, and each piece was developed by far fewer people than the other two kinds of code. The spikes may represent the more involved communication needed when larger groups develop code together. That is, the spikes may have nothing to do with design method and may reflect instead the organization of the development team.

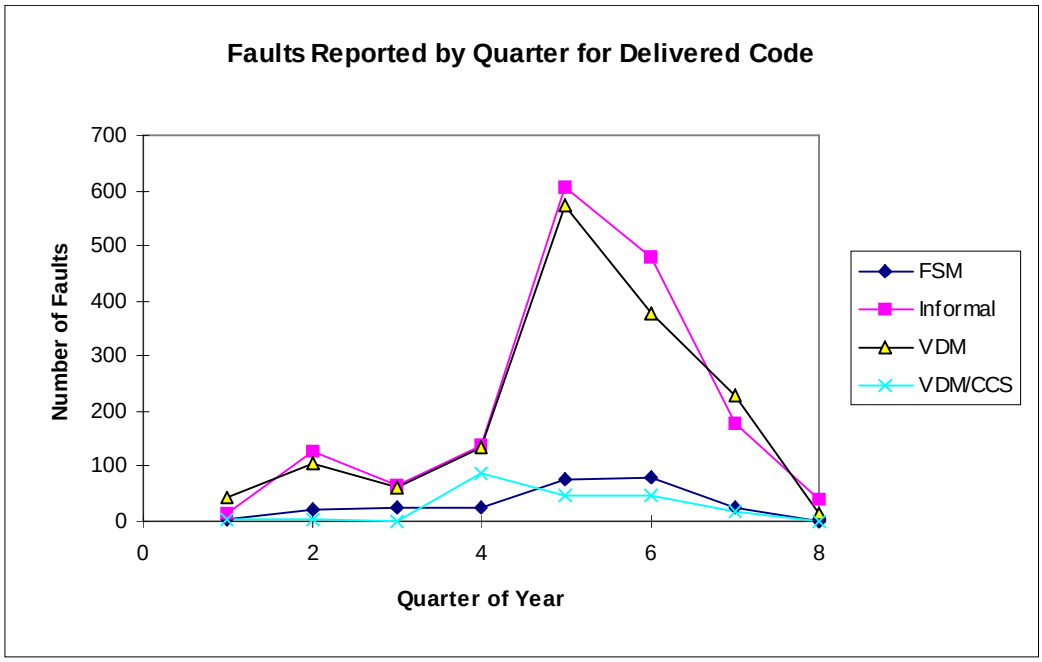

Figure 2: Faults reported over time by design type

To account for the relative difference in code sizes, we normalized all the fault counts by the size of the corresponding code (in thousands of lines of code). As shown in Figure 3, the normalized faults per quarter for VDM and informally-designed code still exceed the rest, but the spiking behavior is common to all four design types.

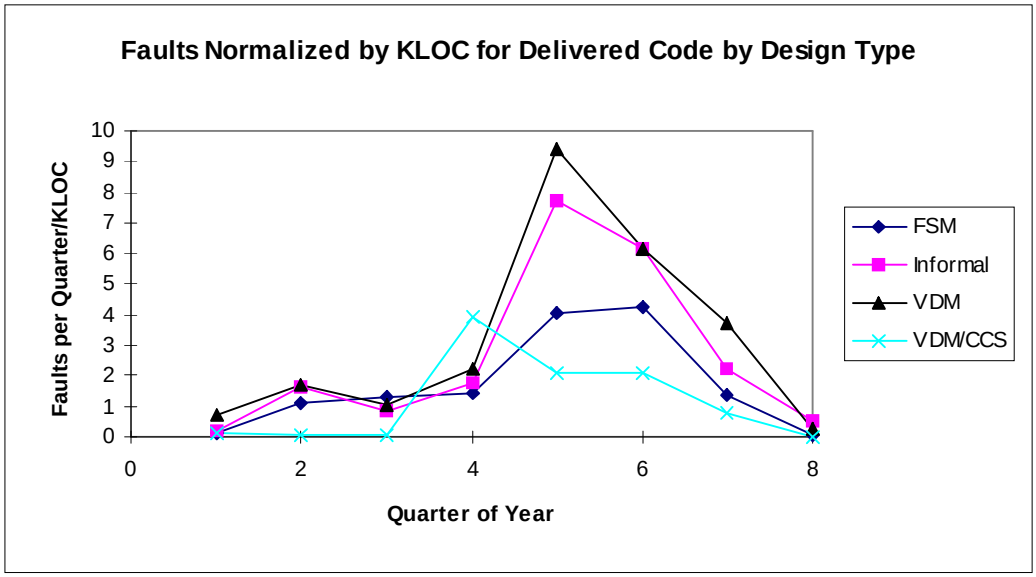

Figure 3. Faults reported by quarter, normalized by size of code

The analysis of fault records was supplemented by a static analysis of the delivered code. Since the CDIS code is written almost entirely in C, the code was audited by Programming Research Ltd. using QAC and QAC++, automated inspection toolsets for the languages C and C++ respectively. In essence, these toolsets detect reliance on unsafe features of these languages as documented in Les Hatton's guidelines for developing safety-critical systems.[14] The C++ parser, QAC++, was used in addition to the C parser, QAC; because QAC has no suitable data structures, QAC++ was able to detect a class of statically-detectable fault in C which currently escapes QAC.

Programming Research has audited millions of lines of code in C packages from around the world in the last few years; this code, representing a wide variety of application domains including many safety-critical systems, formed the population against which the static code analytical results were compared. The audit involved two key steps: analyzing each module for potential faults remaining, and calculating several structure and dependence measures to compare the modules with the larger population in the Programming Research database. The first step helps Praxis to understand what types of coding errors are missed in their testing process; this step is accompanied by a risk evaluation to assess the likelihood that each latent fault will cause a significant error. The second step compares the overall system with other systems written in C, to give a general indication of where Praxis code quality falls in the larger universe of developed systems.

The audit of the CDIS code was not as simple as initially expected. Although the code was relatively big at nearly 200,000 non-blank, non-comment (pre-processed) lines, the processing problems were due in large part to the target platforms concerned (a combination of System 88 and PCs) and to the use of numerous extensions to the base language implementation for C present on both platforms. In a safety-related system, the use of extensions is not recommended insofar as they lie outside the remit of any formal validation.[14] However, in this case their use may be benign; this issue was not pursued in depth, owing to lack of time.

The code was audited under the following categories:

Syntax and constraint violations. These violations are required to be diagnosed by a conforming C compiler. However, none of the compilers used in CDIS appeared to be validated (or perhaps the strict conformance was disabled), and occurrences of both types of violations appeared.

In-line and interface faults. Numerous categories of well-known faults can and do occur regularly in released C code, as noted by Koenig.[15] In this context, an in-line fault is one not associated with an interface, and so is a failure waiting to happen. It is a statically detectable inconsistency in the way the language elements have been combined. Interface faults can be effectively eradicated in C code by using the function prototype construction of Standard C (i.e. as defined by ISO 9899:1990); in general, the CDIS code achieved an excellent 99% or better utilization of this important feature. However, it is not mandatory in C, and some static faults slipped through into the delivered code.

Reliance on imprecisely defined features of C. Standard C lists 197 features in its Appendix F whose behavior is either unspecified (legal but not defined), undefined (illegal but not defined), implementation-defined (well-defined but dependent on a particular compiler) or locale-specific. In addition, there is a wide category of other features which, although apparently well-defined, frequently lead to problems in practice. These are discussed in detail in Hatton's book.[14]

Potential reliance on uninitialized variables. By its nature, this type of fault is a particularly dangerous problem in C.

Reliance on implicit narrowing conversions and implicit conversions between signed and unsigned behavior. Although entirely legal in C, these situations are notorious for producing unexpected behavior. Not all are problematic, but all are worthy of manual inspection.

Clutter. These problems involve unused variables and unreachable code.

Component and system complexity. Using a number of popular software measurements such as cyclomatic complexity,[17] static path count,[15,18] maximum depth of nesting and others, Programming Research has assembled population distributions for large amounts of code from many different application areas. Comparison against these populations allows anomalous distributions to be detected easily.

C++ compatibility. Although not directly relevant to this study, the code is analyzed for compatibility with a C++ compiler. A number of features in C and C++ currently have the same syntax but different semantics. Hence, if such features are present, it is important that C code not inadvertently be compiled by a C++ compiler, especially in a safety-related application.

Auditing the code took place over a four-week period, using a Sparc workstation and Programming Research tools and techniques. The key results are based primarily on population comparisons. First, we noted that the component complexity distributions in CDIS are highly anomalous. Such comparisons are normally done on a percentile basis. That is, the comparison population and the code being analyzed are split up into 10 percentile bins, and the relative populations of each bin are compared. CDIS contains an unusually low proportion of components with high complexity compared to the population at large. In fact, the CDIS code is one of the simplest large packages yet encountered in terms of component complexity.

The code was delivered to Programming Research in three packages, labeled CS, PS and SS. The CS package includes the LAN software and reflects the use of VDM/CCS in the design. The PS package includes the PS/2 interface code, much of which is graphical user interface developed using informal techniques. The SS package holds the remaining software. The population comparisons were done for the individual packages, but the similarity of their profiles is striking. Figure 4 depicts the depth of nesting profile for each of the three packages. The line denoting the average case represents the average over all code in the Programming Research database. A rectangle on the far right of each graph represents modules whose depth of nesting is lower (and therefore better) than 90% of the modules in the larger population. Thus, the figure shows clearly that, in terms of depth of nesting, the Praxis code is far better than average.

Figure 4. Depth of nesting
Include graph for all three packages.

Similarly, Figure 5 depicts the percentile distribution for decision counts (also called cyclomatic complexity), Figure 6 shows the distribution for static path count, and Figure 7 illustrates the use of external coupling. In each case, the Praxis code is significantly better than average.

Figure 5. Decision count

Figure 6. Static path count

Figure 7. Use of external coupling

The other results of the audit are contained in the appendices. It is important to note that many of the latent faults identified by the audit have the potential for severe consequences. Instances of uninitialized variables, inappropriate use of features, and other noted faults should be reviewed for potential threat and fixed as soon as possible.

The results

Considered together, the graphs produced by the audit present a picture of modules that have a very simple design and are very loosely coupled with one another. Since the three packages exhibit the same characteristics, and since the design techniques were different for each package, the simplicity cannot be attributed to a particular design method, formal or informal. Instead, the simplicity seems likely to be a direct legacy of the formal specification. Moreover, the simple modules with few inter-module dependencies suggest that unit testing of components would be highly effective.

Hence, we investigated unit testing techniques and results. Contractually, Praxis had been committed to 100% statement coverage, with some exceptions granted by the Civil Aviation Authority. The statement coverage was tracked using TCAT, a tool available from Software Research in San Francisco. All the user interface code unit testing was done using a specially-built test harness, and all the LAN code was tested using a specially-built test harness. The remaining code was tested using Softest from IPL.

Of all the faults reported, 340 occurred during code review, 710 in unit testing and 2200 during system and acceptance testing (that is, the non-zero fault reports we analyzed). The thoroughness of unit testing is dramatically borne out by the differences in failures reported before and after release; only 250 problems have been reported since delivery (of which 140 were actual code faults), so the delivered code is approximately ten times less fault-prone after unit testing.

It is interesting to compare the inside view (that is, the static code analysis) with the outside view (that is, the faults reported before delivery). The static fault rate measurement is very highly correlated with recorded problems. We compared a list of the 119 files that contained

statically-detectable faults after release (resulting from the Programming Research audit and contained in the appendices) with a list of the 25 non-header files which had most fault reports raised against them (shown in Table 4). Twenty-two of the 25 modules in Table 4 were highlighted in the audit. Eight of the worst 11 and 13 out of the worst 24 identified in the audit are in Table 4. Six of the worst 8 in Table 4 are in the worst 24 list from the audit, 5 of those are in the worst 16 of the audit, and 4 of them in the worst 11 of the audit. In other words, many of the problems that were revealed during pre-delivery testing could have been eliminated before compilation by having done an early audit and fixed the problems identified.

Table 4. Modules identified by at least 20 fault reports

Number of fault reports	*Name of Module*
75	ui_disp.c
55	ui_edd.c
53	asa_p.c
49	ens_s.c
48	nas_s.c
37	dispspl.c
34	asaout.c
31	dvs_p.c
30	err_c.c
29	engutil.c
29	eddatc.c
28	err_c.h
28	pgs_p.c
27	pgo_p.c
26	ctv_s.c
25	edd_p.c
25	sup_s.c
25	aro_s.c
24	ui_text.h
23	lgs_s.c
23	prt_p.c
23	dvs_s.c
22	dvs_c.c
22	grafgedt.c
22	ens_s.h
22	ars_c.c
21	els_c.c
21	cvo_s.c
21	cvd_s.c
20	ui_utils.c

This comparison also suggests that grafutil.c, editutil.c and graford.c, which are among the worst six highlighted in the audit (but do not appear in Table 4), may also cause problems in the future. The modules graftext.c, dpo_s.c, edi_p.c and rlo_s.c should also be reviewed for potential errors.

We can draw several important conclusions from the quantitative analyses performed on the Praxis code.

1. There is no quantitative evidence that formal design techniques produced code of higher quality than informally-designed code. Furthermore, attempts to separate the behavior of code produced using one type of formal method from another showed no significant difference.

2. Formal specification has led to components that are relatively simple and independent, making them relatively easy to unit-test.

3. Static inspection of the code could have led to more effective unit testing.

4. Thorough unit testing has resulted in very low failure rates.

5. To achieve the highest levels of reliability, developers can combine formal specification with static inspection and good unit testing. But all three are required. The Praxis code shows that, without static inspection, the code contains several latent faults that can have severe consequences in actual use.

In other words, formal methods may be most effective in acting as a catalyst for other methods, especially unit testing, by virtue of producing testable components.

The recommendations

As with any development process, there is always room for improvement. The recommendations made here should be considered by Praxis in light of resource constraints and contractual obligations.

Capturing size of change. The analysis performed does not distinguish size of "module" when talking about changes or fixes. Indeed, how do we compare a one-line change to a 1000-line module with a 10-line change to a 10-line module? If Praxis wants to evaluate the impact of a fault in terms of the number of lines of code changed, it must capture data it its fault reports that reflect the "size" of the change. Moreover, thought must be given to which modules should be tracked when considering change impact. Our analysis has included all files, including header files and configuration management files. We included everything because the configuration management system may require a great deal of effort, even for small files. For example, even though module x.c is related to module x.h, it may take almost as long to handle the change to x.h as it does to x.c.

Automating fault tracking and data analysis. No regular development organization has large amounts of time to spend analyzing data of this kind. The majority of the work we did involved data entry and organization, since the fault reports were recorded only on paper. The presence of a fault log in spreadsheet form was quite helpful, but a completely automated fault reporting system would have made analysis easier and more timely. This automation need not be expensive or excessive; all of our analysis was done using a spreadsheet and simple relational database on a PC.

Implementing coding standards, reviews and audits. The number of latent faults in the delivered code, including several with potentially severe consequences, suggests that Praxis needs coding standards, reviews and audits. As shown in comparing the audit results with Table 4, the combination of these practices would have eliminated the bulk of faults before delivery.

Faults, failures and problem reports. Praxis has done an admirable job of capturing data about the problems that arose during development and testing. However, several recommendations can be made about their fault report forms that will enhance understanding on future projects. The major change to their reporting standards involves the difference between a fault and a failure.

As Peter Mellor explains in his excellent paper on data collection,[11] a *fault* is the result of human error that manifests itself as a mistake in code or related products. A *failure* is the incorrect working of the software, as viewed by the user, and may be the result of one or more faults. Thus, maintenance deals with the finding and fixing of faults, while reliability involves the mean time between failures. In some sense, then, maintainability reflects an insider's view of the system, while reliability reflects the outsider's (user's or customer's) view.

These very different perspectives are reflected in the way data is recorded. In many organizations, "incident" or "problem" reports are used to capture the data. Here, "incident" can mean a fault, a failure or even a change. A typical incident report may look something like this:

> **Incident Report**
> **Location:** Where is it?
> **Timing:** When did it occur?
> **Mode:** What was observed?
> **Effect:** Which consequences resulted?
> **Mechanism:** How did it arise?
> **Cause:** Why did it occur?
> **Severity:** How much was the user affected?
> **Cost:** How much did it cost?

On the surface, this report template should suffice for all types of incidents. However, the questions are answered very differently, depending on whether you are interested in faults, failures or changes. A failure report focuses on the external problems: the installation, the chain of events leading up to the failure, the effect on the user or other systems, and the cost to the user as well as the developer.

> **Failure Report**
> **Location:** such as installation where failure observed
> **Timing:** CPU time, clock time or some temporal measure
> **Mode:** type of error message or indication of failure
> **Effect:** description of failure, such as "operating system crash," "services degraded," "loss of data," "wrong output," "no output"

Mechanism: chain of events, including keyboard commands and state data, leading to failure
Cause: reference to possible fault(s) leading to failure
Severity: reference to a well-defined scale, such as "critical," "major," "minor"
Cost: Cost to fix plus cost of lost potential business

However, a fault report has very different answers to the same questions. It focuses on the internals of the system, looking at the particular module where the fault occurred and the cost to locate and fix it.

Fault Report
Location: within-system identifier, such as module or document name
Timing: phase of development during which fault was discovered
Mode: type of error message reported, or activity which revealed fault (such as review)
Effect: category (missing/wrong/extra) or type of fault (logic, data-flow, etc.)
Mechanism: how source was created, detected, corrected
Cause: type of human error that led to fault
Severity: refer to severity of resulting or potential failure
Cost: time or effort to locate and correct; can include analysis of cost had fault bee identified during an earlier activity

Similarly, a report for changes has yet another focus:

Change Report
Location: identifier of document or module changed
Timing: when change was made
Mode: type of change
Effect: success of change, as evidenced by regression or other testing
Mechanism: how and by whom change was performed
Cause: corrective, adaptive or perfective
Severity: impact on rest of system, sometimes as indicated by an ordinal scale
Cost: time and effort for change implementation and test

The Praxis fault reports can be improved in several ways. First, the problem description should be separated into its component issues, so that the location, timing, mode, effect and mechanism are clearly and explicitly described. This clarification will enable those analyzing the fault reports 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. Thus, the data analysis could change from reactive (determining current quality) to proactive (preventing faults or discovering them early before their effects proliferate to other parts of the system).

Second, the cost information should be appended, enabling analysts to determine the effect of a fault on the project's budget and schedule. Again, prevention techniques can be assessed, including cost-benefit analyses to help managers decide whether additional preventive measures (such as code reviews or cleanroom development) are financially justified.

Praxis should distinguish faults from failures. Many practitioners consider faults and failures to be one and the same thing, thinking that it is not important to distinguish them and that both can be predictors of reliability and maintainability. But research indicates that such blurring of the distinction can be misleading. Ed Adams[12] examined thousands of faults in IBM operating systems, identifying the time between when the fault was introduced and when it would be observed by a user. His results are disquieting to those who believe that one fault is much like another. The Adams data indicates that most faults are benign, in that their effects will never be seen by a user. Thus, the removal of those faults has no effect on the reliability of the system; reliability improvements come only when we eliminate the very small proportion of faults that lead to the most common failures.

At Praxis, a fault report is filed when a fault is found *or* when a failure occurs; the two notions should be made more explicit (the reporting forms could even be separated), so that reliability assessment can be made separate from maintainability evaluation. The CDIS system had over 7 faults per thousand lines of code before delivery, but the one problem reported per thousand lines of code after delivery may actually be a combination of faults and failures. In fact, the reliability of CDIS is likely to be even less than one failure per thousand lines of code, an extremely good level of delivered quality.

Conclusions

Formal methods have long been discussed in the software engineering community. Common wisdom suggests that use of formal methods, both in specification and design, will result in highly reliable code. The need for high reliability in safety-critical applications has encouraged organizations to consider or adopt formal methods as standard practice, even in the absence of compelling evidence of their effectiveness. In this study, we have analyzed the fault records and delivered code of a system developed with a variety of formal specification and design techniques. Our results show that the use of formal specification has led to code that is relatively simple and easy to unit test. This, coupled with thorough unit testing, has led to highly reliable code.

On the other hand, static code analysis revealed a significant number of potentially hazardous latent faults in the delivered code. Static code inspection could have identified these problems much earlier, allowing the developers to fix them before delivery. Thus, the combination of formal specification, static inspection and thorough unit testing is necessary for achieving high levels of reliability.

In other words, a formal method (in particular, formal specification) is certainly part of the solution to improving code quality, but it is clearly not the whole answer. Based on our findings, developers may choose to repeat this case study and verify that formal methods (coupled with static inspection and thorough unit testing) can lead to highly reliable code. They may begin by using formal methods on small, critical parts of systems, and then look for other, supplemental techniques for the rest of their large systems.

References

1. C. Potts, "Software Engineering Research Revisited," *IEEE Software*, September 1993, pp. 19-28.

2. Norman Fenton, Shari Lawrence Pfleeger and Robert L. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, July 1994, pp. 86-95.

3. Marie-Claude Gaudel, State of the Art Report on Formal Methods, *Proceedings of the International Conference on Software Engineering*, Sorrento, Italy, May 1994.

4. Susan Gerhart, Dan Craigen and Ted Ralston, "Observation on Industrial Practice Using Formal Methods," *Proceedings of the 15th International Conference on Software Engineering*, IEEE Computer Society Press, May 1993, pp. 24-33.

5. Peter Naur, "Understanding Turing's Universal Machine — Personal Style in Program Description," *Computer Journal*, 36(4), 1993, pp. 351-371.

6. Susan Gerhart, Dan Craigen and Ted Ralston, "Experience with Formal Methods in Critical Systems," *IEEE Software*, January 1994, pp. 21-28.

7. I. Houston and S. King, "CICS Project Report: Experiences and Results From the Use of Z," *Proceedings of VDM'91: Formal Development Methods*, Volume 551, Lecture Notes in Computer Science, Springer-Verlag, 1991.

8. John A. McDermid, "Safety-critical Software: A Vignette," *IEE Software Engineering Journal*, 8(1), 1993, pp. 2-3.

9. Jonathan Bowen and Michael G. Hinchey, "Formal Methods and Safety-Critical Standards," *IEEE Computer*, 27(8) August 1994, pp. 68-71.

10. Shari Lawrence Pfleeger, Norman Fenton and Stella Page, "Evaluating Software Engineering Standards," *IEEE Computer*, September 1994, pp. 71-79.

11. Peter Mellor, "Failures, Faults and Changes in Dependability Measurement," *Journal of Information and Software Technology*, 34(10), October 1992, pp. 640-654.

12. E. Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development*, 28(1), 1984, pp. 2-14.

13. J. Anthony Hall, "Lessons Learned in Applying Formal Methods," submitted for publication.

14. Les Hatton, *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill, December 1994.

15. Les Hatton and T. R. Hopkins, "Experiences With Flint, a Software Metrication Tool for Fortran 77," *Symposium on Software Tools*, Napier Polytechnic, Edinburgh, 1989.

16. A. Koenig, *C Traps and Pitfalls*, Addison-Wesley, Reading, Massachusetts, 1988.

17. T. A. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering*, SE-2(4), 1976, pp. 308-320.

18. Brian Nejmeh, “NPATH: A Measure of Execution Path Complexity and Its Applications,” *Communications of the ACM*, 31(2), 1988, pp. 188-200.

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 number 2160. 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.

Biographies

Shari Lawrence Pfleeger is president of Systems/Software, Inc.

Les Hatton is chief scientist at Programming Research Ltd.