

Applying Formal Specification in Industry

PETER GORM LARSEN, *The Institute for Applied Computer Science*

JOHN FITZGERALD, *Centre for Software Reliability*

TOM BROOKES, *British Aerospace Systems and Equipment*

The authors introduced formal methods into the specification and modeling activities of a security-critical system's development. They gauged the methods' effectiveness by comparing the results of the group that used them with those of the group that did not.

Industrial software developers confront a bewildering array of software-engineering techniques, each with its own promised benefits. When choosing between them, developers often rely more on what Norman Fenton calls "the unsubstantiated advertising claims and biases of producers, both academic and industrial,"¹ than on the actual costs and benefits of these techniques.

Companies are sometimes reluctant to be the first to enter new territory: Without a sufficiently large public body of evidence, new techniques may be considered too immature for commercial adoption. Many industrial companies have already invested a great deal of effort in their development process, which has normally undergone some form of certification to show that it conforms to acceptable standards, such as the ISO 9000 series. Introducing a new technology into that process is difficult if it requires fundamental change.

A more acceptable approach is to introduce an incremental change whose overall effects can be identified and quantified. In this article, we report on the lessons learned during a study of one such change in the software-development process at British Aerospace Systems and Equipment Ltd.

VDM-SL: A SHORT EXAMPLE

A model-oriented specification language, VDM-SL mediates the specification of a system and models the system's internal state using a mathematical model in which data types represent the classes of input and output values. System functionality is modeled by functions and operations working on values of these types.

In the case of a trusted gateway, the messages read in by the gateway might be modeled as sequences of characters:

```
Message = seq of char
inv m == len m <= 10000
```

The *invariant* `inv ...` records an additional restriction that the length of a message may not exceed 10,000 characters. A function on messages might return a Boolean value

true if one message is a submessage of another. This is specified as follows:

```
substring: Message * Message -> bool
substring(s1,s2) == exists i,j in set inds s2
& s1 = s2(i,...,j)
```

VDM-SL defines the result of this function without suggesting a particular algorithm for searching the larger string. The constructors for data types and the ability to specify functionality without bias towards particular algorithms contribute to the language's support for abstraction. Many more examples of VDM-SL can be viewed from the VDM Examples Repository at <http://www.ifad.dk/examples/examples.html>.

At BASE, we were interested in developing a security-critical system to levels of assurance that required the use of formal specification for modeling the security policy. Our study's purpose was to assess the effect of introducing a modest amount of formal specification into an existing development process. The study did not attempt to prove the cost-effectiveness or technical value of formal methods generally. We also intended to introduce a relatively minor delta to the development process, so there was no stress on formal proof — we used the formal specification language largely for system and software modeling.

The study itself consisted of the parallel development by two separate engineering teams of a system component known as a trusted gateway. One team employed what we refer to as the *conventional path*: The conventional BASE development methodology using structured analysis with CASE tool support. The other team employed what we refer to as the *formal path*: Essentially the same design process, but supplemented with formal specification wherever they felt it appropriate.

To maximize the two teams' independence, we enforced procedures such as ensuring that they remained physically separated and could not communicate about the project. We designed the study's structure so that the only difference in the technical approaches employed in the two paths was the use of formal specification. This let us focus on how adding formal specification to the development process affected the cost and quality of the designs produced.

System specifications are usually presented in a mixture of natural language, graphical notations, and pseudocode. A formal specification, by contrast, uses a precisely defined mathematical language to describe the system properties. This leaves less room for ambiguity in the requirements, and also opens the way for automatic checking for errors and inconsistencies in requirements. Formal-specification languages usually provide a variety of abstract, high-level constructs that permit the description of system properties without dealing with implementation details.

In this project, we used the Vienna Development Method Specification

**A monitoring team
acted as central
authority and
customer.**

Language, currently in the last stages of standardization under ISO.^{3,3} The box above shows a short example of VDM-SL.

The trusted gateway system we developed is a small but security-critical message-handling device. It acts as a filter between a high-security system and less secure systems, ensuring that messages of high secrecy are not passed to systems deemed too insecure to handle them. The security-critical nature of the system called for a high-level of assurance under the ITSEC⁴ standard, which advocates the use of formal techniques. In a full commercial product development, certification to ITSEC levels would be carried out by an accredited evaluator separate from

BASE. In our study, costs prohibited taking the two designs through external evaluation. We did, however, obtain the certification authority's assurance that our use of formal specification would most likely not hinder the certification of this system.

In this article, we do not dwell on the details of the application or the particular specifications developed; these have been reported elsewhere.⁵⁻⁷ Instead, we focus on the lessons learned during each phase of the development. Our study included the main phases of the software-development process at BASE: system analysis, software design, implementation, and testing. The development process used at BASE follows the "V" life cycle, with system test plans produced along with designs at each stage. Each development team assigned one engineer to each phase, with a new engineer taking over whenever a new phase started.

In addition to the two development teams, a monitoring team acted as central authority and customer, keeping records of the project's queries, problems, and progress. This team recorded observations, including metrics, at different points during the development. We first present the lessons learned from the comparisons of these observations, phase by phase, then discuss lessons common to all the phases and the distribution of human effort expended across each of the two paths.

SYSTEM ANALYSIS

The system-analysis phase begins when the systems engineer receives a *customer requirement*, a natural-lan-

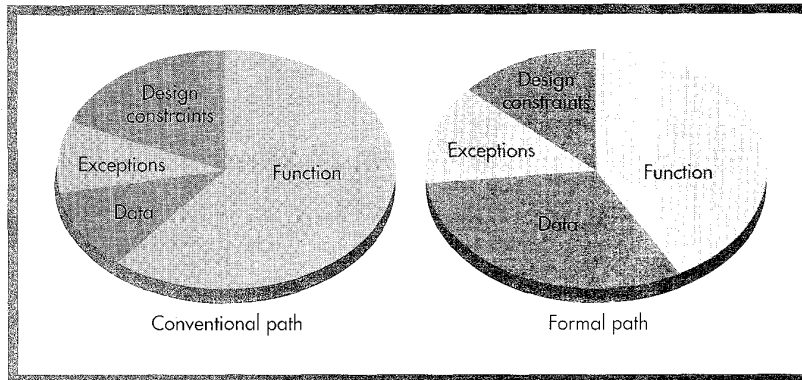


Figure 1. The difference in query patterns between the two paths.

guage document produced by the client that details the product requirements. The systems engineer transforms this into a collection of numbered clauses of roughly equal complexity. He then begins to produce a model of the proposed system, using a CASE tool. In this project, we used Cadre's Teamwork CASE tool. In addition, the engineer on the formal path chose to produce a formal specification of

- ◆ the data types in VDM-SL, recorded in the Teamwork data dictionary; and

- ◆ the principal functions of the system, recorded in the process specifications.

During the analysis and modeling of the system, the engineer normally raises queries against the customer requirement. In the trusted-gateway study, these were submitted to the monitoring authority in writing. The monitor returned written answers, which gave us a log of the questions and answers so we could analyze differences between the two paths.

At the end of the phase, both development teams produced a system specification and system test plan. These documents were reviewed for each path and feedback given to the separate developers. The monitoring team performed a third comparative

review, which kept a log of exposed deficiencies. No feedback from the comparative review went to the developers. BASE considered both engineers employed in this phase to be equally qualified and to have a similar background.

Queries against requirements. What difference did the use of formal specification make to this system-analysis phase? We used the log of queries to track how well the two teams tackled the problems of misstated, incomplete, or ambiguous requirements.

The engineer on the conventional path submitted approximately 40 questions; the engineer on the formal path submitted approximately 60. We gave the actual queries to five other engineers — drawn from the systems, software, and quality departments — and asked them to assign each query, without knowing its source, to one of four categories:

- ◆ **Function:** to clarify the function the system must perform under normal circumstances;

- ◆ **Data:** to understand the data, data type, or data structure better;

- ◆ **Exception:** to determine what the system must do under exceptional circumstances;

- ◆ **Design constraint:** to clarify design constraints under which the system

would be developed.

Figure 1 shows the difference in query pattern. The team using the formal path appears to have focused more on the data and the exceptional conditions than the team using the conventional path. In particular, we consider the emphasis on the exceptional cases valuable in the requirements-capture process of security-critical systems.

System-analysis lessons. The systems engineer on the formal path said the main difference between formal and conventional development was the resulting model's higher level of abstraction. He also noted that early identification of the data needed by the system highlighted the boundary cases. On the negative side, he said abstraction skill is difficult to acquire and requires experience, and the analysis process was more time-consuming because he had to produce the formal specification in addition to the basic Teamwork model. Further review and discussions with the systems engineer produced the following insights:

Effort is concentrated in areas susceptible to design errors. Data definitions and exceptional behavior have traditionally been sources of design error in message-processing systems. A method, formal or informal, that encourages concentration on those areas could lead to eventual improvements in product quality. This was shown clearly during the comparative review at the end of the system-analysis phase. We realized then that the engineer on the formal path had raised some queries to clarify the gateway's behavior under a particular combination of conditions that turned out to constitute an exception. This exception had not been identified as such in the original requirements document. The formal specification was thus designed to handle the exception, but the conventional design, perhaps concentrating



less on the description of data and exceptions, treated the situation as entirely normal. This omission in the requirements analysis came back to haunt the conventional developers some months later.

Of course, this lesson depends on an appropriate choice of specification language. In our case, the concentration on data may in part be attributed to the choice of a model-oriented language that emphasizes data description. A language that stresses communication between processes, on the other hand, might have proved less appropriate for this application domain.

Using formal specification in system analysis seems to push implementation-dependent design constraints to the background, letting the engineer concentrate on defining exactly what the system must do, not how it should do it. BASE generally considers this approach a good thing. However, project management must take this into account and not interpret the time spent on production of an abstract formal model as project slippage. The phenomenon may be familiar from the introduction of structured-design methodologies.

Specification style will be immature at first. The engineer did develop an abstract system specification, although we noted a tendency toward using familiar constructs instead of exploiting the richness and variety of the formal notation. We expect that this problem will diminish as the notation and its use become more familiar to engineers.

Tool support is essential. The engineer reported that the formal specification improved his ability to analyze the system more thoroughly. His ability to test a specification was essential, he felt, for gaining confidence in the specification's correctness.

SOFTWARE DESIGN

Specialist software engineers use the system specification produced in the first phase to produce a more detailed model of those system parts that are to be implemented in software. Designers may employ more concrete data structures and usually describe the functionality in detailed pseudocode.

In the BASE study, this phase resulted in designs and test plans from both development teams. The formal path produced VDM-SL versions of type definitions and procedural specifications of functionality. The monitoring team again reviewed and compared the documentation. The engineers available for this phase had different degrees of experience, which limits the conclusions we can draw from a comparison between it and the system-analysis phase. The engineer on the formal path had more experience overall, but was new to formal specification.

Software-design lessons. In both paths, the engineers raised far fewer queries against the requirements in this phase than in the system-analysis phase. Most of the information sought related to the understanding of the models produced in the system-analysis phase. It appears that the engineers consider the requirements to be fixed and "correct" at this stage.

The reviews noted that the formal path's software design was presented at a higher level of abstraction than would normally be expected. This means that more design decisions would have to be made by the engineer implementing the software. Whether or not this is desirable is a contentious issue, one best determined by the skill of the engineer who is implementing the design. The following lessons are relatively independent of the comparison between the two phases:

Some duplication of work can occur. The engineer on the formal path followed his normal design process: He produced pseudocode for the proposed functions before he translated it into VDM-SL for tool-supported testing. This is not the most efficient method, but was comfortable for this particular engineer, either because

- ♦ the training had not given him the appropriate skills for refining the specification towards implementation, or

- ♦ he was relatively experienced and felt that the new technology did not improve his customary way of working.

Testing functionality may encourage improvements to original requirements. If engineers in the later design phases do not question the correctness of the system specification, errors introduced in the early stages of development may go undetected until the testing phase. This may have contributed to the

**Using formal
specification in
system analysis
lets the engineer
concentrate on
what the system
must do, not how
it should do it.**

problem in the conventional path. There may be greater opportunity to identify exceptions when functionality testing is available through abstract modeling, as in the formal path.

Training must be appropriate for different development roles. We originally felt that the same training would be appropriate for systems engineers and soft-

TABLE 1
CODE SIZE OF FINAL
KERNEL IMPLEMENTATION

Lines	Conventional Path	Formal Path
Code	371	63
Comment	82	63
Ratio	0.22	1

TABLE 2
FINAL-IMPLEMENTATION PERFORMANCE

Performance Measure	Conventional Path	Formal Path	Ratio
Initialization time (seconds)	17	70	4
Processing rate (cps)	18	250	13.8

ware engineers alike. We would revise that view. The systems engineer's task is to establish the logical abstract description of the system's functionality as a whole. This requires skills in abstraction, functional specification, and data specification. The software designer takes the abstract logical specification and refines it toward implementation. This requires additional skills in functional refinement and consideration for the implementation aspects of the design.

IMPLEMENTATION

In the implementation phase, the detailed software design for both paths was first written in pseudocode. The final code was then written in C and compiled using Microsoft's Visual C++ compiler on an MS-DOS platform. One of the central authority's software engineers designed a common user interface for the trusted gateway software, which we gave to both implementation teams. This permitted a blind comparison of the two products, which could not be distinguished by their screen layout. The implementation engineers tested the developed code with respect to the test plans produced in each path. As in the system-analysis phase, the implementing engineers had comparable qualifications and experience.

In an additional test, the engineers ran each of the final implementations against the test suites developed in the other path. Although the conventional

path failed several of the tests developed in the formal path, the root cause of the failures was the same: The condition that caused this fault had been detected in the formal path in the system-analysis phase, but had never been detected in the conventional path. The conventional development team wrote a software patch to correct the problem.

But had the trusted gateway been a commercial product developed with conventional techniques, the error would have gone undetected until the software entered service. Furthermore, had the design been subjected to external evaluation as a secure product, the documentation for all phases would have required revision and the design would have required re-evaluation. This would have involved repeating a significant portion of the original development effort. Although we cannot make a detailed quantitative estimate, we believe that this would have doubled the evaluation costs, with overall development costs increasing by 20 to 30 percent.

Code characteristics. We compared the source code that each path developed for the kernel routine, which implements the security-enforcing function. We found the McCabe complexity of this routine to be 74 on the conventional path and 10 on the formal path, suggesting that the formal path produced better-structured and more-maintainable code. Investigating the conventional path's source code before and after the addition of the software patch showed a significant leap in complexity.

After the code was rewritten to correct the deficiency, the McCabe index increased from roughly 10 to 74. This suggests that conventional development does not in itself produce more complex code, but that complexity can be introduced when the code is revised at the end of a development to correct problems discovered late in the testing process.

The number of lines of code in a routine is not a particularly meaningful metric, although in the case of the kernel routine it is valuable because this routine is responsible for the same functionality in both implementations. The data in Table 1 indicate that the routine produced by the formal path is much more compact.

The ratio of lines of comment to code is a rough guide to the maintainability of the final system. The differences identified here, however, cannot be attributed solely to the use of formal specification: They could also be an indication of the implementor's ability.

We tested the trusted gateway's speed of operation by passing a large block of messages through the system. To minimize external factors, we installed the software on the same machine for each test and disabled all other applications.

Table 2 shows the results. The formal implementation spends roughly four times as long checking the system data as does the conventional implementation. However, when the formal-path system processes messages, it does so almost 14 times faster than the conventional-path system. For a trusted gateway, which is designed to be set up once and then left to operate indefinitely, the relative speed of initialization is not a major factor in assessing system performance. The software developed in the formal path would thus be considered much faster than the conventional implementation. The difference in performance identified here is, again, partly attributable to the software patch.

TABLE 3
COST DISTRIBUTION

Phase	Allocated Time	Conventional path		Formal path	
		% of Budget Allocated	Relative % of Project	% of Budget Allocated	Relative % of Project
System analysis	40%	30%	33%	35%	43%
Software design	40%	40%	47%	33%	41%
Implementation	20%	17%	20%	13%	16%
Total	100%	87%	100%	81%	100%

Implementation lessons. We derived the following lessons from the implementation phase.

Errors detected late in the design process are expensive to correct. In addition to the cost of fixing the errors, the corrections may result in code that is poorly supported by design documentation and difficult to maintain.

An abstract software design allows implementors some design decisions. The abstract nature of the software design produced in the formal path left some design decisions to the implementing software engineer. This posed some problems for the engineer, who was more used to implementing software that had already been designed such that it required only final coding. The engineer experienced particular difficulties because he had not been trained in the use of VDM-SL.

Training in the use of formal specification is essential for an engineer who will implement from a formal specification. The ability to read a specification is not sufficient; the training must also encompass the concepts behind the use of formal specification. The freedom left in the formal specification to be resolved by the implementor should be emphasized.

PROJECT OVERVIEW

We now turn to the conclusions drawn from the development process as a whole, beginning with the costs associated with the formal specification techniques employed.

Table 3 gives an overview of the cost distribution in the two development paths, broken down by both the percentage of the allocated budget spent on each activity for each path

and by what percentage of the project's cost each activity consumed for each path. If we restrict the effort spent during the system-analysis phase to the actual development, removing effort spent for comparative reviews between the two paths and so forth, the formal path required roughly 25 percent more person-hours than the conventional path. Available resources limited neither engineer. Both underspent the budget — by 25 percent in the conventional path and by 12.5 percent in the formal path. We have not included the time spent on training and external consultancy

Available resources limited neither engineer. Both underspent the budget.

because these tasks would not be needed for subsequent projects that used formal methods.

In the software-design phase, the engineer in the formal path had more experience. This resulted, we feel, in the formal path requiring less effort to complete than the conventional path. Taking this into account, the BASE engineers estimated that the effort in the phase would be similar in both paths if equally experienced engineers had been available.

In the implementation phase, the formal and conventional paths took the same amount of effort to complete the first version of the system. The conventional implementation required rework, however, which increased the effort required by roughly 15 percent. Both tasks were completed within the allocated budget. As a percentage of the overall

project, the formal implementation took 13 percent of the effort, the conventional implementation took 17 percent.

Comparing the effort over the entire development, the formal path did not incur an overhead. The overall effort required was slightly less than that of the conventional development, but not significantly so. This result tends to confirm the conventional wisdom that formal specification adds to costs in the early stages of the development process, when engineers analyze system requirements, but that this additional effort is recovered in the later stages. This change in the effort profile is also typical of the introduction of structured-design methods, which promote systems understanding before development.

Using formal specification. For the message-processing systems developed by BASE, we can divide into two parts the use of formal specifications in a development process controlled by a structured methodology: data modeling and functionality definition.

Using formal specification in data modeling is valuable: Data definitions written in a common, implementation-independent manner let this information be passed through the various development phases without transformation, although detail is added until the system is implemented. The rigor required to define the data using a formal notation also forces the designer to question the customer or clarify assumptions about the data very early in the development process.

We do not consider formal specification of functionality advantageous when the required function could be written unambiguously or specified using existing and well-understood



processes. As Jonathan Bowen and Michael Hinchey point out, we must determine exactly when it is advantageous to use formal specification.⁸ We

The choice of specification language is crucial for the application's success.

conclude that formal specification is useful when

- ♦ there is a complex data structure that must be handled correctly;
- ♦ a precise function definition is required when a simple function is needed, but it is vital that the function be implemented correctly; or
- ♦ complex functionality is involved, when there are many choices to be made or many exceptional conditions arise.

SPECIFICATION LANGUAGE

Although this study used VDM-SL, we believe that similar results could be obtained by using other model-oriented formal-specification languages that have tool support for interpreting an executable subset of the notation.

Different specification languages are suitable for different applications.^{9,10} Experience from other applications shows that the choice of specification language is crucial for the success of the given application.¹¹ Specifically, the notation must focus on concepts that are fundamental to the given application domain.

In our project, the engineers preferred to use an ASCII representation of VDM-SL over the more commonly used mathematical notation, although ASCII is more verbose. They probably made this choice because the IFAD (Institutet For Anvendt Datateknik)

VDM-SL Toolbox used ASCII, but the engineers also claimed that it eased their presentation of formal specifications to colleagues unfamiliar with mathematical notation. We learned from this that the mathematical notation used by experts may be a barrier to the introduction of formal specification in an industrial environment.

What did formality contribute? At several points in this article, we have suggested that some of the success of the formal modeling exercise could be attributed to the choice of a model-oriented specification language. We have also observed that the study made no use of formal proof. Suppose we had used a language with the same facilities as VDM-SL, but only an informal semantics. Would the same benefits have been gained? If proof is not exploited, what does the formality of the specification language contribute?

By using a language with an extremely rigorous semantics, the customer, developer, and external certification authority share an agreed basis for specification analysis. Indeed, the level of assurance sought by BASE for the trusted gateway required the use of a formal specification language. The formal statement of the trusted gateway's security policy has been studied by an external evaluation authority, who reported that it would meet the requirements for certification at the level of assurance required were it submitted as part of a product development.

A second benefit of using a language with formal semantics is that doing so opens the way to exploiting proof when the engineers feel it will be valuable or when seeking higher levels of assurance demand it, as is already the case with some safety-critical software. When that happens, we expect to introduce proof gradually into the development process by first concentrating on validation of critical properties and then eventually moving on to verification of

refinement steps.

Although the formality of VDM-SL played an important role in the gateway's development, it did not greatly affect the engineers' use of the language. The formal semantics were not looked into directly and the training given was similar to that which would be available for an informal language with the same abstraction features. This suggests, but does not prove, that the costs of using a formal language as the basis for system design are not excessive.

Tool support. Formal specifications need not be executable, but they can be checked for syntax and some semantic errors and, when they are executable, they can be tested. Tool support for some of these aspects is still emerging, but the tool set used in this project — the IFAD VDM-SL Toolbox^{12,13} — supports syntax checking, extensive static semantic checking, LaTeX prettyprinting, test-coverage analysis, interpretation, and source-level debugging.

The toolbox's interpreter for executable specifications proved valuable in allowing specification tests right from the early system-development phases, as well as increasing the rate at which the engineers acquired skill in the specification language. By the end of the development, we felt that the use of this tool was central to the project's success in introducing formal techniques. During the formal development, the interpretation facility proved particularly valuable in guiding the engineers' understanding of the language. The test-coverage feature allowed selection of test cases such that all parts of the model were exercised.

It has almost become a truism in the formal-methods community that the availability of high-quality tools is essential for the successful introduction of formal specification into documented development processes on the industrial scale.^{14,15}

Training needs. The study also gave us



some understanding of the training needed to introduce formal specification into the development process. We found that a basic, one-week course in formal specification and the IFAD VDM-SL Toolbox was sufficient for using formal specification to the extent necessary in this project. Short one- or two-day supplementary courses would better meet the specific needs of software designers and implementors. Adding training in proof would introduce a much greater overhead and was not required to meet the assurance levels required by this project.

Our study at BASE found that introducing formal specification into a development process has a training overhead that is typical of any new technology introduced into a company. We found this encouraging. The engineers, who in general had no background in formal specification, found its application straightforward. Consultant support from an expert user was essential when the engineers started to apply the technique. In the longer term, we also see a modest level of such support as valuable in improving specification skills: raising abstraction where appropriate, becoming familiar with more of the language, and so on.

Introducing new technology. The introduction of a new technology to the development process is not simple. Those involved must be convinced that it adds to rather than detracts from their established way of working. In most cases, as engineers learn a new technology they perform more slowly the task it affects.

The learning delay became particularly obvious during the software-design phase, when the engineer initially developed the software design using conventional pseudocode before he did so using the formal notation. Only over time will a new technology's full benefits be realized — a concern that is not limited to formal specification but that applies to any new technology.

Development process. The use of formal specification can complement structured or object-oriented development methodologies. Frequently, specifications for large systems are so complicated you cannot analyze them until they have been decomposed into functional blocks, data blocks, or objects. Formal specification can most advantageously enhance certain development processes, such as obtaining a detailed understanding of the critical parts of the system and the data used.

The evolutionary rather than revolutionary approach we took in this study helped us easily integrate formal specification into the existing development process.

BASE's future. As a result of performing this study, BASE now has a nucleus of personnel who have been exposed to formal specification and can apply their new skills where appropriate in other project teams. To promote the use of these techniques within the company, we have planned a series of internal presentations that describe the results of the study and where formal specification is applicable in the development process.

BASE expects to gradually change its development process as a result of this study. This change will permit the use of formal specification together with other techniques to increase the rigor of system specifications — mainly for applications with critical functionality, at least initially. BASE has already applied VDM-SL in the development of a subsystem in a larger secure system that required a high level of evaluation.

The BASE study set out to provide evidence on the effects of introducing formal specification in projects that could benefit from the use of a formal language, such as those that must attain high assurance levels. We did not intend to confirm

or deny the value of formal specification generally. To do so would require a large-scale "clinical trial," not a sample-size one! Nevertheless, BASE feels that we have some evidence on which to base a modification to its existing development processes that will encourage analysis at early development stages. We offer some final conclusions:

- ◆ Formal specification's use in our development process did not impose a significant cost or time overhead across the entire development.

- ◆ Formal specifications can improve understanding of the system being developed and can prevent errors from being propagated through the development process.

- ◆ Formal specification can be integrated gradually into a traditional, existing development process.

- ◆ Formal specification's benefits are mostly obtained in the early stages of development.

**Developers must
be convinced that
new technology
adds to rather than
detracts from their
established way
of working.**

- ◆ Formal techniques can be used by engineers after as little as one week's training, if expert support is available when they first apply the techniques.

- ◆ Formal-specification notation must be supported by industrially applicable computer-based tools that provide specification interpretation, which allows test-case examination. ◆

ACKNOWLEDGMENTS

We acknowledge the excellent work of the BASE engineers involved in this project. We thank the European Commission (ESSI Grant 10670) for its support. John Fitzgerald thanks the UK Engineering and Physical Sciences Research Council for its support

under the terms of an EPSRC Research Fellowship. We thank Hanne Christensen and Michael Hinchey for commenting on an earlier draft of this article and the editor and referees for their constructive comments.

REFERENCES

1. N. Fenton, "How Effective are Software Engineering Methods?" *J. Systems and Software*, 1993, pp. 141-146.
2. J. Dawes, *The VDM-SL Reference Guide*, Pitman, London, 1991.
3. D.J. Andrews et al., *Information Technology — Programming Languages. Their Environments and System Software Interfaces — Vienna Development Method-Specification Language Part 1: Base Language*, ISO Draft International Standard 13817-1, New York, 1995.
4. *Information Technology Security Evaluation Criteria*, Office for Official Publications of the European Community, Luxembourg, 1991.
5. J.S. Fitzgerald et al., "Formal and Informal Specifications of a Secure System Component: First Results in a Comparative Study," *Formal Methods Europe '94 — Industrial Benefit of Formal Methods*, Springer-Verlag, Berlin, 1994.
6. J.S. Fitzgerald et al., "Developing a Security-Critical System using Formal and Conventional Methods," in *Applications of Formal Methods*, M.G. Hinchey and J.P. Bowen, eds., Prentice-Hall, Englewood Cliffs, N.J., 1995, pp. 333-356.
7. J. S. Fitzgerald, *ESSI Project ConForm: Home Page*, <http://www.cs.ncl.ac.uk/research/csr/projects/Conform.html>, 1994.
8. J. P. Bowen and M. G. Hinchey, "Ten Commandments of Formal Methods," *Computer*, Apr. 1995, pp. 56-62.
9. A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, Sept. 1990, pp. 11-19.
10. J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, Sept. 1990, pp. 8-24.
11. *Applications of Formal Methods*, M.G. Hinchey and J. P. Bowen, eds., Prentice-Hall, Englewood Cliffs, N.J., 1995.
12. R. Elmstrom, P. G. Larsen, and P. B. Lassen, "The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications," *ACM Sigplan Notices*, Sept. 1994, pp. 77-80.
13. P. Mukherjee, "Computer-Aided Validation of Formal Specifications," *Software Eng. J.*, July 1995, pp. 133-140.
14. D. Craigen, S. Gerhart, and T. Ralston, "Formal Methods Reality Check: Industrial Usage," *IEEE Trans. Software Eng.*, Feb. 1995, pp. 90-98.
15. J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods," *IEEE Software*, July 1995, pp. 34-41.



Peter Gorm Larsen is currently working as a research scientist at the Institute of Applied Computer Science where he is in charge of the R&D group working on formal methods. He is also taking an active part in the ISO (and BSI) standardization of VDM-SL. He has

served on several program committees and has been a reviewer for several journals and conference publications. He is a consultant in using VDM and has developed and presented VDM-SL courses for several industrial organizations. He is also the author of numerous journal and conference publications.

Larsen received an MSc in computer science and electronics and a PhD in computer science from the Technical University of Denmark. He is a member of the IEEE Computer Society, ACM, Formal Methods Europe, and British Computing Society — Formal Aspects of Computing Science.



John Fitzgerald is a lecturer in computing science at the University of Newcastle upon Tyne. His current projects include the formal analysis of models of critical computing systems in medical, nuclear, and aerospace applications. He has previously worked as a

researcher at the University of Manchester, studying proof-support systems for model-oriented specification, and at the British Aerospace Dependable Computing Systems Centre at Newcastle on the formal modeling of timing properties of complex avionics systems. He undertook the work for this article as a research fellow with the UK Engineering and Physical Sciences Research Council. He has published many papers and co-authored a recent book on proof in VDM.

Fitzgerald received a BSc and a PhD in computing from the University of Manchester. He is secretary of Formal Methods Europe, a member of the ISO/BSI working group on formal specification languages and the British Computer Society specialist group on formal methods, and a member of the IEEE Computer Society.

Tom M. Brookes designs secure communications systems and information systems for British Aerospace Systems and Equipment. Previously he was employed at the University of Manchester in the design and maintenance of microwave radio astronomy systems and at British Aerospace Dynamics as a microwave engineer.

Brookes received a BSc in physics from Bristol University, England and a PhD degree in astronomy from Manchester University, England. He is a member of the IEEE.

Address questions about this article to Peter Gorm Larsen at IFAD, Forskerparken 10, DK-5230 Odense M, Denmark; peter@ifad.dk; or John Fitzgerald at Centre for Software Reliability, Bedson Building, University of Newcastle upon Tyne, NE1 7RU, UK; John.Fitzgerald@ncl.ac.uk.