Lessons Learned from Applying Formal Specification in Industry*

Peter Gorm Larsen

The Institute for Applied Computer Science (IFAD)
Forskerparken 10
5250 Odense M, Denmark,

John Fitzgerald

Centre for Software Reliability
Bedson Building
University of Newcastle upon Tyne
Newcastle, NE1 7RU, UK

and Tom Brookes

British Aerospace (Systems and Equipment) Ltd
Clittaford Road
Plymouth
PL6 6DE, UK

Introduction

Industrial software developers are faced with a bewildering array of software engineering techniques, each with its own promised benefits. Yet the choice between them is often informed more by what Fenton calls "the unsubstantiated advertising claims and biases of producers, both academic and industrial" [4] than by evidence of the costs and benefits of these techniques in an industrial setting. Companies are sometimes reluctant to be the first to enter new territory: without a sufficiently large public body of such evidence new techniques may be considered too immature for commercial adoption. Many industrial companies have already invested a great deal of effort in the 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 where the overall effects can be identified and quantified. In this paper, we report on the lessons learned during a study of one such change on the software development process at British Aerospace Systems and Equipment Ltd. (BASE) in the UK.

At BASE, there was interest in developing a security-critical system to levels of assurance at which the use of formal specification for modelling the security policy was mandated. The purpose of our study was to provide evidence on the effect of introducing a modest amount of formal specification into an existing development process applied to this system. Note that the study was not an attempt to prove the cost-effectiveness or technical value of formal meth-

*Published by a special issue of "IEEE Software" about "Lessons Learned", May 1996.

ods generally. Also, our aim was to introduce a relatively minor "delta" to the development process, so there was no stress on formal proof, our use of a specification language being largely for system and software modelling.

The study itself consisted of the parallel development of a system component known as a trusted gateway by two separate teams of engineers. One team employed the conventional BASE development methodology using structured analysis with CASE tool support (referred to as the *conventional path* below). The other team followed the same design process, but used formal specification wherever it was felt appropriate (referred to as the *formal path* below). Procedures, such as ensuring that the development teams were located apart and instructed not to communicate about the project, were enforced to maximise their independence. The structure of the study was chosen so as to ensure that the only difference in the technical approaches employed in the two paths was the use of formal specification. This gave us the opportunity to observe the effects of adding formal specification to the development process in terms of cost and in terms of the qualities of the designs produced.

The system under development was a small but security-critical message-handling device called a *trusted gateway*. The gateway acts as a filter between a high-security system and lower security 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 meant that there was some incentive to use formal specification techniques to model at least the central security-enforcing function, namely a high level of assurance under the ITSEC [10] 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.

In this article we do not dwell on the details of the application or the particular specifications developed because this is already reported elsewhere [6, 7, 5]. Instead we will focus on the lessons learned during each phase of the development.

The 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. The development teams consisted of one engineer in each phase, a new engineer taking over whenever a new phase started.

In addition to the two parallel developments of the system, a central authority acted as customer and monitor, keeping records of the queries, problems and progress made on the project. This monitoring team recorded observations, including metrics, at different points during the development. Below we present the lessons learned from the comparisons of these observations phase by phase. We then discuss lessons common to all the phases and the distribution of human effort expended across each of the two paths. Finally, some conclusions are drawn from the study as a whole.

The lessons described in this article are informed by our participation in this and other industrial applications of formal methods, and not the BASE study alone.

The system analysis phase

The system analysis phase begins when the systems engineer is presented with a *customer requirement*, a natural language document produced by the client detailing the product requirements. This is transformed into a collection of numbered clauses of roughly equal complexity. The systems engineer then begins to produce a model of the proposed system, represented in a CASE tool, in this project Teamwork[1]. The engineer on the formal path chose additionally to produce a formal specification in VDM-SL of the data types (recorded in the Teamwork data dictionary) and principal

[1] Teamwork is a Registered Trademark of Cadre Technology Inc.

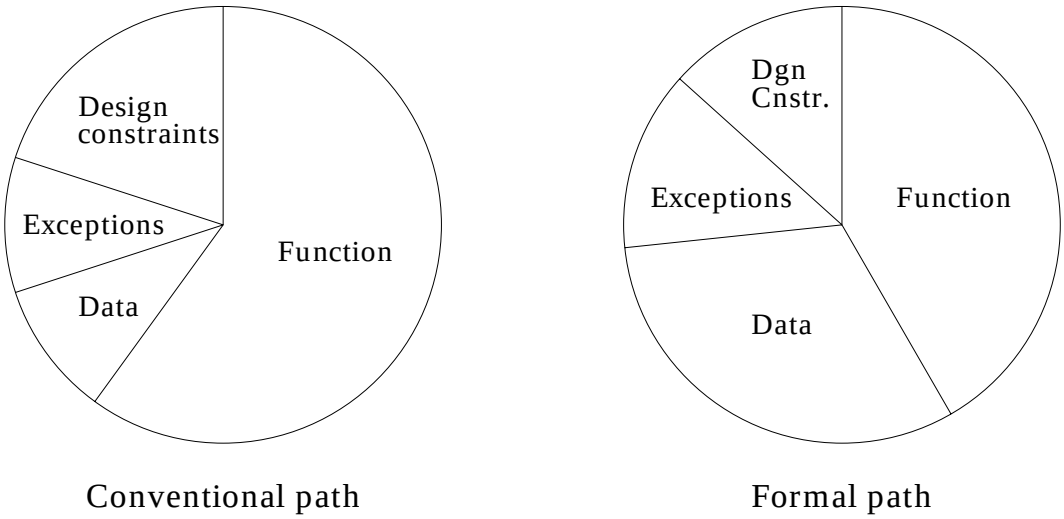

Figure 1: Classifications of queries

functions of the system (recorded in the process specifications).

In the course of analysing the system and producing a model of it, queries are normally raised against the customer requirement. In the trusted gateway study these were submitted to the monitoring authority in writing and written answers were returned, allowing a log of the questions and answers to be maintained so that differences between the two development paths could be analysed.

At the end of the phase a system specification and system test plan were produced by both development teams. These documents were reviewed in each path, with feedback to the separate developers. A third comparative review was performed by the monitoring team, which kept a log of exposed deficiencies. No feedback from the comparative review went back to the developers. The engineers employed in this phase were considered by BASE to be equally qualified with a similar background.

Queries against customer requirements

What difference did the use of formal specification make to this system analysis phase? We used the log of queries as a source of information on how well the two teams had tackled the problems of misstated, incomplete or ambiguous requirements.

The engineer on the conventional path submitted approximately 40 questions; the engineer using formal specification submitted approximately 60. The actual queries were given to five other engineers drawn from the systems, software and quality departments who were asked to assign each query to one of the following categories, without knowledge of the query's source:

- **Function:** clarification of the function the system has to perform under normal circumstances;
- **Data:** to understand the data, data type or data structure better;
- **Exception:** to determine what the system has to do under exceptional circumstances;
- **Design Constraint:** to clarify design constraints under which the system was to be developed.

The difference in the pattern of queries can be seen in Figure 1. Looking at the relative distribution of queries between the categories in each path, the formal path appears to have

focused more on the data and the exceptional conditions than the conventional team. In particular the emphasis on the exceptional cases is considered valuable in the requirements capture process of security-critical systems.

The engineer's view

The systems engineer on the formal path was interviewed to obtain his impression of using formal specification in the system analysis phase. He stated that the main difference between this system design phase and that of a conventional development was the higher level of abstraction of the model produced. He also noted that the early identification of the data needed by the system highlighted the boundary cases. On the negative side he stated that the skill of abstraction is difficult to acquire and needs experience, and that the analysis process was more time consuming as more work was needed to produce the formal specification in addition to the basic Teamwork model.

Lessons from the system analysis phase

- *Effort is concentrated in areas susceptible to design errors.* The definitions of data and exceptional behaviour have been sources of design error in message processing systems. A method, formal or informal, which encourages concentration on those areas could lead to improvements in eventual product quality. This was illustrated clearly in the project during the comparative review at the end of the system analysis phase. We realised then that the engineer on the formal path had raised some queries to clarify the gateway's behaviour under a particular combination of conditions which turned out to constitute an exception, but which had not been identified as such in the original requirements document. The formal specification had therefore been 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 which emphasises data description. A language which, for example, stresses communication between processes 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*, allowing the engineer to concentrate on defining exactly what the system has to do rather than how it should do it. BASE generally considered this a good thing. However, project management needs to take this into account and not interpret time spent on production of an abstract formal model as project slippage. The phenomenon may be familiar from the introduction of structured design methodologies.

- *Expect specification style to be immature at first.* The engineer was able to develop an abstract system specification, although a tendency towards using familiar constructs instead of exploiting the richness and variety of the formal notation was noted. 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 analyse the system more thoroughly. He considered that the ability to test a specification was essential for gaining confidence in its correctness.

Software design phase

From the system specification produced in the first phase, a more detailed model of those parts of the system to be implemented in software are produced by specialist software engineers. The data structures employed may be more concrete and the functionality is usually described in detailed pseudo-code.

In the BASE study, this phase resulted in software designs and software test plans from both development teams. The formal path produced VDM-SL versions of type definitions and procedural specifications of the functionality. The documentation was again reviewed and compared by the monitoring team. The engineers available for this phase had different degrees of experience, the engineer on the formal path being more experienced, although he was new to formal specification. This limits the conclusions we can draw from a comparison between the two phases. Below, we describe lessons which are relatively independent of the comparison.

In both paths, far fewer queries were raised against the requirements in this phase than in the system design 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 taken by the engineer implementing the software. Whether or not this is desirable is a contentious issue and it is determined by the skill of the engineer who is implementing the design.

Lessons from the software design phase

- *Some duplication of work can occur.* The engineer on the formal path followed his normal design process, producing pseudo-code for the proposed functions before he 'translated' this into VDM-SL for testing using the tool support. This is obviously not the most efficient method to use, but it was comfortable for the engineer in question. We feel that there are two possible causes for this: either the training has not given him the appropriate skills for refining the specification towards implementation or the engineer was relatively experienced and felt that the new technology did not improve his current way of working.

- *Testing functionality may encourage improvements to original requirements.* If engineers in the later design phases do not tend to 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 been a contribution to the problem in the conventional development path. If testing of functionality is possible through abstract modelling, as in the formal path, there may be greater opportunity to identify exceptions.

- *Train appropriately for different development roles.* We originally felt that the same training course would be appropriate for systems and software engineers alike. Now we would revise that view. The systems engineer's task was to establish the logical abstract description of the functionality of the system as a whole. This requires skills in abstraction, functional specification and data specification. In the software design phase the engineer takes the abstract logical specification and refines it towards implementation. This requires additional skills in functional refinement. Implementation aspects of the design also need to be considered.

Implementation Phase

In the implementation phase, the detailed software design was hand-crafted into running code

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

Figure 2: Code size for the final kernel implementation

Performance measure	Conventional Path	Formal Path	*Ratio*
Initialisation time (seconds)	17	70	4
Processing rate (chars per sec)	18	250	13.8

Figure 3: Performance of the final implementations

in C using the Microsoft Visual C++ compiler on a PC platform. A common user interface for the trusted gateway software was designed and given to both implementation teams. This permitted a "blind" comparison of the two products, in which it was not possible to distinguish them by examination of the screen layout. The developed code was tested 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, each of the final implementations was run against the test suites developed in the other path. The conventional path failed several of the tests developed in the formal path, although the root cause of the failures was the same: the condition which caused this fault had been detected in the formal path in the system analysis phase, but had never been detected in the conventional path. Had this been a commercial product, the error would have been detected in service. A software patch was written to correct the problem. However, had the design been subjected to external evaluation as a secure product, the documentation for all of the phases would have to be amended, and a re-evaluation of the design carried out. This would involve repeating a significant proportion of the original development effort. Although no detailed quantitative estimate can be made, it is believed that this could double the evaluation costs, with overall development costs increased by 20% to 30%.

Code Complexity

The source code of the kernel routine, which implements the security enforcing function, was compared between the two paths. The McCabe Complexity of this routine was found 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 software patch was produced showed that there had been a significant leap in complexity when the code was re-written to correct the deficiency, the McCabe index increasing roughly from 10 to the 74 measured. This result suggests that conventional development does not in itself produce more complex code, but complexity can be introduced when the code is revised at the end of a development in order to correct problems discovered late in the testing process.

Code Size

The number of lines of code in a routine is not a particularly meaningful metric, though 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 Figure 2 indicate that the routine produced by the formal path is very 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 cannot

be attributed solely to the use of formal specification: they could also be an indication of the ability of the implementor.

Code Performance

The speed of operation was tested by passing a large block of messages through the system. To minimise external factors, the software was installed on the same machine for each test, and all other applications disabled. The results are shown in Figure 3.

The formal implementation spent roughly four times as much time on checking the system data as the conventional implementation. However, when the system is processing messages it is almost fourteen times faster. For a trusted gateway, which is designed to be set up once and then left to operate for a long period of time, the relative speed of initialisation is not a major factor in assessing system performance. The software developed in the formal path would thus be considered to be much faster than the conventional implementation. The difference in performance identified here is again partly attributable to the software patch.

Lessons from the implementation phase

- *Errors detected late in the design process are expensive to correct.* The corrections may result in code which is poorly supported by design documentation and is difficult to maintain.

- *An abstract software design means implementors have some design decisions.* The abstract nature of the software design produced in the formal path left some design decisions to be made by the implementing software engineer. This posed some problems for the software engineer concerned who was more used to implementing software which had already been designed to the point of only requiring final coding. It was particularly a problem that the engineer had not been trained in the use of VDM-SL.

- *Training in the use of formal specification is essential for an engineer who is going to implement from a formal specification.* The ability to read a specification is not sufficient, the training must encompass the concepts behind the use of formal specification. Emphasis should be placed on the freedom which is left in the formal specification to be resolved by the implementor.

The view overall

We embarked on this study in the hope of observing the effects that the introduction of formal specification would have on parts of the development process for secure message processing systems at BASE. Having looked at the lessons we learned in each phase, we turn to the conclusions drawn from the development process as a whole, beginning with the costs associated with the formal specification techniques employed.

Cost Effectiveness

An overview of the cost distribution in the two development paths is given in Figure 4. If we restrict the effort spent during the system analysis phase to the actual development (i.e. removing effort spent for reviews etc.) the formal path required roughly 25 % more person-hours than the conventional path. Neither engineer was limited by available resources as both under-spent the budget, by 25 % in the conventional path, and 12.5% in the formal path. The time spent on training and external consultancy have not been included.

In the software design phase the engineers had different skills and experience, the engineer in the formal path having more experience. The formal path required less effort in order to complete the task, but it was felt that this result was biased by the difference in the engineers' experience. When this was taken 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.

Phase	Allocated Time	Conventional Path		Formal Path	
		% Alloc	% Project	% Alloc	% Project
System Analysis	40 %	30 %	33 %	35 %	43 %
Software Design	40 %	40 %	47 %	33 %	41 %
Implementation	20 %	17 %	20 %	13 %	16 %
Totals	100 %	87 %	100 %	81 %	100 %

Figure 4: Cost distributions compared

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 re-work which increased the effort required by roughly 15%. Both tasks were completed within the allocated budget. As a percentage of the overall project, the formal software implementation took 13% of the effort as compared to 17% in the conventional path.

Comparing the effort over the entire development, the formal path did not incur an overhead. In fact the overall effort required was slightly less than that of the conventional development, but the difference is not felt to be significant. This result tends to confirm conventional wisdom that using formal specification adds to costs the early parts of the development process where system requirements are being analysed and understood, but that the additional effort is recovered in the later stages of the development process. This change in the effort profile is also typical of the introduction of structured design methods where systems understanding is promoted before development.

Where Formal Specification is Applicable

For the classes of message processing system developed by BASE, the use of formal specifications in a development process which is controlled by a structured methodology can be divided in two parts: data modelling and definition of functionality. Using formal specification in data modelling was felt to be valuable: data definitions written in a common non-implementation-specific manner allow this information to be passed through the various development phases without transformation, although detail is added, until the system is implemented. The rigour required to define the data using a formal notation also forces the designer to question the customer (or to make and justify assumptions about the data) very early in the development process. Formal specification of functionality was not considered advantageous when the required function could be written down in an unambiguous manner, or specified using existing well-understood processes. It is necessary to determine when it is advantageous to use formal specification and this is also pointed out in [2].

Based on this experience, BASE concluded that formal specification is useful when:

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

The Specification Language

This study used the VDM Specification Language which is currently being standardised by ISO (see the box on page 9). It is believed that similar results could be obtained with the use of other model-oriented formal specification languages which have tool support for interpreting an executable subset of the notation.

Formal Specifications and VDM-SL

System specifications are usually presented in a mixture of natural language, graphical notations and pseudo-code. A *formal* specification by contrast uses a very precisely defined mathematical language to describe the properties of the desired system. This leaves less room for ambiguity in the requirements, and also opens the way to automatic checking for many kinds of error and inconsistency in requirements. Formal specification languages usually provide a variety of very abstract high-level constructs which permit the description of system properties without dealing with implementation details.
The formal specification language used in this project was the "Vienna Development Method" specification language VDM-SL, currently in the last stages of standardisation under ISO.
Further References

1. J. Dawes, "The VDM-SL Reference Guide", Pitman, 1991, ISBN 0-273-03151-1.
2. "*ISO Draft International Standard: Information Technology Programming Languages – VDM-SL*", June 1995. For information on the standard, contact the Working Group Chairman, Derek Andrews (derek@mcs.le.ac.uk).

There exists a variety of different categories of specification languages which are suitable for different application areas [8, 11]. Experience from other applications illustrates that the choice of specification language is crucial for the success of the given application [9]. The notation must focus on concepts which are fundamental to the given application domain.

The engineers preferred to use an ASCII representation of VDM-SL over the more commonly used mathematical notation, though it is more verbose. This is probably motivated by the fact that the tool used the ASCII representation, but the engineers also claimed that it eased their presentation of formal specifications to colleagues who are not familiar with the notation. The lesson to be learned from this is 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 modelling 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 have an agreed basis for the analysis of specifications. 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 report 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 effect of using a language with formal semantics is that the way is now open to exploiting proof when the engineers feel it will be valuable, or higher levels of assurance sought 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 eventually moving on to verification of refinement steps.

Although the formality of VDM-SL was an important factor in the gateway's development, it did not greatly affect the use made of the lan-

A small example of VDM-SL

VDM-SL is a *model-oriented* specification language. This means that the specification of a system is mediated by a mathematical model in which data types represent the classes of input and output values, and modelling the system's internal state. System functionality is modelled 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 modelled 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 10000 characters. A function on messages might return a Boolean value **true** if one message is a sub-message 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)
```

Note that the result of this function is defined 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 are available from the VDM Examples Repository on the World-wide Web at `http://www.ifad.dk/examples/examples.html`

guage by the engineers. 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

The IFAD VDM-SL Toolbox was used to support the development (see the box on page 11). Its interpreter for executable specifications was valuable in allowing specification test right from the early system development phases, as well as increasing the rate at which skill in the specification language was acquired. 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. It has almost become a truism in the formal methods community that the availability of high quality tools are essential for the successful introduction of formal specification into documented development processes on the industrial scale [3, 1].

Training Needs

The study also gave us some understanding of the training requirement needed to introduce formal specification into the development process. A basic one week course in the use of formal specification and the IFAD VDM-SL Toolbox was found to be sufficient. Short one or two day supplementary courses are needed to meet the the specific needs of software designers and implementors. Adding training in proof would introduce a much greater overhead.

We were encouraged by the indication that introducing formal specification into a development process has a training overhead which is typical of introducing any new technology into a company. The engineers, who in general had no background in this technique, found it straightforward to apply. Consultant support from an expert user was found to be essential when the engineers were starting to apply the technique. In the longer term, we also see

Tool support

Formal specifications are descriptions of the desired behaviour of a system. As such, they need not necessarily 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, supports syntax checking, extensive static semantic checking, LaTeX pretty printing, test coverage analysis, interpretation and source-level debugging. The latest release incorporates a C++ code-generator, but that was not available for the study described in this article.
During the formal development, the interpretation facility proved to be particularly valuable in guiding the engineers' understanding of the language. The test coverage feature turned out to be important in the selection of test cases such that all parts of the model were exercised.

Further References

1. R. Elmstrøm, P.G. Larsen and P.B. Lassen, "The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications", *ACM Sigplan Notices*, September, 1994.
2. P. Mukherjee, "Computer-aided Validation of Formal Specifications", *Software Engineering Journal,* 10(4):133–140, July 1995.

a modest level of such support as valuable in improving specification skills (raising abstraction where appropriate, becoming familiar with more of the language etc.)

Introducing New Technology

The introduction of a new technology to the development process is not simple as those involved need to be convinced that it adds to rather than detracts from their own way of working. In most cases, there is a reduction in the speed of performance of a task while the technology is learned and only over time will the full benefits be realised. This point needs to be considered when any new technology is introduced and it is not specific to formal specification. This was particularly clear in the software design phase of the project where the engineer initially developed the software design using conventional pseudo-code before using the formal notation.

The Development Process

The use of formal specification can be seen to be complementary to structured or object-oriented development methodologies. Specifications for large systems are frequently so complicated that until they have been decomposed into functional blocks, data blocks, or objects, it is difficult to obtain a sufficient understanding of the problem in order to be able to analyse it. Formal specification can be introduced as an enhancement to the development processes to be used where it is most advantageous: i.e. in obtaining a detailed understanding of the critical parts of the system and the data used.

The evolutionary rather than revolutionary approach taken in this study helped to integrate the use of formal specification into the existing development process easily.

The Future inside BASE

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

BASE expects to gradually change its development process as a result of this application study. This change will permit the use of formal specification together with other techniques to increase the rigour of system specifications. It is expected that this will initially be used mainly for applications with critical functionality.

Concluding Remarks

The study at BASE set out to provide evidence on the effects of introducing formal specification where the motivation to use a formal language already existed, i.e. the desire to reach high assurance levels. It was not our intention to confirm or deny the value of formal specification generally: to do so would require a large-scale "clinical trial", not a sample size of one! Nevertheless, BASE feel that we have some evidence on which to base a modification to existing development processes to encourage analysis at early development stages. We offer some final conclusions:

1. Using formal specification in this development process did not impose a significant cost or time-scale overhead across the whole development.
2. Using formal specification can improve the understanding of the system which is being developed and can prevent errors being propagated through the development process.
3. Formal specification can be integrated gradually into a traditional existing development process.
4. The majority of the benefits are obtained by using formal specification in the early stages of the development.
5. Engineers can begin to use formal techniques with as little as one week's training provided expert support is available when they first apply them.
6. It is important that the formal specification notation employed is supported by industrially applicable computer based tools which provide specification interpretation to allow test cases to be examined.

Acknowledgements

The authors would first like to acknowledge the excellent work of the BASE engineers involved in this project. We are grateful for the support of the European Commission (ESSI Grant 10670). JSF is grateful to the United Kingdom Engineering and Physical Sciences Research Council for support under the terms of an EPSRC Research Fellowship. We would like to 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] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(3):34–41, July 1995.

[2] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, 28(4):56–62, April 1995.

[3] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal Methods Reality Check: Industrial Usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[4] N. Fenton. How Effective are Software Engineering Methods? *Controversy Corner in the Journal of Systems and Software*, 22:141–146, 1993.

[5] John S. Fitzgerald. ESSI Project ConForm: Home Page. World wide web at URL http://www.cs.ncl.ac.uk/research/csr/projects/ConForm.html, 1994.

[6] J.S. Fitzgerald, T.M. Brookes, M.A. Green, and P.G. Larsen. Formal and Informal Specifications of a Secure System Component: first results in a comparative study. In *Formal Methods Europe '94 – Industrial Benefit of Formal Methods*. Springer-Verlag, 1994.

[7] J.S. Fitzgerald, P.G. Larsen, T.M. Brookes, and M.A. Green. *Developing a*

Security-critical System using Formal and Conventional Methods, chapter 14. In [9]

[8] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[9] Michael G. Hinchey and Jonathan P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall, 1995. ISBN 0-13-366949-1.

[10] Office for Official Publications of the European Community. *Information Technology Security Evaluation Criteria*, June 1991.

[11] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.