Quantitative Analysis of an Application of Formal Methods

Juan Bicarregui[1] Jeremy Dick[2] Eoin Woods[3]

[1] Rutherford Appleton Laboratory, Oxfordshire, UK.
[2] Now at B-Core UK. Work done whilst at Bull Information Systems.
[3] Now at Sybase. Work done whilst at Bull Information Systems.

Abstract. This paper reports on the experience gained in the MaFMeth project, which undertook a formal development with tool support for several parts of the life cycle from requirements capture through to C code generation. We explore the hypotheses that formal methods enable the early detection of faults in design by examining the development process in the light of the stages at which faults were introduced and discovered.

1 Introduction

One of the major planks of the argument for formal methods is that they allow fewer design errors to be introduced in to software and allow remaining errors to be identified early in the development process, so minimising the cost of correcting them.

The process of formalising the specification encourages certain kinds of question to be asked of the requirements, raising issues which may not otherwise have been manifest until later in development. Having a machine-processable description at an early stage allows certain validation and verification activities not otherwise available, such as animation which serves to validate the specification against intended requirements, and proof of consistency which verifies the internal coherence of the specification. The ability to have precise discussion about the relationship of the specification to an implementation is an important enabling factor, since both descriptions are formalised, each design layer can be compared and verified against the layer above.

The longer a fault in design goes undiscovered, the more rework is necessary when it is found, and so the more expensive it is to correct. In the worst case, when a fault in the original specification is not found until after the product is delivered, in order to ensure the integrity of the product, the specification, design, implementation, testing, integration and delivery stages all have to be reworked. If that fault had been discovered early on the design phase, far less expense would have been involved in its rectification.

Very little quantitative evidence has been published to substantiate the supposed benefits of the formal approach. This paper reports quantitative aspects

of the experience gained in the MaFMeth project[4] which used a combination of VDM and the B-Method and kept certain metrics with the aim of showing evidence for the early detection of errors and assessing the relative effectness of various activities in the overall process. These metrics amount to fault counts correlated against the activities during which the corresponding errors were introduced and discovered. For further descriptions of the system developed, the techniques used and a qualitative description of the problems encountered see [9], [10] and [1].

2 The Application

The project in question was the development of the second release of Groupe Bull's FlowBus^{TM} product. FlowBus is an application integration product (of the type often known as "middleware") which allows applications to communicate in a number of ways via a single application programming interface. Its primary function is to provide distributed, multi-platform, inter-application message handling services involving message routing, storage, transformation and enrichment, all under administrator control, transparently to the applications.

FlowBus is intended to serve the needs of corporations with requirements driven by business in the area of inter-application communication, or wishing to restructure their business processes. Such environments are characterised by the need for extensive, flexible, inter-application communications that can be altered with the minimum of disruption. FlowBus is intended to provide the integration services to allow new applications, legacy applications and package software to be integrated to meet the needs of these environments.

FlowBus allows applications to communicate without explicit knowledge of each other's existence, form or function. Interaction between applications and FlowBus is via the sending and receiving of messages. Messages within FlowBus are typed and FlowBus routes messages between applications according to this type and possibly the message contents. FlowBus is also capable of message enrichment, transformation and conversion and is accessed using a single high-level API across all the supported platforms. Other FlowBus facilities include deferred message delivery (i.e. message queuing) and centralised administration facilities allowing large multi-platform FlowBus networks to be administered from a single workstation.

This development project was centred in the area of control and administration, particularly of queues. The *Queue Administration Tool* (QAT) is able to list the queues in the system, monitor the status of each queue and report it on demand, list the messages within a queue and view or update individual messages when required. When messages are updated, it must allow the administrator to reroute them given certain system-wide integrity constraints. The QAT is also

[4] The MaFMeth project is an application experiment funded under the EC ESSI programme. It is a collaboration between the Bull development centre (Hemel Hempstead), Bull S.A. (Paris), B-Core Limited (Oxford) and the Rutherford Appleton Laboratory.

capable of generating alarms when certain types of messages arrive on certain queues. Which combinations of message types and queues raise which alarms is configurable.

3 The Development Environment

The project was undertaken in a conventional system software development environment consisting of a department of some sixty software developers engaged in all aspects of system software supporting three Unix based software products. The development process used was relatively mature for a Unix system software development process, having been certified as ISO9001 (TickIt) [7] compliant for its quality management system and operating at a point close to level 3 of the SEI Capability Maturity Model [8]. Some use was being made of structured approaches, specifically the Yourdon Structured Method [11], although much of the development was still utilising a less rigorous approach using natural language augmented with informal diagrams. There was no general awareness or understanding of formal methods throughout the development staff.

The FlowBus project involved about twelve software developers, of whom three were involved directly in the development of the QAT subsystem's functional engine using formal methods[5]. Of these three staff, two had previous experience in applying formal methods, but one of these, the primary developer with responsibility for delivery of the finished component, had worked with formal specification only on trial projects and had not previously used formal approaches for product development. In particular he had no prior knowledge of the B Method or its associated tools which were critical to the development process used.

4 The Method

The development process adopted used was influenced by

1. the desire to cover as much of the development life cycle as possible by formal techniques in order to test the "faults discovered early" hypothesis;
2. the nature of the tool support available for each stage of development; and
3. the fact that the resulting code had to be closely integrated with other code both supplied by a third party as an existing package and developed in-house using more traditional techniques.

These three requirements immediately posed potential problems due to unavailability of a single tool supporting all the required features. In order to cover as much of the life-cycle as possible, from requirements capture to integration testing of code, and faced with the lack of a single tool providing all these facilities,

[5] Some consultancy on formal techniques used was also brought in under the ESSI project.

we found it necessary to use a number of different notations, each with their own forms of tool support.

The use of C as a target language was imposed by the development organisation. The B-Toolkit was chosen for its support of development from the design phase to C code generation. The decision to employ VDM rather than the Abstract Machine Notation (AMN) of the B method to capture the initial specification was motivated by three reasons:

- previous experience [2] that AMN encourages the specifier to think very much in terms of assignments to state variables, whereas VDM-SL facilitates the capture of the initial specification at a more abstract level;
- the desire to evaluate the diagrammatic approach to formal specification offered by the "VDM through Pictures" tool [3], and the style of specification imposed by this approach; and
- the desire to evaluate and take advantage of the ability to generate test cases from the VDM specification using the VDM Analysis Tool [4].

Naturally, using different notations introduced concerns about the training of staff and the co-existence and interaction of the various formalisms and tools however this could not be avoided if we were to cover the desired breadth of activities.

For the purpose of the assessing the development process, we identified 13 activities, with varying degrees of tool support. These are depicted in Figure 1. Measurements relating to these activities were taken for two purposes:

1. to compare a formal development process with a conventional one;
2. to compare the relative effectiveness of various stages of the formal process.

To meet the first of these objectives, the results of a number of development projects, all producing sub-products with similar characteristics, were compared[6]. The measurements were made according to the departments existing metrics programme which, for development projects, consists primarily of the following standard metrics being collected:

- number of faults per thousand lines of code found during unit tests and integration tests,
- number of faults per thousand lines of code found during validation test,
- number of faults per thousand lines of code found during customer use,
- person months of effort per thousand lines of code produced.

For these purposes a fault is recorded when a change is required to an design decision made at an earlier development stage. A design made and changed within one stage is not considered a fault.

[6] Though it would have made a more scientifically thorough trial, it was not considered economical to conduct a parallel development of the same component under the different methodologies.

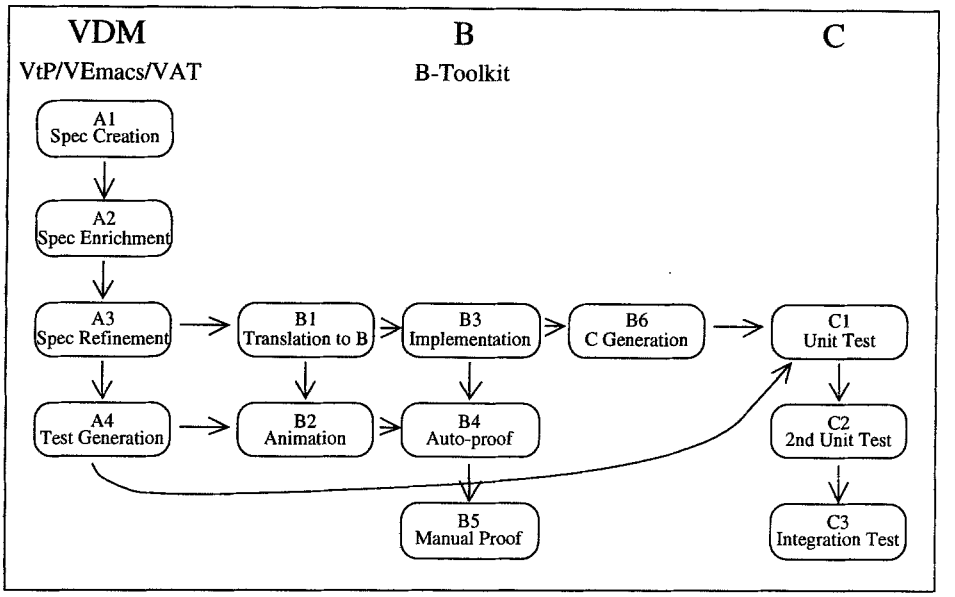

Fig. 1. Development activities identified in the MaFMeth project

To meet the second objective, counts of faults discovered were kept during each of the stages named above. Since some of the activities in the formal process took place in parallel, some faults were discovered at more than one stage. For this reason, each fault *discovery* was given a unique identity, and for each of these a record was made the stage it was first discovered, and the stage it was introduced. From these metrics, it was possible to estimate the effectiveness of each stage of the process in terms of the numbers of faults introduced and detected, though no attempt was made to assess the severity of each fault.

5 Overall Fault count

Despite the differing notations and the lack of integrated tool support described above, quantitative analysis of the overall fault count shows the approach to be very effective both in cost and quality.

Figure 5 compares data from this project with three others undertaken by the user partner using structured design.

The four projects were all developed in the same environment over a period of about 3 years and all used a similar development process apart from the technology involved. All projects were undertaken by engineers from the same development group and all were fragments of much larger developments. All, bar project 2, were new developments, whereas project 2 was a complex modification to an already heavily maintained system software component (hence, perhaps, the low productivity and quality of that development).

	Project 1	Project 2	Project 3	MaFMeth
Application	System software utilities	Transaction monitor modifications	System software application	System software middleware
Approach	Yourdon	Yourdon	VDM / Yourdon	VDM / AMN
Size (LOC)	3000	1100	1300	** 3500
Effort (days)	65	80	27	43
Effort / KLOC	21.5	72.5	20.5	12.5
Faults at unit test	27	17	7	3
Faults / KLOC	9	15.5	5.5	0.9

** *Normalised against amount of library code used. (Total was 8000).*

Fig. 2. Comparison of overall fault count

The LOC figure (Lines of Code) is clearly central to the metrics and for projects 1 to 3 refers to C language statements. For MaFMeth, in all 8000 lines of code were generated, however much of this arose from library components. The figure of 3500 lines of code is the developer's estimate of the amount of code that would have been produced to implement the same functionality without attempting any reuse. In fact, 1200 lines of implementation level B notation were produced to generate the final C code.

None of the effort figures include the learning and technology transfer time which is inevitable in applying new approaches.

The figures show that the MaFMeth project produced, on average, more code per day than any of the previous projects. Of course, this result must be tempered by the different application areas and the possible inaccuracy in the estimate of the equivalent number of lines of code. However, the improvement of nearly 100% is noteworthy.

Even more significant are the results concerning the number of faults at unit test. The unit testing used aimed at 100% functional black box test coverage and 100% branch level white box coverage. This was achieved by identifying test cases using techniques including equivalence partitioning, boundary value analysis and a judicious amount of error guessing! The MaFMeth project produced less than 20% of the faults of the next best project.

Unfortunately, no figures for faults found during validation testing and customer use are available.

6 Early detection of Faults

To explore the "faults found early" hypothesis, we analysed the process adopted according to the time taken, in terms of process stages, to discover faults in the system. We present this data using "Fault Grids" [5, 6] which display the faults found according to the stages at which they were introduced and detected.

6.1 Fault grids

Fault grids provide a means of presenting fault counts against process stages so as to

- highlight the effectiveness of each process stage in terms of the faults introduced and detected;
- assess the overall effectiveness of the process in terms of the number process stages between the introduction and discovery of faults.

The first requirement is to have a well-defined development process. Of interest here is the sequence (temporal and logical) of activities that make up the process. The temporal sequence of activities corresponds to the order in which the activities are carried out. The logical sequence corresponds the dependency of activities on each other; for instance, High-Level Design may be logically dependent on Requirements Capture, but not logically depend on Specification Animation (although animation may take place earlier).

For each fault found, a record is made of

- the activity that enabled the fault to be found;
- the estimate of which previous activity introduced the fault.

For each activity, a record is made of the cost of running that activity.

Figure 1 shows the general scheme for the presentation of fault counts. It shows a hypothetical but typical process consisting of 9 activities from specification through to production. The diagonal grid is used to record numbers of faults found during each activity against the originating activity. Thus, for example, the highlighted lines show that 2 faults were found during Integration Testing that were introduced during Design; adjacent top that we see that 10 faults originated from Coding were found at integration test.

Although hypothetical, the groupings of figures on the grid are typical. The group of figures near the top of the grid reflects the exploratory phases of development were requirements are understood and different specifications are proposed and assessed. The group in the middle correspond to those errors introduced in design and coding. The lonely "2" at the peak of the fault grid should cause concern; it took too long to find these faults!

The diagonals from top-left to bottom-right show the total number of faults introduced at each development stage, for example 3+2+10 faults were introduced during coding. The diagonals from top-right to bottom-left show the total faults found at each review state, here 3+8 faults were found at unit test.

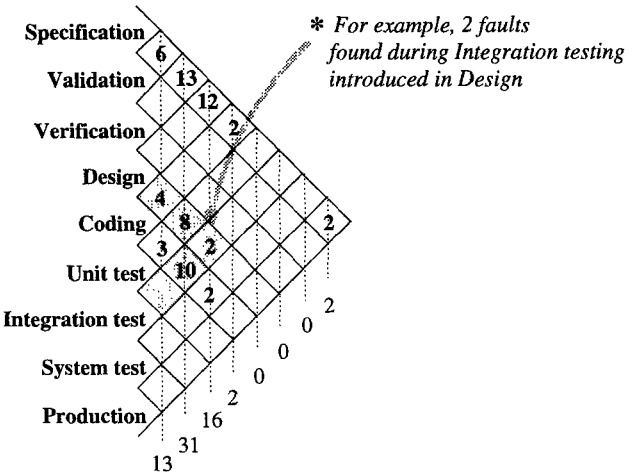

Fig. 3. Hypothetical example of fault grid

The vertical columns in the grid, indicated by the dotted lines, show faults that were found 1, 2, 3, etc. temporal stages down the process. The figures at the bottom of the grid show, therefore, that 13 errors were found after one stage, 31 after two, and so on. However, it may be more interesting to consider at how many logical stages were taken to discover faults. The logical ordering for our example process is shown in Figure 3. With this dependency between activities, for instance, Verification and Design is only one stage down the line from Specification.

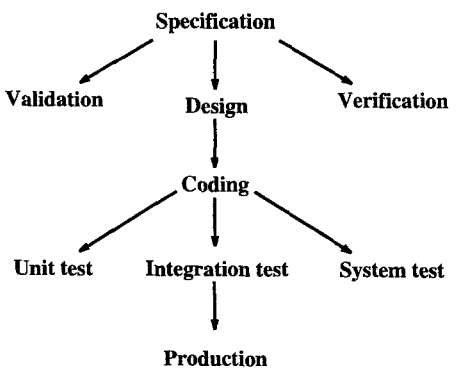

Fig. 4. Hypothetical activities dependences

The number of stages taken to discover faults in our hypothetical development are presented as bar charts in Figure 5. A comparison between the temporal and logical orderings may suggest that a change in the order of the activities would allow faults to be discovered more quickly. Here the activities Validation, Verification and Design all discovered a large number of faults and could all be done immediately aft the initial specification. Note how almost all faults are found after a single logical stage.

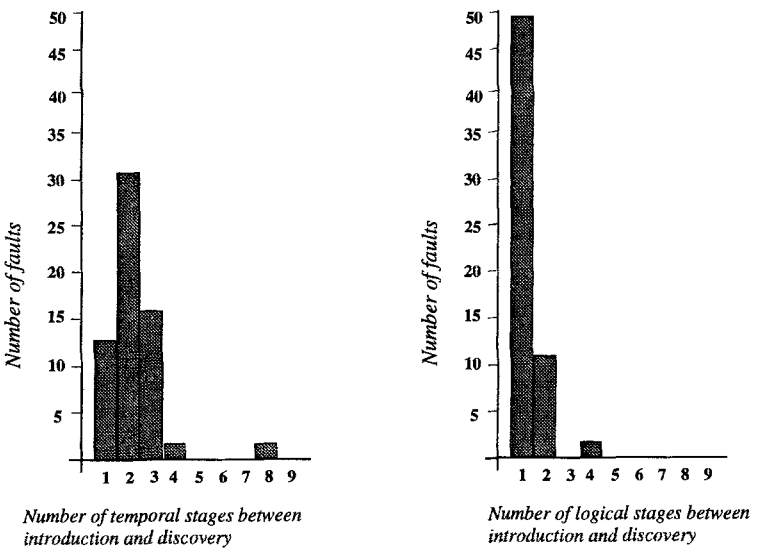

Fig. 5. Bar charts for number of stages between fault introduction and discovery

Of course, some caution must be exercised when considering these charts. Firstly, the columns do not represent the sum of like quantities, each step of the process is not equivalent in magnitude. Secondly, it is not meaningful to compare different project processes by their resulting bar charts as there may be a difference in the granularity of the process decomposition used in each project.

6.2 MaFMeth Results

Figure 6.2 shows the fault grid for the MaFMeth project[7].

Many faults found early. Significant is the tiny number of faults that were discovered during unit and integration testing. The single positive value in the lower part of the grid reflects that only three errors picked up by the testing stages.

[7] Note that in practice the three testing stages were amalgamated into one test suite

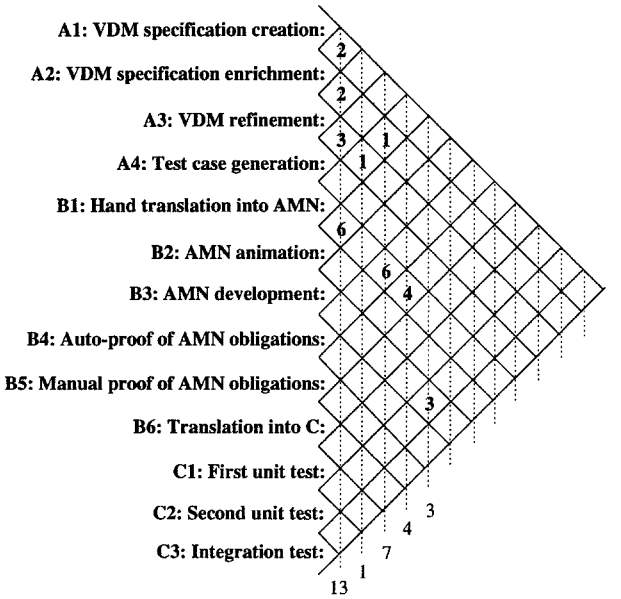

Fig. 6. MaFMeth fault grid

This could well be a consequence of the effort expended in the earlier validation and verification stages.

The figures confirm the tenet of formal methods that formal analysis reduces the number found late in the day. Though several faults were introduced in the early stages, these stages are typically were requirements are being explored and alternative approaches being tried in design. It can be seen as a good thing to introduce and discover faults here. In particular, no faults from the early VDM specifications persisted beyond stage B1.

Faults found quickly. The complete absence of any positive values on the right hand side of the grid is encouraging. It is revealing to note that nearly half the total number of faults were found immediately after their introduction. The number of stages taken to discover faults is summarised in Figure 7.

In this case there is no great difference between the two bar charts. What difference there is can be largely attributed to the temporal ordering of animation and proof stages (B 3,4 and 5). In fact, the logical independence of animation and proof was recognised in advance and these stages were actually carried out in parallel.

When faults were introduced. Highlighted by the diagonal 6,6,4 is the relatively large number of faults introduced by the manual activity of translation from VDM into AMN. 16 out of the total 28 faults were introduced at this stage.

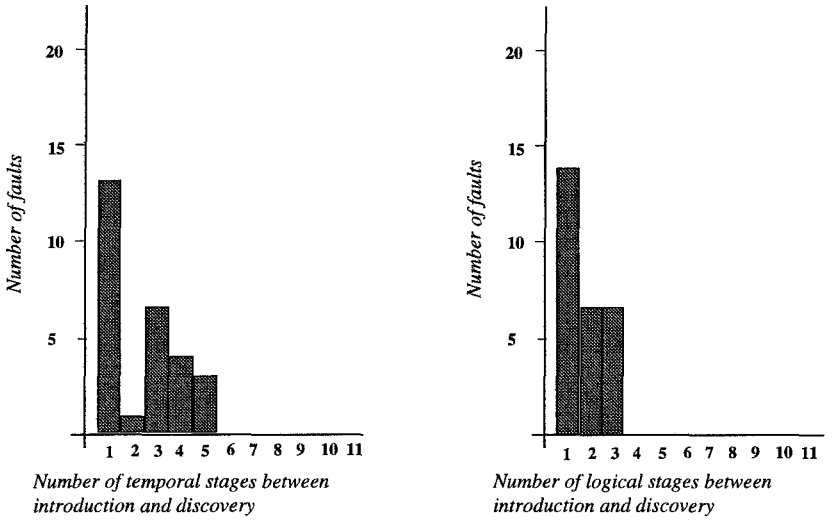

Fig. 7. Bar charts for MaFMeth

With hindsight, we realise that, although we had made a point of trying to keep this step as simple as possible, we were in fact simultaneously undertaking a change of data model, a change from implicit to explicit operation definitions, and the introduction of more structure into the specification [1]. It is clear that, in a tool supported development, the unsupported stages are likely to be the most error prone and it is imperative therefore to minimise the complexity of the unsupported stages.

It is sobering to note that all stages where development took place introduced faults!

7 Effort by stages

The distribution of effort by project stage is shown in Figure 8. As might have been expected, the bulk of the design effort was in the main development in B. A substantial component was also expended on the early specifications in VDM. Very little effort was required during the testing stage.

Some activities, for example the initial B specification and its animation, are grouped together as they were carried out simultaneously and no separate effort figures were kept.

The faults found can be plotted against these efforts as a histogram with the width of columns representing the relative effort expended in each stage. However, when inspecting this it must be remembered that some stages involved development whereas others purely involved review.

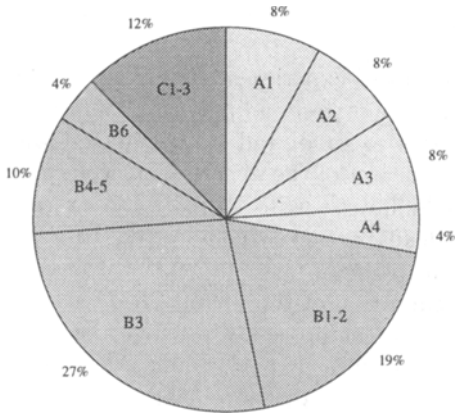

Fig. 8. Pie chart of effort by project stages

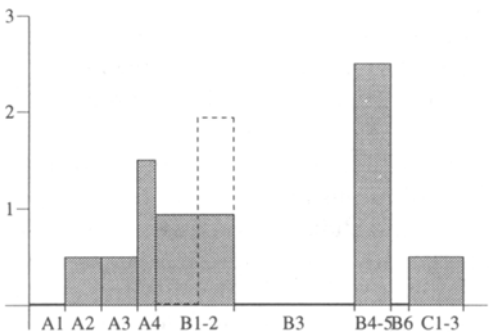

Fig. 9. Faults found per day by project stages

For stages B1-2, one cannot assess how much effort was expended in finding faults through animation and how much on development, but if one assumes that approximately one half of this effort was spent on each activity, then the dotted line applies.

Note how the most efficient fault finding occurs during test generation, animation and proof. Although this can perhaps be attributed to the fact that most faults were found before testing occurred, the test generation and proof stages allow a different perspective on the specification and highlight problems which might otherwise be invisible to the developer.

8 Conclusions

Conclusions drawn from this experiment should be moderated by the small size of the development and the correspondingly small number of faults detected. The development team was also small and staffed by self-selected individuals who, being keen to make a success of the experiment, were perhaps better motivated than average. It would not be wise therefore to extrapolate these results to larger projects. The lead partner will, however, be repeating the experiment with another development team.

An underlying assumption of the "Fault Grid" style of presentation is that activities in the development process are carried out in a linear fashion; it is difficult to present the results of performing activities in parallel, where the same faults may be discovered by more than one activity. Another feature of the "Fault Grid" presentation is that there is no record of faults found and introduced at the same stage. This is not considered to be a serious drawback, since, by their very nature, the cost of correcting such errors is low.

No attempt was made to moderate the effectiveness of fault finding by the severity of the faults found. Such an analysis could contribute to an estimate of the cost-effectiveness of each activity. None of the diagrams emphasise the fact that early fault detection saves money. It might be possible to estimate how much effort a process has saved in relation to how soon faults are discovered by keeping a record of how much effort is required to repair each faults found, and estimating how much effort would have been required to fix that fault if it had not been discovered until the last stage. The appropriate facts were not recorded in this project.

Unfortunately, due to some large-scale restructuring in the lead organisation, the code developed in this project never reached the production stage, and so no data is available on validation test and customer use. In particular, it remains unknown whether any types of error, perhaps peculiar to the use of formal methods, remained undetected by the development process.

Despite these qualifications, there is evidence in these results in favour of formal methods. Faults are inevitable and there detection is aided by formalisation. Amongst other things it is noted that all early stages, whether testing or development, found faults. It seems that any analysis , whether animation, PO generation, proof, or testing, is worthwhile. These activities are only possible once the objects involved are formalised.

This project has contributed to the beginnings of an accumulation of evidence for the benefits of formal methods. It has raised awareness of the need to gather such evidence for larger projects and suggested some techniques for doing so.

References

1. J.C. Bicarregui, J. Dick and E. Woods, Supporting the length of formal development: from diagrams to VDM to B to C Proceedings, 7th International Conference on: Putting into practice method and tools for information system design, Nantes (France), October '95, IUT de Nantes, H. Habrias (Editor) 1995.

2. J.C. Bicarregui and B. Ritchie. Invariants, frames and postconditions: a comparison of the VDM and B notations. In *Proceedings of Formal Methods Europe '93*, Lecture Notes in Computer Science, Springer-Verlag, 1993.
3. Jeremy Dick and Jerome Loubersac. A Visual Approach to VDM: Entity-Structure Diagrams. Technical Report DE/DRPA/91001, Bull, 68, Route de Versailles, 78430 Louveciennes (France), January 1991.
4. Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284, Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.
5. J.Dick, Fault grids: another way of presenting fault counts, Software Reliability and Metrics Club Newsletter, Issue 16, July 1995, p 2-4. (published by the Centre for Software Reliability, University of Newcastle upon Tyne).
6. Des Maisey and Jeremy Dick, Measuring the quality of the development life cycle process, Submitted to SQM96, Software Quality Measurement.
7. U.K. Department of Trade and Industry, TickIT: Guide to Software Quality Management, System Construction and Certification using ISO9001/EN29001/BS5750 Part 1, February 1992, TickIT Project Office, 68 Newman Street, London, W1A 4SE, UK.
8. M.C. Paulk, W. Curtis, M.B. Chrissis, C.V. Weber, Capability Maturity Model for Software, Version 1.1, Carnegie Mellon University Software Engineering Institute Technical Report, CME/SEI-93-TR-24, February 1993.
9. E. Woods, The Development of a Software Subsystem Using VDM and B, University of Manchester, Board for Continuing Education, Department of Computer Science, MSc Thesis, 1995.
10. J. Dick and E. Woods, Lessons Learned Applying Formal Methods to System Software Development, submitted (July 1995) to IEEE Software.
11. Yourdon Inc., The Yourdon Systems Method: Model Driven Systems Development, Prentice Hall, Englewood Cliffs, NJ, USA, 1993, ISBN 0-13-285818-5