

Making formal methods work: what is being done and what can be done

P. Lundy & D.W. Bustard

*Department of Computing Science, University of
Ulster, Cromore Road, Coleraine, BT52 1SA,
Northern Ireland, UK*

ABSTRACT

Despite the apparent advantages offered by formal methods, industry has been very slow to take on the techniques involved. This paper attributes some of this reticence to a lack of clear guidance on where and how formal methods fit into the software engineering process. The paper distinguishes between *formal methods* - techniques for modelling and reasoning about systems mathematically - and a *formal development process* - a defined process for specifying, constructing and maintaining software using formal techniques. The application of formal methods in a development process are discussed and examples of a few of the industrially-oriented processes currently in use are described. These are: (i) RAISE; (ii) Cleanroom; and (iii) the formal specification of classes in object-oriented software development. Conclusions are drawn on the approach that might be taken to introducing formal techniques as a routine aspect of software development.

INTRODUCTION

Among computer science academics, it is a commonly held belief that formal methods are an essential element of any adequate software development process. It is acknowledged that the adoption of the techniques involved is problematic but there is strong conviction that the benefits to be gained will eventually outweigh any disadvantages [1]. This transition point has apparently not yet been reached as formal techniques have so far had little effect on the general approach to software development in industry. Certainly there have been some reported examples of successful use across a broad range of applications, including hardware design [2], transaction processing software [3] and safety

486 Software Quality Management

critical systems [4]. However, these are isolated cases [5,6,7] and commercial practice remains rooted in traditional structured approaches. Thus, the available evidence suggests that formal methods are failing to realise their potential as a general aid to software development [8].

One problem is that the role of formal methods in the software development process is not well understood and so, inevitably, there is confusion about the possible benefits to be gained and how those benefits might be realised [9,10]. Indeed, many are even unclear about the meaning of the term ‘formal methods’ which is reflected in the variety of definitions that are offered. The purpose of this paper is to make progress towards removing such confusion by considering where and how formal methods can be used effectively in the construction and maintenance of quality software.

The paper has three main sections. The first addresses the question “What are formal methods?” in which a distinction is drawn between formal methods and the formal development process which incorporates such methods. The second section reviews some of the formal development processes currently in use. The final section suggests the steps that could be taken to introduce formal techniques as a general aid to software development.

The formal development processes considered are:

- RAISE [11], techniques based around an extended version of VDM [12];
- Cleanroom [13,14], devised and used by IBM; and
- the formal specification of classes in object-oriented software development [15].

WHAT ARE FORMAL METHODS?

The term ‘formal methods’ is of relatively recent origin. Its roots lie in ‘program proving’ [1], which is concerned with demonstrating that a program possesses properties stipulated in its specification. Some authors assume that formal methods and program proving are synonymous [16]. More commonly, however, formal methods are given the broader interpretation of covering any mathematical approach to the description of a system and the various forms of analysis thereby enabled. For example, Wing [17] describes formal methods as

“...mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop and verify systems in a systematic rather than an ad hoc manner”.

The word ‘formal’ is used in the sense that the approach has a sound mathematical basis. It does not mean that precise methods are defined [18]. Indeed, the use of the word ‘method’ in this context is misleading, in that there are no step by step procedures involved. Formal methods do not dictate or even

advise how a formal description should be constructed or analysed, they just provide basic tools that can be used in the software development process [18]. The word ‘technique’ is therefore perhaps more appropriate than ‘method’.

This paper is concerned with the actual *method* of software development, so to avoid confusion the term ‘formal development process’ will be used to refer to this method, meaning a defined process for specifying, constructing and maintaining software using formal techniques.

The essential component of any formal technique is a notation with a mathematically defined syntax and semantics. Examples include Z [19], VDM [12] and LOTOS [20]. Such notations can be used to construct a specification of properties of a system - a *formal model*. In essence there are two main ways to use formal models in software development, as summarised in Figure 1. In a *subsidiary support* role, a formal model helps to clarify requirements that are initially expressed informally and provides a baseline for software design and implementation. The modelling is shown at the initial development phase, as this is where the greatest benefit is to be obtained, but could be introduced later if desired. In a *central construction* role, a formal model is *refined* in stages towards an implementation. Each refinement or *reification* extends the preceding model by dropping down to some lower-level, less abstract description, which may bring in additional detail or *constraints* from the informal specification. Each lower level description is sometimes referred to as a *design*.

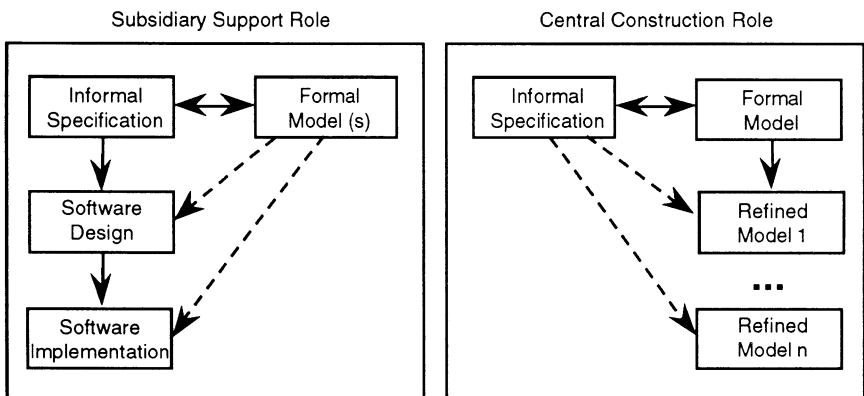


Figure 1: Possible Uses of Formal Models

Models describe a system concisely and unambiguously. Through constructing a model the specifier gains a greater understanding of the system being modelled than would otherwise be the case. There are further advantages. A formal model can be validated using mathematical reasoning and designs can be proved consistent with the higher level models from which they were derived. These activities of model construction, analysis and design verification

488 Software Quality Management

collectively constitute ‘formal methods’. Such methods or techniques can be used in whatever combination the developer finds beneficial. This may be as little as constructing an initial formal model without subsequent analysis, refinement or verification, right through to a full formal refinement cycle yielding executable code, proven consistent with the initial formal model.

Whichever approach is taken it needs to be integrated with the software development process in use. Consider, for example, the classical waterfall-type model of the software life cycle as illustrated in Figure 2 [21]. In such a model the development process moves through a series of *phases* from the initiation of a project, the determination of requirements and software design, to the implementation, operation, maintenance and final withdrawal of the software product. Each phase develops *phase products*. A phase is considered complete when those products have been evaluated and accepted. After acceptance, the phase products serve as the *baseline* for the next phase of development.

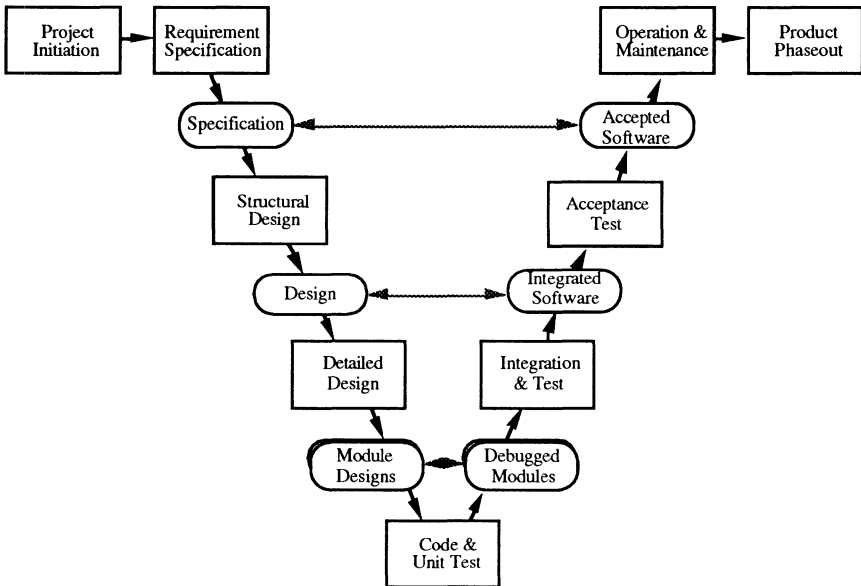


Figure 2: Software Development Life Cycle

Requirements specification is the first phase where formal methods are relevant. Here a formal model, or models, of the system under consideration can be developed to help clarify requirements and specify them precisely. Many assert that this work is the most important contribution of formal methods to the development process. It is argued that the production of a formal model provides such an insight into the nature of the problem that even if subsequent development proceeds informally the benefit will be felt throughout the process [22].

In the structural design phase, emphasis switches from a user's view of the system to that of the developer. The concern here is to partition the system into software components which when integrated will achieve the system's purpose. Formal modelling could begin at this level and be used to specify the function of each software component. Where formal modelling has been used at the requirements specification stage there are, in principle, two options available:

1. refine the top level functional model through a succession of formal transformations down to the structural design level, performing whatever proofs of equivalence are necessary on the way; or
2. create a new model at the structural design level and prove it equivalent to the higher level functional model.

In practice the approach taken will depend on the formal notation in use, the machine support available for transformation and proof and the amount of restructuring required. The refinement approach is preferable in that it can be very difficult to prove two separately created models equivalent. However, the refinement techniques supported for any particular notation may not facilitate the type of restructuring required. A third variation is to omit the full formal proofs of equivalence in the second approach. Instead, informal arguments of equivalence would be offered and examined in much the same way as requirements and designs expressed informally are compared at present. This would reduce confidence that requirements had been carried correctly into design but allow different modelling techniques to be used at each development level.

Moving from structural design to detailed design, and from detailed design to implementation, is really much the same process, and with the same options identified for the transition from requirements specification to structural design. However, major restructuring of models is less likely and so the process is more amenable to systematic refinement. This aspect of formal development has been relatively well explored [12].

Using formal methods to establish the validity of the specification and to develop an implementation through verified design reduces the need for testing[10]. It might even be argued that testing the functional aspects of the system becomes unnecessary. In practice, however, testing is always required [23] and a full test procedure should be followed. As far as possible, the formal models should be used for the generation of test data [24]. Overall, there is the expectation that a formal approach means that during testing fewer errors will be found and that the completed system will more closely match the client's expectations.

The use of formal methods during the operation and maintenance phase of the life cycle has been a largely neglected area. In principle, there is a need to be able to specify a change and ripple that change through all derived models. This could require a substantial effort in proving equivalence at each stage and

490 Software Quality Management

automated support is essential. Some initial work on supporting change to formal models has been undertaken for the formal notation LOTOS [25].

The waterfall model gives a general framework for software development. Usually, in addition, however, the developer will work under the constraints of some specific *structured method* giving more precise guidance on the processes and products associated with each life cycle phase. Examples include Structured Analysis/Structured Design [26], SSADM [27], and Object Oriented Analysis [28]. Ideally, formal techniques should be integrated directly with these structured methods. The integration of Structured Analysis with various formal techniques has been considered [29] but most of the research is at a relatively early stage. Nevertheless the approach does seem viable as long as adequate tool support can be provided. The next section considers three processes incorporating formal techniques to illustrate what is currently being achieved and to help identify where such techniques can be of benefit.

EXAMPLES OF FORMAL DEVELOPMENT PROCESSES

Raise

RAISE is an example of the use of formal modelling in a central construction role. It promotes the classical approach to formal development in which a model developed at the requirements specification stage is transformed through successive refinements into an implementation.

The RAISE (Rigorous Approach to Industrial Software Engineering) project was carried out under the EC ESPRIT programme from 1985 to 1990 by a group of European industrial partners led by Dansk Datamatik Centre (DDC), which was subsequently taken over by Computer Resources International (CRI) in 1988. In 1983, as a precursor to the RAISE project, DDC/CRI and STC Technology Ltd (STL) undertook a 'Formal Methods Appraisal' study [30]. Its conclusion was that the model oriented approaches to formal description, and in particular VDM [12], had proved the most viable for industrial use but that VDM had deficiencies, particularly in its structuring mechanisms and in its support for the modelling of concurrency. This led to the development of the RAISE Specification Language, based on VDM, extended with features to overcome its deficiencies.

The outcomes of the RAISE project, summarised in the RAISE Overview [31], were (i) the RAISE Specification Language (RSL) [32]; (ii) techniques and strategies for formal development and proofs (the RAISE Method) [11]; and (iii) a range of products and services including training courses, documentation and software tools [33].

RSL is a unifying framework in which the basic features of VDM are extended with facilities for property oriented specification, structuring and

concurrency. RSL includes facilities for specifying in both applicative and imperative styles (with or without variables), abstractly, as in algebraic specification languages, or model based, as in VDM or Z. Concurrent behaviour may be modelled via a process concept with synchronous channel communication based on the CCS[34] and CSP[35] approaches. The RSL notation is designed to support modular development. RSL modules can build on the definition of other modules and modules may be generic allowing them to be instantiated with actual parameters.

RSL has addressed many of the perceived limitations of earlier formal notations, making it flexible and powerful. The only weaknesses reported in initial case study reports [36] are lack of support for (i) the specification of temporal constraints; (ii) the specification of asynchronous behaviour; and (iii) the full representation of object oriented systems. This range of facilities in RSL has, however, been achieved at the expense of an increase in the level of effort required to learn the language. RSL is not easy for newcomers as they first need to become familiar with the various underlying modelling concepts involved. Indeed, there is evidence that the learning process is difficult even for those experienced in VDM. A programme of training courses in RSL, the RAISE Method and Justification and Proof, is provided in the UK by Lloyds Register. Video training courses are also available [33]. Experience of actual learning time, including attendance at courses is estimated to be between five and ten weeks [36].

Tools supporting the use of RAISE [26] include a syntax directed editor, syntax checker, pretty printer, RSL to ADA translator, module library management tool and a justification and proof support tool. The need for other tools has been identified [36] including, in particular, a graphical tool to support the explanation of models to clients.

The RAISE Method is based on a development life cycle of six stages: Requirements, Specification, Design, Translation, Maintenance and Extension. Each stage may have up to five phases: Capture, Formulate, Verify, Validate and Commit. All five phases are relevant to Design, in which the specification is refined in successive steps to a level of detail appropriate for translation to a target language, while verifying each refinement step. The Commit phase, in general, is a decision point where the results of the development are accepted or rejected on the basis of the verification and/or validation results.

The Method Manual [11] includes a library of standard sample specifications that can be used directly in particular applications. It also includes guidance on quality assurance and concludes with a case study of an inventory control system for a warehouse to illustrate the application of the RAISE Method.

The initial RAISE project has now been superseded by the ESPRIT funded LaCoS project [37], with objective to establish and demonstrate formal methods

492 Software Quality Management

as a viable industrial technique in the production of Large Correct (Information Technology) Systems. The LaCoS project involves the use of RAISE on a range of industrial applications in the areas of distributed system, safety critical software, transaction systems and real-time monitoring systems. The projects are being undertaken by companies in the UK, France, Spain, Italy and Greece. Reports on the experiences gained from these experiments are described in [36,38].

RAISE has clearly made significant technical advances towards producing a method which will cater for a wide range of applications. Persisting difficulties include the fundamental problem of assisting specifiers to produce an initial specification; the need to define a better managed transition path into the use of the notation; and the problem of relating the formal method to established development methods. Experience in the use of RAISE is increasing but the user base is still relatively small [39].

Clean/Z

IBM's Cleanroom Engineering is a software development process [40] that uses formal techniques to support a rigorous approach to software construction. The process is based on Cleanroom hardware development techniques and aims to produce error-free software by verification against a specification before execution is attempted. Software is developed in *increments* of between 5000 and 15000 lines of third generation code.

There are three teams involved in the process:

1. a specification team, who prepare and maintain specifications of the complete system and its specialisations for each increment;
2. a development team, who design and implement the software; and
3. a certification team who compile, test and certify the software's correctness using statistical quality control methods.

The process involves four stages:

1. *Specification*: the production of a specification by the specification team. This must be approved before the next stage begins. The specification is in three parts: i) a precise but 'informal' external specification which defines the software from the user's perspective; ii) a formal internal specification of the software's function, expressed in terms of stimuli and responses; and iii) the expected usage profile which is used to guide the preparation of certification tests.
2. *Construction Plan*: the determination of the development and certification sequence by the specification team. This involves decomposing the specification into increments and specifying each increment. An increment is executable so no special testing routines are needed; also incremental integration can occur as each new increment is completed.

3. *Design and Build*: a series of increments are designed and built by the development team. The design is developed top down, in three levels, with verification performed at each stage. The implementation is then performed by rigorous stepwise refinement of the lowest level design into executable code. Finally the code is assessed with respect to its specification using verification arguments.
4. *Certification*: in parallel with the development team, the certification team prepare test cases that exercise the increments thus far developed. When an increment is ready it is tested and its quality assessed by the Mean Time To Failure based on the usage profile. Following testing the increment is either certified as satisfactory or returned for redevelopment.

The reported results from this process, in terms of residual errors found after release, are very impressive [40].

Recently, IBM have experimented with the use of Z [41] for software specification in Cleanroom, instead of their own notation. (This represents a continuation of earlier collaborative work between IBM Hursley and Oxford University's PRG in which retrospective specifications in Z were used in the restructuring and development of the CICS transaction processing system [3,42].)

Formal Object-Oriented Development

Object-oriented software development, involving object-oriented analysis, design and implementation [43], is rapidly becoming the most popular approach to software development. An *object* represents an “individual, identifiable item, unit or entity, either real or abstract, with a well-defined role in the problem domain” [44]. For example, a data structure such as a stack or file is an object. An object has associated *data attributes* that describe its *state*, and defined *operations* that are used to inspect or modify that state. For example, one attribute of a stack would be a position marker identifying its top data item; an operation ‘pop’ might then have the effect of removing that data item and adjusting ‘top’ accordingly.

Techniques for specifying objects formally are well established, mostly based on the notion of an abstract data type [45]. In general, an abstract data type introduces functions and defines properties of those functions. The properties may be defined algebraically using equations that specify the inter-dependent effect of the functions. The functions may also be defined using an axiomatic approach [1] by specifying i) *preconditions* that hold prior to the execution of each function; ii) *postconditions*: that hold on completion of the function; and iii) *invariants* that hold across all functions.

Formality is used in object-oriented development to ensure that the function of each object is defined precisely before it is implemented.

494 Software Quality Management

Myer, in designing the object-oriented language Eiffel [15], has attempted to provide direct support for the specification and refinement of object classes. This has been done by enabling class specifications to be produced separately from implementations. An implementation is then based on such a specification class by inheriting its definitions. The meaning of class operations can be expressed axiomatically by enabling assertions to be placed in specification classes to define preconditions, postconditions and invariants for the operations. Such assertions help to guide the implementation and are translated into run-time code to assist testing. Implementation code can also be enriched with assertions to provide additional check points. The run-time cost of these checks is rather high, however, and so the checking code is usually suppressed in production versions of the software.

Object specification occurs at the design stage of software production, which is rather late in the process to gain full benefit from formal modelling. On the credit side, however, the techniques involved are relatively easy to apply and can provide a good return for a modest effort.

ASSESSMENT

The preceding sections have attempted to clarify what is meant by 'formal methods', emphasised the importance of integrating formal techniques within a general software development framework, and given brief descriptions of three of the best publicised examples of formal development processes where this integration has been attempted. Four main conclusions can be drawn from this discussion:

1. Formal methods are alive and kicking! Despite the slow uptake by industry, interest in the practical use of formal techniques persists.
2. There is no obviously 'best' way to apply formal methods. The term covers a range of techniques that can be used in various combinations. Specific uses, such as RAISE and Cleanroom have been developed, but no such approach has yet received widespread acceptance.
3. Formal methods need to be aligned more closely with the software development process. Current techniques are mostly directed towards the building of software from scratch, in a single development pass. In the formal methods literature, very little reference is made to maintenance or indeed to the need to make adjustments to formal models during development.
4. Formal methods need to be used in conjunction with existing software development methodologies. It is impractical to expect formal techniques to replace existing ways of working and more effort needs to be put into developing integrated approaches that can ease the introduction of formal techniques.

Given the uncertainty over how best to use formal techniques many commercial enterprises are probably tempted to sit back and wait for the water to

clear! However, as formal methods offer a family of techniques it is possible to consider introducing formal techniques incrementally and so gain some benefit immediately. The following recommendations are offered:

Recommendation 1. Establish a quality ethos

Formal techniques can only be used effectively in an environment where quality is a general concern. Cleanroom illustrates this total quality management commitment to getting things right first time, every time [46].

Recommendation 2. Establish a quality process

Formal techniques will assist the production of quality software but cannot carry full responsibility for the adequacy of the final product. Thus, formal techniques are best used within a well defined software development process, preferably one capable of obtaining ISO 9001 or TickIT Certification [47].

Recommendation 3. Increase awareness of formal techniques

An awareness of the reasons for formal modelling and of the techniques involved encourages software engineers to appreciate the benefits of being precise and the dangers of ambiguity. This in turn means that even if formal modelling is not an explicit part of the development process software engineers will tend to be more critical of system descriptions they are given and more careful when producing descriptions themselves.

Recommendation 4. Experiment and measure the cost-benefit of using formal techniques

As formal techniques are introduced, measure the costs and the benefits to ensure that the new techniques are of adequate value. Benefits may be measured in terms of increased reliability of the software, reduction in overall cost of development, clearer assessment of requirements, and so on. The benefits may be obtained in the short term during initial development of the software, or in the long term during maintenance. Against such benefits must be balanced the cost of introducing and using the formal techniques, again taking the assessment over both the short and long term. The cost-benefit analysis must necessarily be influenced by the type of application involved. In the case of safety critical systems, for instance, it may be that the benefits of reduced likelihood of error would outweigh almost any reasonable increase in development costs incurred by the use of formal techniques.

Despite many broad claims made about formal techniques, there is a lack of hard evidence that formal modelling is beneficial [8] and indeed there is little evidence of any research aimed at evaluating cost-benefits precisely [23,48,5]. There is even scepticism [8] about the very few quantified measures that have emerged, such as the use of Z in the enhancement of CICS [3], which is claimed to have reduced costs by 9% and reduced the level of faults in the final system by about 30% . A key issue is the absence of measurement techniques to support such conclusions and the view is expressed by Fenton [8] that until there is a

496 Software Quality Management

more widespread demand and expectation for objective measurement-based evaluation, software organisations and standards bodies will continue to place their trust in the unsubstantiated claims and biases of suppliers - both academic and commercial.

Recommendation 5: Develop formal model reading skills first

The task of constructing a formal model is far from trivial and, despite views to the contrary [49], it takes considerable time to develop both the required mathematical skills and a facility with the formal notation involved [9]. Indeed, it is probably unrealistic to expect all software engineers to achieve such competence. However, it is still beneficial for software engineers to be able to understand models produced by model builders as these would serve to more precisely specify their work. This might happen, for example, when object classes are implemented from object specifications or when software is designed taking account of user requirements expressed formally.

Recommendation 6. Don't try to model everything

Software is the ultimate formal model of a system so models produced at earlier stages in the life cycle must necessarily be more abstract to serve their purpose. These models should focus on key aspects of a system rather than aim for completeness. Completeness is not usually attainable anyway because the non-functional aspects of a system, such as performance characteristics, typically cannot be described using the available formal modelling notations.

Recommendation 7. Start introducing formal techniques at the beginning of the life cycle

The earlier formal techniques are used in the life cycle the greater will be the benefit gained. Start with 'throwaway' specification models and gradually try to gain greater benefit from such models by, for example, deriving tests from them or giving them to designers. In effect, this means gradually integrating the techniques with the existing development process.

Recommendation 8. Introduce formal modelling through experts in the field

One very simple way to assess the value of formal techniques (though not necessarily the cheapest approach!) is to employ an expert to do the formal modelling and evaluate the recommendations made. The role of the formal modelling expert in this case is similar to that of a quantity surveyor or heating engineer in relation to an architect, namely as a provider of a specialised service to give an evaluation of, and contribute to, some aspect of a design. If the work of the consultant proves worthwhile then an organisation could go on to build up their own formal modelling specialists to provide the same service internally.

CONCLUSION

Formal techniques promise to be a significant aid to software development but have so far had minimal influence over the way that software is produced commercially. This paper has suggested, in effect, that this phenomenon is due largely to a lack of adequate communication between formal technique suppliers and potential users: users don't fully understand what it means to introduce formal techniques and are unaware of the variety of approaches that are possible; suppliers, likewise, don't understand the precise needs of users and so tend to ignore basic software engineering issues such as change control and integration with existing methodologies. The barriers are slowly coming down as the problems involved are addressed. The process can be accelerated if both sides keep an open mind and take positive steps to resolve differences. This means users actively attempting to introduce formal techniques using the approach suggested in this paper and suppliers trying to support developers more directly. Joint experiments involving the quantitative critical evaluation of the results would also be beneficial.

ACKNOWLEDGEMENTS

The work in this paper has been undertaken as part of the RUNES project (Requirements Understanding, Negotiation, Expression and Scrutiny), funded by the University of Ulster. RUNES is broadly concerned with investigating techniques for improving the requirements definition process and the development of tools to support the process.

REFERENCES

1. Hoare CAR, *An axiomatic Basic for Computer Programming*, Comm. ACM, **12** (10), pp. 576-80, October 1969
2. Cullyer WJ and Pygott CH, *Application Of Formal Methods To The VIPER Microprocessor*, IEE Proc. E., **134** (3), pp133-141, 1987
3. Hayes I, *Applying Formal Methods To Software Development In Industry* IEEE Trans. Soft. Eng., **11** (2), pp169-178, February 1985
4. Bowen JP and Stavridou V, *Safety-critical Systems, Formal Methods And Standards* Software Engineering Journal, **8** (4), pp. 189-209, July 1993
5. Bjorner D & Druffel L, *Industrial Experience with Formal Methods*, Proc 12 ICSE Workshop on Industrial Use of Formal Methods, Nice, March 1990
6. Tierney M, *Potential Difficulties in Managing Safety-Critical Computing Projects: A Sociological View*, in Directions in Safety-Critical Systems, pp 43-64, Springer Verlag, 1993
7. Barroca L.M. and McDermid J.A. *Formal Methods: Use And Relevance For The Development Of Safety-Critical Systems* Comp. J., **35** (6), pp. 579-599, 1992
8. Fenton N, *How Effective are Software Engineering Methods?*, J Systems Soft, **20**, pp. 93-100, 1993



498 Software Quality Management

9. Cooke J, *Formal Methods - Mathematics, Theory, Recipes or What?*, Comp J, **35** (5), pp419-423, 1992
10. Plat N, Van Katwijk J and Toetenel H, *Applications and Benefits of formal Methods in Software Development*, Soft Eng J, **7** (5), pp. 335-346, 1992
11. Brock S and George CW, *RAISE Method Manual*, Technical Report LACOS/CRI/DOC/3/V1, CRI A/S, 1990
12. Jones C, *Systematic Software Development Using VDM*, Prentice Hall, London, 1986
13. Mills HD, Dyer M and Linger RC, *Cleanroom Software Engineering*, IEEE Software, **4** (5), pp. 19-25, 1987
14. Dyer M, *The Cleanroom Approach To Quality Software Development*, Wiley Series In Software Engineering Practice, 1993
15. Myer B, *Object-oriented Software Construction*, Prentice Hall International, 1988
16. Flynn DJ, *Information Systems Requirements: Determination and Analysis*, McGraw Hill, 1992
17. Wing JM, *A Specifiers Introduction to Formal Methods*, Computer, **23** (9), pp. 8-24, 1990
18. Woodcock J and Loomes M, *Software Engineering Mathematics*, Pitman, London, 1988
19. Spivey JM, *The Z Notation: A Reference Manual*, Prentice Hall Int., Englewood Cliffs, New Jersey, 1989
20. ISO - *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 8807, 1989.
21. Rook P, *Controlling Software Projects*, Software Engineering Journal, pp. 7-16, **1** (1), Jan 1986
22. Thomas M, *The Industrial Use Of Formal Methods*, Microprocessors and Microsystems, **17** (1), pp. 31-36, 1993
23. Brinskma E, *What is the Method in Formal Methods*, Proceedings of the 4th Int. Conf. on Formal Description Techniques, pp. 31-51, Sydney, Australia, 1991
24. Hayes IJ, *Specification Directed Module Testing*, IEEE Trans.Soft.Eng., **12** (1), pp. 124-133, 1986
25. Bustard DW & Winstanley AC: *Making Changes to Formal Specifications: Requirements and an Example*, Proceedings of ESEC '93, pp. 115-126, Springer-Verlag, Garmisch, Germany, September 1993
26. Yourdon E, *Modern Structured Analysis*, Prentice Hall, Englewood Cliffs, New Jersey, 1989
27. *SSADM Version 4 Reference Manual*, NCC/Blackwell, Oxford, 1990
28. Coad P and Yourdon E, *Object Oriented Analysis*, 2nd Ed. Yourdon Press/Prentice Hall Int., 1991
29. Semmens LT, France RB and Docker TWG, *Integrated Structured Analysis and Formal Specification Techniques* Computer Journal, **35** (6), pp. 600-610, 1992

30. Prehn S and Hansen IO *Formal Methods Appraisal*, Technical Report FMA/DDC/SP, Dansk Datamatik Center, 1983
31. Haxthausen A, Storbak Pedersen J and Prehn S, *RAISE Overview*, CRI A/S, 1992
32. *The RAISE Specification Language*, BCS Practitioner Series, Prentice Hall International, 1992
33. Bruun PM, Dandanell B, Gortz J, Haastrup P, Haff P, Prehn S, Snog H, and Zierau E, *RAISE Tools User Guide*, Release 1.3.0, LACOS Report DOC/4, Computer Resources International A/S, Denmark, May 1992
34. Milner R, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, 9, Springer-Verlag, 1980
35. Hoare CAR, *Communicating Sequential Processes*, Prentice Hall International, 1985
36. Chalmers D, Dandanell B, Gortz J, Storbak Pedersen J, and Zierau E, *Experiences From Applications of RAISE , Report 1*, LACOS/CRI/CONS/13/V1, June 1991
37. Dandanell B and George C, *The LACOS Project* LACOS/CRI/BDH/8/V4, September 1992
38. Dandanell B, Gortz J, Storbak Pedersen J, and Zierau E, *Experiences From Applications of RAISE, Report 2* , LACOS Report CONS/20, CAP GEMINI A/S, March 1992
39. Austin S and Parkin GI, *Formal Methods: A Survey* National Physical Laboratory, UK, March 1993
40. Cobb RH and Mills HD, *Engineering Software Under Statistical Quality Control*, IEEE Software, **7** (6), pp 44-54, November 1990
41. Normington G. *Cleanroom And Z* in Bowen J.P. and Nichols J.E. (Eds.) *Z User Workshop*, London, 1992 (Springer-Verlag 1993, pp. 281-293)
42. Collins BP and Nix CJ, *The Use of Software Engineering, Including the Z Notation, in the Development of CICS*, Qual Assur, **14** (3), pp. 103-110, 1988
43. Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorensen W, *Object-Oriented Modeling and Design*, Prentice-Hall International, 1991
44. Booch G, *Object Oriented Design with Applications*, Benjamin Cummings, 1991
45. Guttag JV, *Abstract Data Types and the Development of Data Structures*, Comm. ACM, **20** (6), pp. 396-404, June 1977
46. Oakland, *Total Quality Management*, Heinemann, 1989
47. DTI, *TickIT Guide to Software Quality Management System Construction and Certification*, 2.0, February, 1992
48. Glass RL, *Of Understanding, Acceptance,... and Formal Methods*. J. Systems Software, **21** pp115-116, 1993
49. Hall A. *Seven Myths Of Formal Methods* IEEE Software , **7** (5), pp11-20, 1990