

13 Formal Methods

The value of using formal methods¹ in the development of critical systems has been hotly debated for a number of years.

On the one hand a specification expressed in one of these notations can be a very precise description of behaviour and this description can, at least in principal, be shown to be both self-consistent (or at least not internally inconsistent) and to exclude particular undesired behaviour, such as behaviour which could lead to an accident. There is also the prospect of totally formal development which would create an executable software system which is guaranteed to be consistent with the original formal specification.

On the other hand, the fact that a specification is precise does not mean that it is necessarily right. A large proportion of problems in critical systems derive from the fact that the specification did not reflect the real requirement. Formal notations are unfamiliar to system users and to professionals from other disciplines so the problems of ambiguous natural-language specifications may simply be replaced by difficulties in communicating the meaning of a formal specification². In addition, formal specification, and particularly formal verification or proof, is expensive and uses scarce staff resources - it could be that this money and effort could be more effectively deployed using conventional methods more thoroughly.

In view of these uncertainties and the relatively immature state of practice in this field, two projects explored:

- the several roles which formal methods could play in system development and procurement;
- three trial applications of a particular formal method (RAISE).

The relationship between formal specification and implementation was also explored by the SADLI project in connection with functional languages (see section 14.2). Section 14.4 reports on the part of the MORSE project which looked at using natural language processing tools as an aid to producing formal specifications. A somewhat different use of formal methods is in language specification and the proof of language transformations. The work of the BYLANDS project in this area is the subject of section 13.3. In a similar vein, the SSI Tools project (section 10.1) applied mathematical methods to the definition of the SSI Data Language and examined the possibilities for proving the safety of railway layouts. A further use of formal techniques, to explore the semantics underlying standards and guidelines, is described in *Communication in engineering design* in Chapter 9.

The DATUM project also investigated the role of formal methods in the design process. They did this using idea writing techniques to analyse the contributions and limitations of formal specification and proof in safety-critical design [DATUM ref 23]. They concluded that formal methods were considered to be useful in increasing the descriptive rigour and hence the probable accuracy of specifications; however, proof is rarely used effectively in formal specifications, hence no guarantee of reliable code can be offered. The main benefit seems to be in making

¹ ie methods employing mathematically based notations for specification or design.

² for an interesting approach to bridging this gap see section 14.4 on using natural language processing tools.

designers think harder about the problem because they have to describe it in a concise, formal language. Lack of effective tool support and the (in-)comprehensibility of formal notations were seen as the principle barriers to take up, and they found little empirical evidence that formal methods actually improve software reliability (for example see [DATUM ref 11]).

13.1 Formal Methods for cost-effective procurement of high integrity systems

It has long been recognised that software engineering lacks the formal, mathematical approach to specification and design that other engineering disciplines employ. Such formal methods (i.e. with a basis in mathematics) are becoming more important in software and Defence Standard 00-55 requires their use for the development of software which is deemed to be safety-critical. Thus it is becoming necessary, at least in some contexts, to adopt a formal development lifecycle.

The objectives of the SafeFM project were to develop guidelines for the procurement of high integrity systems using formal methods. The three project partners each focused on a different discipline of the procurement process. These disciplines were:

- Risk Engineering
- Formal Software Development
- Coherent Specification.

A Safety Framework

The framework into which these three disciplines fit is shown in Fig. 13.1. The diagram shows the important relationships between the various products of the three disciplines. The Risk Engineering discipline covers the initial assessment of system component integrity and the subsequent assessment of the development products. The Coherent Specification discipline is part

of the system analysis activities for the highest levels of integrity. The diagram shows how the products of the Software Development activities are linked to systems level activities and also to the associated risk analysis activities.

The process starts with an initial risk analysis, the aim of which is to determine integrity levels for different components of the system. The results of this analysis will ultimately decide the level of rigour of further systems analysis and software development and also the on-going risk analysis activities. The system is modelled from various viewpoints (functionality, con-currency, timing) using appropriate techniques, to form a fragmented system specification. Appropriate hazard analysis techniques are applied to the fragmented specifications to identify the criticality of the system's components.

System Analysis activities are guided by the results of the initial Risk Analysis. Where a high level of integrity is required a system component is re-modelled in a notation which allows many of the viewpoints modelled in the fragmented specifications of the initial risk analysis to be captured in a coherent, formal specification. Software and Safety requirements of the system are captured in the same formal notation, along with assumptions about the environment in which the component operates. The coherence and formality of these specifications mean that the system can be analysed by formal, mathematical proof. The Coherent Software specification used in the system analysis is also the first product of the Software Development process.

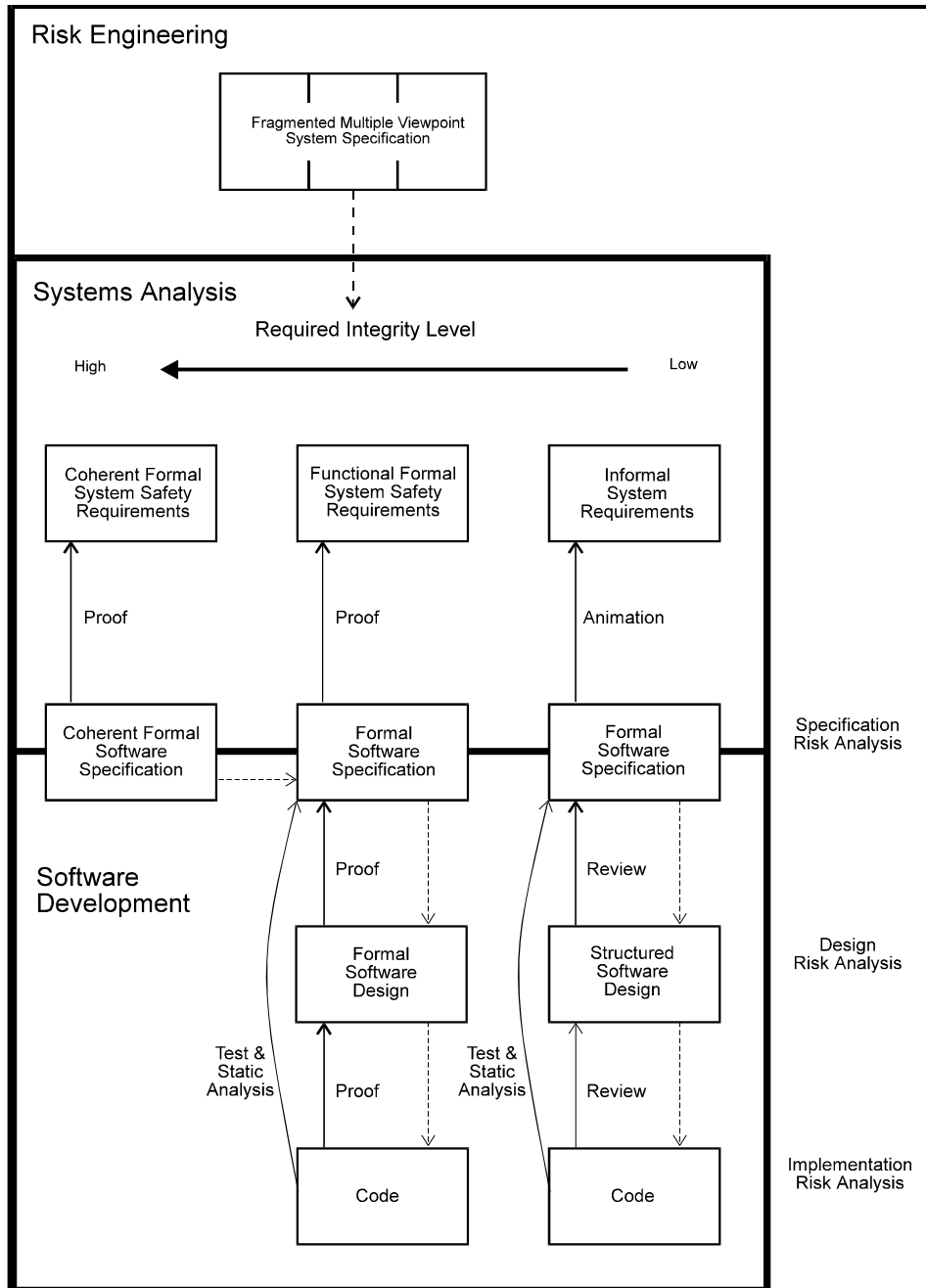


Figure 13.1. The SafeFM safety Framework

Where the components is of medium criticality, a less comprehensive analysis is performed. Only functional safety properties are analysed by formal proof. For lower levels of integrity, system requirements are verified by animating a formal software specification and testing against informally recorded system requirements.

The rigour of the Software Development process is also determined by the initial Risk Analysis work. For the higher levels of integrity a fully formal development is carried out. For lower levels of integrity, a formal specification is written and used as the starting point for a conventional structured development.

Throughout the development process, risk analysis activities are performed to assess the products of each stage.

Coherent Specification

In any large software development, requirements capture and specification is a crucial starting point to subsequent activities. Errors in the initial requirements can propagate to design and implementation phases and are known to be very expensive and difficult to correct. Requirements errors have been found to account for a large number of failures in software and have had implications in several accidents. Therefore, techniques to facilitate requirements specification and to support early validation are extremely important.

One main theme of SafeFM was to investigate the practical impact of formal methods on requirement analysis for high integrity systems. This work addressed some of the issues mentioned above and included the following activities:

- The definition of a practical method, supported by existing tools, for producing coherent system requirements. The method is targeted to real time applications and is designed to allow a thorough analysis of both functional and non-functional properties.
- The development of a concrete notation supporting the methodology proposed. This part of the work was based on the PVS theorem prover and consisted in the definition of various PVS functions, theories, and libraries for the expression and verification of real-time requirements.
- An evaluation of the method, notation, and tool on a realistic case study based on an existing avionics controller which presents many of the complex characteristics of safety-critical real-time systems. As the emphasis of SafeFM was on improving current practice rather than the theory it was essential to evaluate the method on an example from the real world.

The requirements were divided into three classes: functional requirements (usually related to software), assumptions about the environments, and non-functional requirements representing the properties one wishes to verify. The functional and non-functional requirements, as well as the assumptions on the system under control were all formally specified with PVS, and three different formal validations techniques were applied. The whole formalisation and verification was successfully completed. The amount of effort involved can be evaluated to around 18 man months.

The experimentation has shown that formal requirements analysis can be applied to a realistic system of substantial size and complexity. The tool support and the formalism were adequate.

The case study also confirmed all the potential interests of formal methods at an early stage in system development. Formal requirement specifications give a precise model of the future system whose behaviour can be thoroughly analysed. The proof process allows you to gain very deep and detailed understanding of complex systems and reveals even the more obscure and exceptional of behaviours. Currently no other analysis technique can give the same degree of confidence.

Another important lesson from the case study was the necessity of adopting a wide perspective when analysing requirements for control applications. Concentrating on software aspects is not sufficient to get an adequate understanding of the system. Requirements analysis for safety-critical controllers requires examination of the controller, the system under control, and the environment. Failing to realise the importance of elements external to the controller caused substantial delays and difficulties in the completion of the case study verifications.

Formal Development

The requirements of Defence Standard 00-55 call for the use of formal methods for the development of safety-critical software. The focus of the Formal Development work within the SafeFM project has been the practical use of formal methods to create a process suitable for development of software for systems which have been identified as having high levels of safety-criticality by the Risk Analysis activities.

This work has addressed three main areas of concern. These are:

- The integration of formal methods with the current development process,
- The link between system level specifications and the software development process,
- The integrity of the development process.

The ability to integrate formal methods with currently used development methods is an important factor in creating a practical process. Integration of methods provides a way of controlling the transition to a formal lifecycle. This makes it easier to access the benefits and to limit the scope of problems caused by the use of new methods.

Software Development with Formal Methods

The integration of formal and current methods was investigated within the SafeFM project by creating two different development processes and assessing the practicalities of the individual approaches.

The first approach combined formal specification in VDM with current structured methods and verification techniques to create a development process. The use of a formal specification in the combined formal and structured approach made animation possible, providing a much better understanding of and greater confidence in the specification. The preciseness of the formal specification and the use of animation to derive code test cases improved both the design and verification aspects of the development process. The use of a formal notation to improve the specification phase of a development was considered to be a sensible, initial use of formal methods.

The second approach made more extensive use of a formal method (B-Method and its tool support (B-Toolkit)) to replace not only the specification but also the design and code generation phases of the development process. Dynamic testing was employed for code verification. This formal process used proof to verify the transformation between top level specification and implementation specification.

The more formal development proved to be well supported by the tool and made the formal verification of refinement phases much easier than expected.

The overall conclusion reached was that there are benefits to using formal specification techniques to replace those of a more conventional structured development and this combined approach requires minimal extra training. However, the effectiveness of the tool support used for the formal development makes this approach much more practical than expected.

A detailed account of the project's work in this area is available in [SafeFM *ref* D12].

Formal Verification of Safety Requirements and Design

The project developed a method for proving safety requirements for high integrity systems. The method, described more fully in [SafeFM *ref* D13], uses a combination of formal specification and proof with the B-Method, and formal proof with the PVS.

The method analyses the system at an intermediate level, between the comprehensive PVS analysis investigated during the Coherent Specification work, and pure B-Method analysis which may be used for less critical systems. The translation is much smaller than that required for the full PVS analysis but some of the power of the PVS analysis is lost.

The combined method has been tested on a case study and has worked well. An example requirement that was difficult to describe in B was fairly easy to describe and prove in PVS. The transformation from B to PVS was limited to a few states and two invariants and could be verified by hand.

Process Integrity Assessment

The integrity of software depends to some degree on the integrity of the tools used to support the software development. In general it is thought that transformation tools, such as compilers, need to be of high integrity, while verification tools, such as type checkers can be of lower integrity. Def Stan 00-55 mandates that the safety integrity requirements for the tools shall be established by a hazard analysis and safety risk assessment implemented by the application of Def Stan 00-56.

Tools are essential to support the use of formal methods on realistic examples. However, at present there are no tools, or sets of integrated tools, that provide full support for every stage of the formal methods lifecycle. Therefore, any proposed lifecycle must contain compromises to allow for the incomplete tool support. In order to judge these compromises, the combination of tools and processes used by the development lifecycle need to be analysed.

A fault tree analysis was performed on an example development methodology to assess the required tool integrity and the critical processes. The first translation, between the safety proof tool and the refinement tool was identified as the most critical part of the development. Improvements to this process would have a significant effect on the overall software reliability. The other translation was found not to be critical.

The critical tool was found to be the compiler. This was expected as this is the only transformation (as opposed to verification) tool in the example development. The safety proof

tool was surprisingly found not to be critical. Most of the other tools had an average level of criticality.

The results of the analysis suggest that the two areas where improvements would be of most value are in improving the link between the safety and refinement proof tools and improving the quality or verification of the compiler.

13.2 Case studies with RAISE in avionics, plant control and laboratory information

The MORSE project carried out three case studies concerned with investigating the advantages and disadvantages of using the formal method RAISE and the RSL language in a commercial environment:

- The study by Ultra Electronics was concerned with the development of a formal specification of software to drive an aircraft landing gear control and interface unit (LGCIU).
- Transmitton Ltd investigated a system for the control of a gas storage and distribution site.
- The West Middlesex University Hospital carried out a study to assess the efficacy of formal methods in the clinical laboratory environment.

13.2.1 Avionics

Ultra Electronics' interest in the case study, in addition to the general project objectives, was that it offered an opportunity of using a formal method in a typical software development project and of learning what formal methods could offer to commercial software engineering. The advantages and disadvantages of the formal approach could be compared with those of the traditional software engineering approach. To enable this, the software project chosen as subject matter for the case study was one that had already been developed in a conventional way .

The desired outputs from the case study were some ideas as to how the formal method performed in such a development task, and how the personnel assigned to the work fared with it. A measure of how much time and money could be saved or lost by using formal methods was to be made, and an indication gained as to the general advisability of adopting formal methods in future software development work.

Over the course of the MORSE Project, it was hoped that lessons would be learned that would aid both in making the decision whether or not to adopt formal methods, and if the decision were made to adopt them, doing so in as profitable and non-disruptive manner as possible.

This work has been carried out in several stages

- informal expression of the software requirements in a natural language
- expression of those requirements as a formal specification, in RSL
- a study of the safety aspects of the software requirements, together with recommendations for additional requirements to increase safety

- the transformation of the additional safety requirements into RSL, and their amalgamation with the original RSL specification, to produce a refined requirement specification
- refining the requirement specification into a formal detailed design specification
- submission of the detailed design to a code generator in order to produce compilable software in C++
- the design, formal specification, translation and testing of a simulator to interact with the LGCIU software, in order to simulate the behaviour of a suitable set of aircraft systems
- interfacing two instances of the LGCIU design specification with the simulator specification, and submitting the whole to the C++ code generator
- dynamic testing of the LGCIU software in the environment of the simulator.

Advantages and disadvantages of the formal approach

The avionics case study identified a number of both advantages and disadvantages in using a formal approach.

An obvious advantage of formal specification of software, and of RSL particularly, was found to be that it offers a great aid to clearly visualising a software design, and the interfaces inherent in it. Additionally, it allows aspects of the software to be stated explicitly, without ambiguity, and can make many implicit aspects of the software explicit, thus enabling them to be analysed with greater ease than with some other methods.

The explicit statement of properties in axiomatic form is especially useful in capturing and analysing software requirements, where contradictions, conflicts, errors, and incompletenesses can be more easily trapped and eliminated at an early stage in development.

On the other hand, there are some serious problems to be overcome when making use of formal methods, and certain of these assume much greater proportions when doing so for the first time.

A major problem with the use of formal methods at present is the impenetrability of the formal languages that they use. The vocabulary of RSL is large, and its syntax and semantics complex. This is necessary in order to make it a wide-spectrum language, applicable to many different stages in the development process, but it also makes it difficult for someone untrained in its use to unambiguously comprehend the import of many of the statements that can be made in a formal RSL specification. Confusion over subtleties in the use of various kinds of RSL statements, and the ways in which they can affect other statements was a serious problem at the time when the draft version of the first-cut RSL specification was being produced, and led to syntactic and semantic errors in the RSL generally.

RSL is not an easy language to learn to use. There is a very long learning curve associated with it, whose shape is 'flat' in its early stages. Like any other language, natural or artificial, correct and clean use can only come about with constant practice and feedback.

It is felt that an increased amount of formal training in RSL would not have significantly improved the situation or sped up the learning process, particularly since any such course can only concentrate on the most superficial aspects of the language, without going to any useful depth into the RAISE Method itself. Similarly, no great help would have been gained by studying examples of others' work. Perhaps paradoxically, the best way to get a good feeling for

the underlying principles of the Method is its continual use in a case study such as this, with regular feedback on work done.

The complexity of RSL means that not only would all affected members of the organisation using it need to be trained and become experienced in its use, but also that its use in a project in which complex technical discussion of specifications with the customer was needed would entail manually translating formal statements into informal statements in a natural language such as English. One of the main advantages of formal methods - unambiguity - would thereby be lost. The alternative measure, training all those required to understand and discuss such specifications in the formal language also seems impractical in the short term.

The contents of a formal RSL specification can be expressed using a natural language “paraphrasing”, but with difficulty. It is not easy to describe the precise meaning of a portion of RSL fully without using a great deal of RSL-specific jargon, which of course has itself to be explained in detail. The size of this problem can be gauged by consideration of the description of the RSL language: a book of nearly 400 pages. It is possible that an automatic translator would be a feasible proposition, to translate from a formal language into one more comprehensible to the layman, but it is difficult to see what the target language would have to be like if it were to be both unambiguous and informal.

Another major problem was found to be the fact that RSL has no concept of time built into it. It makes the implicit assumption that any process takes exactly zero time to execute, no matter what it does. It is thus difficult to easily express the requirements of real-time software. RSL must recognise that the majority of operations are composed of divisible sub-operations, and therefore that the state of the system external to an operation can change while it is being executed. The problem of expressing temporal aspects of software attributes in RSL can, to some degree, be worked around by use of ‘tricks’, such as supplying functions with an additional parameter of type ‘Time’. However, extensive use of techniques such as this makes any specifications thus produced increasingly unwieldy and awkward to manipulate as they grow larger and are refined. If RAISE and RSL are to grow in popularity, a solution to these problems will have to be found, that defines a model for the dynamic behaviour of the systems that are specified.

It is difficult to convince non-users of the worth of expressing software attributes in a formal language. There is a great tendency to see it as just another programming language. Even fairly experienced users can at times be lulled into considering merely its syntactic aspects, without considering the semantics and what kind of things it is possible to express in RSL.

Conclusions of the avionics study

The outcome of this investigation has been to suggest to some members of the Ultra Electronics Controls Division Software Engineering Department that formal methods may offer valuable advantages in the development of software.

The particular advantages that are noted are those of improved confidence in the software development, the possibility of improved safety analysis, and the potential for elimination of a greater number of potential software problems from the development in its early stages. There is also the prospect over a period of time of reuse of software specifications in other, similar, software development projects.

Having said this, it is not definite that Ultra will adopt RAISE or RSL as the formal language of its choice. As has been pointed out earlier in this document, the RAISE Method and RSL language are still under development, and have at this stage some problems, particularly with regard to the development of real-time software. However, the majority of problems discovered in the course of this work are concerned with the Tool Set (with the C++ translator particularly) rather than with the method or language themselves. RAISE and RSL do have some advantages over a lot of Formal Methods, in that they support object orientation and thereby re-use, and RSL provides several specialised constructs which contribute to good design. Currently, the conditional plan in the Software Engineering Department is to introduce formal specification gradually into a new software project, addressing only parts of the development cycle to begin with, and increasing this number if and when user and customer apprehensiveness toward the new technique are diminished. Naturally this can only be done as suitable projects present themselves.

13.2.2 Plant control case study

Transmitton Ltd is engaged in the business of designing and manufacturing industrial data gathering, monitoring and control systems. These systems are mainly used to manage geographically distributed or “wide area” plant. This plant is usually part of the essential infrastructure of society involving the supply of resources such as water, gas, electricity, oil and railway transport.

In the design of the equipment that is used for managing these networks, a very large proportion of the operating characteristics are determined by the installed software. This software comprises both the relatively small embedded real time programs used at a plant interface level and the rather larger conventional software employed in the central supervisory computer systems.

Transmitton chose a representative sample of the real time/embedded software requirements for a single gas storage and distribution site (part of a wide area network of gas supply in Portugal) as their case study.

At the start of the project, Transmitton’s engineers had little or no knowledge of Formal Methods and the associated languages. Their initial approach was therefore one of exploration of the possibilities with the general expectation that it should be possible to formally specify part or all of the PES software. By employing this formal approach they hoped to reap the benefit that the resulting code would be of superior quality (fewer errors of design) and therefore more ‘safe’ for use in critical applications. Recognising that carrying out work formally would be expensive, they intended to design their system in such a way that the various software elements might be re-used on different projects, thus making the initial effort cost-effective for a wider range of applications.

The experience gained during the project has allowed them to place Formal Methods into perspective as far as their own domain is concerned, comparing its costs and benefits with other approaches to achieving a better result. It should be noted that the conclusions reached may well not apply to other types of system.

For the case study, the engineers received training in the RAISE Method and have subsequently specified some of the requirements formally. From these statements it has been possible to refine

the design while retaining the properties of the original statements. Producing the RAISE specifications has been difficult, but ultimately straightforward to the point at which the RAISE syntax checker has accepted the statements.

Conclusions of the plant control case study

The first and overwhelmingly significant point which emerged from the study is that the syntax and semantics of the Formal Methods languages available today are generally impenetrable to the non-specialist. This obscurity is not dispelled by anything less than diligent attention to the detail of the languages and their methods. In the earlier part of the project, considerable efforts were made to become proficient in the application of the RAISE technology. It became apparent, however, that the lack of comprehension is widespread outside of a small core of largely academic proponents of these methods. This absence of understanding does not mean that the approach has no validity, but does lead to a number of severe difficulties in the practical evaluation of a Formal Method. RAISE, in particular, is a large and complex language; the problem might be less marked with other Formal Methods.

One difficulty arises when the original requirements are essentially ambiguous or not fully stated, as will be the case in the vast majority of projects. The normal method of correcting any design errors which may occur is then by the universal process of feedback. At various stages, the designer submits his work to the specifier for review, checking, correction and approval. Indeed this process is so universal as to be almost imperceptible. It is difficult to find a field of human endeavour in which the element of feedback is absent. How can the design process be returned to the specifier for review if a Formal Method is being used? Given the inaccessibility of the material, this is a difficult process since, at least in Transmitton's case the specifiers are invariably engineers of disciplines other than software, or even electronics.

This is particularly important because errors of system design caused by misunderstandings between the experts in the user's domain and experts in the designer's domain are a very significant element in the total tally of possible system errors. Furthermore, the kinds of errors produced by such misunderstandings are often of the most costly and potentially hazardous nature.

The Case Study team concluded that it is a key requirement for the effective use of Formal Methods that there should be no ambiguity in the initial requirements of the selected project. For this to be the case, the original specifier of the system must be aware that it is not permissible to leave unwritten those parts of the requirements specifications which might normally be regarded as "obvious". These omissions typically occur where reliance is placed on the domain expertise of the system designer or the organisation in which the designer is employed.

Systems where the majority of requirements can be rigorously expressed, however, tend to be either those where the functionality required is essentially simple or where the requirements are complex but with constrained interactions. An example of the former type might be a nuclear reactor shutdown system, while the latter is typified by the design of a computer operating system. Even in these circumstances, the extent of the contextual requirements are likely to outweigh the explicitly stated formal requirements and this diaspora of unwritten requirements can cause operability hazards that cannot easily be addressed by the formalism of the specification.

In summary, the Case Study concluded that it is a necessary condition for the use of Formal Specifications and Methods that both the designer and the user (specifier) of the system should be fully skilled in the language and method before the process is attempted. In those situations where such a condition is met, then the formality of the RAISE language may provide a convenient medium for the expression of requirements. Outside of that circumstance, they conclude that the use of RAISE as a specification tool is not generally justified.

13.2.3 Laboratory information management system case study

Clinical biochemistry is a laboratory discipline in which analyses are performed on patient samples providing information which clinicians use to diagnose illness, prognosticate and monitor the effects of treatment.

In a modern, computerised environment, test requisitions are entered into a computer system which is bidirectionally interfaced to automated analytical equipment, the most powerful of which are capable of producing thousands of test results per hour. Within two hours of arriving in the laboratory, this data is conveyed back to the clinician. Many laboratories depend on printed reports for this purpose, but electronic requesting and transmission to terminals based in wards or in general practice surgeries, is becoming more common.

As incorrect test results can have mortal consequences, clinical laboratories have developed quality control (QC) procedures which monitor the performance of analytical techniques. For example, the quality of every analytical run is assessed by algorithms which take into account the degree to which values reported for QC material deviates from the target values (in-house QC). Typically, one QC sample alternates with twenty patient samples. The run is rejected and repeated if the deviation is above a critical limit. In addition to 'in-house' QC checks, clinical laboratories subscribe to 'external' QC schemes, in which test samples are sent to participating laboratories for analyses.

The Laboratory Information Management System (LIMS) in the Biochemistry department at West Middlesex Hospital was designed, coded and implemented by laboratory staff. The LIMS was tested in the conventional manner (checking that data retained integrity throughout the system). The case study was designed to test the hypothesis that the safety properties of the LIMS could be enhanced by re-implementing components of the system using the RAISE Methodology and Toolset. Safety properties would be measured before and after re-implementation using formal safety techniques.

The first task was to produce a system definition of the LIMS. This would constitute the 'substrate' on which the formal safety assessment was carried out. Thereafter, portions of the LIMS would be respecified using RAISE.

The RAISE Methodology in the Laboratory

The laboratory team were attracted by the philosophy behind these techniques: a software specification could be transformed by a rule driven, step wise approach from the generalities of natural language to a specification so rigorous that it could, without ambiguity, be translated into code. The methodology tracked critical properties across conceptual boundaries ensuring a

predictable and safe outcome when different modules were brought together to form a complete application.

The reality was less exciting than the philosophy. The methodology was seen as highly specialised. Meaningful RAISE can only flow from the pen of an expert trained in its use. The laboratory staff learned to read RAISE but, because of the time constraints, did not become proficient writers.

Watching the Morse RAISE expert compose was a revelation in that output seemed effortless, comparable to the composition of text in natural language. In fact, it seemed easier in some respects because the 'vocabulary' and notation were so restricted. Watching also lead to the impression, that in common with the HAZOP and FMECA, the value of the output was more determined by the intelligence and overall understanding of the operator, rather than by the methodology itself.

This realisation strengthened as the project progressed. RAISE appeared to be efficacious, not because of any intrinsic properties but because the methodology was being used by someone who 'knew exactly where he was going'. In other words, the operator had a template in his mind as to how the design should appear. The methodology was guided along this path a-priori and not because of its innate logic. Understanding guided the method; the method did not seem to guide understanding.

In view of these observations, the laboratory staff would like to carry out an experiment in which the identical natural language specification was translated into pseudocode by three operators working independently with the RAISE Methodology. Significant inter-operator variation is predicted.

The Requirements Document

The laboratory team were completely unprepared for the rigours demanded by the preparation of the requirements document. Superficially the exercise seemed similar to the system definition, but the reality was completely different. The system definition describes an existing entity in great detail. The requirements document is a method of conveying, to a third party, the design and intent of a proposed system.

The problem, from the laboratory staff's point of view, was that there appeared to be no rules of engagement. No formulae or models seemed to exist to help the unwary translate ideas into statements that are clear and unambiguous.

The English language, often described as the language of science, is perhaps unsuited for the purpose of specification because of its richness. Many words can have similar meaning and the same words in different settings can have different meanings. This problem was evident when the software engineers tried to capture the requirements of the LIMS. Despite careful selection of specific words and construction of logical text, the laboratory staff experienced difficulty conveying ideas clearly and unambiguously.

This problem was resolved by instructing the software engineer with a course in the fundamentals of pathology. Progress was rapid after this, leaving the laboratory staff with the

belief that for the best results, requirements documents should be composed by operators with a special interest and training in the subject of interest.

The laboratory system study in summary

Although viewing RAISE with scientific scepticism, the laboratory staff accept that it ensures high level requirements are implemented at all stages of the development process. Whether this would be achieved without RAISE, is worth verifying. Finally, the method is difficult to learn and produces output which is not intuitively understood. For hard-pressed laboratory staff trying to keep abreast of their own subjects, the time needed to master RAISE represents a significant barrier to its widespread adoption in the clinical laboratory environment.

13.3 Reverse engineering by formal transformations

As described in Chapter 10, the BYLANDS project has been extending the formal program transformation system and wide spectrum language WSL developed at Durham University, to incorporate concurrency and real-time constructs into what was originally a sequential imperative system. WSL is based on an imperative kernel language consisting of only two statements: an atomic specification statement and a guard, plus three compound statements: sequential composition, non-deterministic choice, and recursion. The semantics of the language are specified denotationally, i.e. in terms of initial and final states, and program equivalence, refinement and abstraction are defined using the familiar weakest precondition approach of the refinement calculus. Equivalence-preserving transformations and program refinements are proved either by proving equality of weakest preconditions, expressed using infinitary first-order logic, or directly from the denotational semantics.

Modelling of concurrency is achieved by defining a structural operational semantics for (at present a subset of) WSL and adding to this a parallel combinator for statements defined by interleaving. This semantics has been formally defined using the Lego proof development system [BYLANDS *ref* 3]; a spin-off from this process has been the formal proof in Lego of a selection of transformations from the original Maintainer's Assistant transformation tool, which gives the project confidence in the viability of the approach. The primitive parallel composition construct can then be used together with local variable blocks, etc., to define higher-level concurrency structures such as concurrent processes, which are important in modelling real systems.

A notion of program equivalence in this extended language is also required in order that transformations may be defined and proved. Bisimulation equivalence as defined in the CCS formalism and elsewhere has been adopted as having properties appropriate for software maintenance. Bisimulation formalises the notion of "observational equivalence", making it possible to regard a program in effect as a "black box" with a specific external behaviour. Programs or sub-programs are regarded as equivalent if they cannot be differentiated by observing their visible behaviour. This implies that a program or component of a system should be replaceable by one which is equivalent under the above definition, and the resulting system should be equivalent to the original. This is very important in ensuring scalability to industrial-size systems; it is impossible for a practical tool to manipulate an entire software system en-masse, and undesirable that it should have to. In practice, maintenance and reverse engineering are performed on components of a system, and one of the reasons for the success of the

Maintainer's Assistant is its ability to transform programs and subprograms in such a way that the resulting program can directly replace the original.

The operational approach gives semantics to programs as generators of sequences of states; this has a natural expression in temporal logic, which may be formalised in Lego too. For this reason the project has so far concentrated on the formalising of concurrency, in the knowledge that real-time issues may be addressed by adopting an appropriate temporal logic (for example metric or interval temporal logic).

The project is due to finish in January 1998. At the time of writing, the formalisation of low-level WSL with concurrency has been completed, and case studies in reverse engineering some standard concurrent programming examples are being carried out. For further information about the initial work on concurrency, see [BYLANDS *refs* 1 and 2].

By the end of the project it is intended that the following will have been achieved:

- Transformations will have been identified and proved valid in concurrent systems. This will include existing transformations which remain valid in concurrent systems, and also new transformations specifically tailored to concurrent programs.
- First level WSL will have been extended to include concurrent processes, defined in terms of the primitive concurrent execution construct.
- A method for reverse engineering from concurrent WSL programs to CCS (Calculus of Communicating Systems) will have been developed, and case studies carried out.
- Domain Specific Languages will have been defined and evaluated for one or two specific domains.
- Principles and criteria for the design of such languages, including underlying the formalism and the interface to concurrent WSL will have been specified.
- The applicability in safety case analysis will have been demonstrated.