Formal Methods: Use and Relevance for the Development of Safety Critical Systems

Leonor M. Barroca* John A. McDermid †

1 Introduction

We now are starting to see the first applications of formal methods to the development of safety critical based systems. However, discussion on what are appropriate methods and tools is still intense and there is no standard approach that presents a complete solution for the formal development of such systems. Some of the protagonists claim (or at least are said to claim by their detractors) that formal methods offer a complete solution to the problems of safety critical software development. Others claim (or at least are said to claim by the formal methods protagonists!) that formal methods are of little or no use — or at least that their utility is severely limited by the cost of applying the techniques. The aim of this paper is to try to cast some light on this debate and to discuss from a technico-philosophical viewpoint the benefits and limitations of formal methods in this context. It is, perhaps, useful however to expose our prejudices now by summarising our view — *formal methods are both over-sold and under-used*.

In order to provide justification for this view it is necessary first to lay some terminological groundwork and to consider current practices. The term *formal method* is widely used, but with differing meanings. In this paper we use the term to refer to methods with a sound basis in mathematics. We use the term *structured method* to refer to methods which are well defined but which do not have a sound basis in mathematics for (completely) describing functionality. Technically the most significant difference between the two classes of techniques is that formal methods permit functionality to be specified precisely whereas structured methods only allow system structure to be specified precisely. (Interestingly many formal techniques are weak at describing system structure and boundaries.) In practice some formal techniques also explicitly address other, non-functional, aspects of systems eg their timing behaviour.

It is possible to distinguish five types, or classes, of formal methods which can be roughly characterised as follows:

1. *model based approaches* — giving an explicit, albeit abstract, definition of system (program) state and operations which transform the state, but giving no explicit representation of concurrency — eg Z [[Hay86, Spi89]] and VDM [[Jon86]];

2. *algebraic approaches* — giving an implicit definition of operations by relating the behaviour of different operations without defining state, again giving no explicit representation of concurrency — eg OBJ [[GT79]] and PLUSS [[Dau89]];

3. *process algebras* — giving an explicit model of concurrent processes and representing behaviour by means of constraints on allowable observable communication between the processes — eg CSP [[Hoa85] and CCS [[Mil89]];

4. *logic based approaches* — a variety of approaches using logic to describe properties of systems, including low level specification of program behaviour and specification of system timing behaviour — eg temporal and interval logics [[Gal87, Kro87]];

5. *net based approaches* — giving an implicitly concurrent model of the system in terms of (causal) data flow through a network, including representing conditions under which data can flow from one node in the net to another — eg Petri Nets [[Pet77]] and Predicate Transition Nets [[Vos80, GL81]].

*University of York
†University of York, York Software Engineering Ltd

In practice the distinctions are not always clear and there are hybrid methods which incorporate facets of more than one approach. Most of the methods have set theory and predicate logic as their underlying basis so there is some technical similarity between all the approaches. However there are significant differences between the expressive power of the methods and this was the essence of our classification above. In commenting on formal methods we will, where appropriate, identify the classes of methods to which the comments apply.

Formal methods can be used in two distinct ways. First, they can be used for production of specifications which are then used as the basis of a fairly conventional system development. Second, formal specifications can be produced as above, then used as a basis against which the correctness of the program is verified (proven). In the first case the mathematics is used, essentially, as a documentation medium. The benefits of the formalism include precision, abstraction, conciseness and manipulability. Manipulations might include consistency checking, automatic generation of prototypes or animation, and derivation of properties by means of proof. In the second case similar benefits accrue but, in addition, it is possible to prove the correspondence of program and specification — to show that the program does what it is specified to do — thus giving software development the same degree of certainty as a mathematical proof.

Structured methods are used fairly widely in industry. Formal methods are used much less widely, but their use is on the increase. In practice most industrial scale applications of formal methods have involved model based approaches where programs were developed 'conventionally' from formal specifications. Formal verification of programs is much less common and the main examples, outside academia, are in the security community in the USA.

There are some examples of the use of formal methods for safety critical systems, most notably by Rolls Royce and Associates [[Hil88]] and at the Darlington reactor in Canada [[PAK90]]. Reports from such projects indicate that formal methods were effective and contributed to the success of the work. Thus there is some practical evidence that formal methods are of utility in producing safety critical systems, although it is always difficult to isolate the factors that lead to successful projects. Also the use of formal methods is advocated by a number of standards, most notably DefStan 00-55 [[MoD91]] in the UK. This standard implies that the techniques are of central importance in the development of software for safety critical systems.

The paper is based on the premise that formal methods are, in principle, valuable to industry for at least some aspects of the development of safety critical systems and that their introduction represents a significant step in the evolution of software development towards a true engineering discipline. However there are theoretical and philosophical limitations to the methods and it is not entirely clear how relevant and useful the methods are for solving the particular problems encountered in the development of safety critical systems. This is the main point which we hope to illuminate in this paper. As well as discussing limitations of formal methods, in principle, the paper sets out what the author sees as being practical problem with formal methods, *vis a vis* application in the development of safety critical systems, given their current state of development.

In section 2 we set out the issues which have to be addressed in developing software for safety critical systems focussing particularly on how we gain confidence in the safety of systems containing software. In subsequent sections we discuss the (potential) role of formal methods in the software development life-cycle. This enables us to return to our main concern: the utility and relevance of formal methods, both in principle and in practice, in the development of safety critical computer-based systems.

2 The Development of Software for Safety Critical Systems

Even when used in a safety critical application software cannot, directly (of itself), cause loss of life but it may control some equipment that can cause loss of life. Thus software can contribute to the safety (or otherwise) of a system. In practice we often apply the term safety integrity to software to denote the extent to which the integrity (freedom from impairment) of the software contributes to the overall safety of a system.

We might think that we simply require software in safety critical systems to be highly reliable, however this misses a key point. First, software can fail frequently but still not lead to unsafe behaviour — if the failures do not cause hazardous consequences[1] . Second, reliable software can be unsafe — if in the rare event of failure there are catastrophic consequences. This suggests that we need to consider both failure modes and their consequences. However, for the purpose of discussing the effectiveness of formal methods, we need to focus primarily on failures. Although we cannot take reliability as the only measure of safety, or safety integrity, we must accept that reliability remains a valid measure and objective — so long as it is related to classes of failure which can lead to hazards.

[1] Reports indicate that there have been five 'anomalies' in the software controlling the trip systems in the French nuclear power plants, but none of these have lead to safety related 'incidents'.

2.1 Safety integrity goals and assurance

In this section we discuss the objectives of techniques for producing software with a high degree of safety integrity — although following Laprie we more often use the term dependable, or dependability [[Lap86]]. Also we present some fundamental principles which we believe facilitate the assessment of the contribution to safety of various (alternative) software development techniques. To simplify the discussion we will assume that the system to be produced is to be assessed by some agency independent of the developers — this is the case in many industries, eg civil aerospace, and probably should always be true where human life is at stake. We also assume that normal software engineering discipline is applied (see for example [[MB87]]) and focus on the *additional* issues which affect dependable systems.

A characteristic of safety critical systems is that a failure can be catastrophic. Thus in developing software for safety critical systems we have to achieve two distinct goals:

1. to develop the software in such a way that it is impossible or extremely unlikely that its behaviour (execution) will lead to a catastrophic failure, and

2. to provide *evidence* that will convince both the developers and the assessment authority of the dependability of the software (that the software will not, or at least it is very unlikely to, cause catastrophic behaviour in its operational environment).

Note that the above points cannot be established for software in isolation, but we will deal with software as independently of its operational environment as possible. We used terms such as 'extremely unlikely' above without quantification. Ideally we would like to attach a reliability figure or probability to these undesirable events. However this is not necessarily straightforward as we noted above, and we will return to this point later.

As a consequence of the above observations we can see that we would like to achieve and to demonstrate, for the software in a system that:

1. its requirement specification does not admit (allow) executions which would lead to catastrophic failure in its intended operational context;

2. it is free from design flaws which could lead to catastrophic failure in its intended operational context; ie that it satisfies its specification or, at least, the safety relevant portion thereof (note that this might involve taking into account new failure modes which are only apparent at the design, rather than the requirements level);

3. it can protect itself against the failures of other components of the system (which are not trapped by other means, eg hardware memory protection), and from external threats or attacks which could cause catastrophic failure.

These are objectives and it is is useful to discuss the degree to which the objectives are attainable.

Demonstrating to our complete satisfaction that we have achieved the first objective, ie adequate specifications, is generally accepted to be impossible (see for example [[Lev86]] for discussions of this point). In essence the difficulty is that we do not have any way of knowing that we have identified all the possible threats to, or failure modes of, the system so we can never be sure that our specification(s) is(are) complete. However it is possible to apply techniques which reduce the likelihood that the specification is catastrophically flawed (see section 3.2 below).

As indicated above design is a fallible human activity, but it is rather less problematical than specification, so we can (usually) be rather more confident that we have got the design and implementation 'right' with respect to the specification than that we have got the specification 'right'. Clearly the distinction arises because, once we have written the specification, we have bounded the issues which we need to address in the later stages so we are less likely to make major omissions in the design and implementation. We have previously used the term *assurance* for the degree of confidence that we have in the specifications and design [[McD89b]] and we amplify on the issue of levels, or degrees, of assurance below.

There are generally applicable techniques which can assist with the third point, eg solutions to the so-called Byzantine Generals problems [[LSP82]] where each system component assumes that all other components can fail in any manner, including maliciously. There are also techniques, eg the work of Ezilchelvan *et al* [1986], which are effective in the face of rather less pessimistic fault assumptions. However achievement of protection against failures is largely application dependent so we will primarily concern ourselves with the first two points.

As indicated earlier we cannot have complete confidence that we have achieved safety integrity. Instead we need to achieve assurance, or confidence.

Assurance is based on a number of issues including the level of trust we have in the individuals carrying out the development, etc. However one of the main contributing factors to assurance is the *evidence* produced during software development — and this in turn derives from the verification and validation activities which we carry out throughout the software development process (see also section 3).

It is common to equate validation with answering the question 'are we building the right thing?' and verification with answering the question 'are we building the thing right?'. Clearly this interpretation of the terms identifies validation as dealing with the first of our three demonstrable properties above, and verification as dealing with the second point.

Whilst we have some reservations about these terms, we continue to use them as they are in widespread usage. A key issue for us is how much assurance do we get from particular verification and validation techniques.

2.2 Fundamental Principles of Assurance

Assurance could, in principle, be based on reliability figures if they could be linked to catastrophic (rather than non-critical) failures. However it is generally accepted that it is not practical to assess reliability at the high levels required for safety critical systems [[Lit89]]. Further we have previously argued [[McD89b]] that deployment decisions for critical systems are actually made on subjective grounds (perhaps subjective reliabilities) not calculations of reliability based on frequent data because of the uncertainties introduced by the inherent limitations of synthetic reasoning. Thus we present the principles which we believe underlie the choice of software engineering techniques in terms of assurance.

Assurance can be thought of as confidence — based, of course, on objective evidence. Our fundamental tenet is that assurance arises from *comprehension* and *diversity* (perhaps the terms *understanding* and *independence* are more evocative). Simplistically we can say that the greater is our comprehension of some artifact, the greater is our confidence about the dependability of the artifact. There is nothing remarkable about this statement it simply reflects the fact that confidence increases with understanding. Similarly confidence increases with the number of independent, or diverse, ways that we have arrived at compatible or equivalent understandings of the system.

More practically we recognise that in developing or evaluating a putatively safe system we may discover a flaw, or flaws. Clearly discovery of a flaw reduces our confidence in the dependability of the system. Thus we can define assurance in the following way:

> Assurance that we have correctly assessed the dependability of an artifact increases as our comprehension of the artifact, and the number of ways we have obtained compatible understandings, increases.

Thus we need to base our discussion of which methods and techniques to use in achieving dependability on the criterion of which yields the greatest understanding of the system under development. For a simple artifact we may be able to gain sufficient comprehension of the artifact itself that we can *directly* assess its conformance to the specification (and the 'validity' of the requirements). For a more complex artifact we may find it impossible to gain adequate comprehension directly, or simply more cost-effective to gain assurance in the process. In practice it is helpful to address assurance from both the product and the process points of view, ie from the point of view of what is produced and how it is produced.

Also software tools are extensively used in developing dependable systems. The use of the tools is nugatory unless we can trust them. Consequently we require assurance in the tools themselves! Thus assurance in tools is one of the factors influencing assurance of a 'target' system, and, for very simple artifacts, greater assurance may arise *without* the use of tools as the benefits of using the tools may be outweighed by the need to comprehend them (to gain assurance in their correct functioning). In practice this probably means that manual techniques are more effective only for programs of a few tens, or hundreds, of lines of code.

The use of diversity in various forms of fault-tolerant systems, including design diversity, is becoming more commonplace. The principle extends to the development process. For example the use of more than one (independently developed) tool to carry out some analysis reduces the risk of common-mode failure, and increases confidence. Similarly, in the author's view, one of the psychological bases behind the value of formal techniques is that specifications, programs and proofs are redundant structures, and the risk of complete 'system' failure is reduced as failures (design or construction errors) in one form will probably be detected by comparison with the others. Thus we believe that diversity is a ubiquitous principle and that it can be applied to analysis methods, personnel, tools, and so on, but we will return to this point in relationship to formal methods later in the paper.

This discussion enables us to clarify the fundamental principle behind assurance.

Assurance arises from comprehension of, and diversity in, the complete procurement process, including the artifact which is developed, and the methods and tools used in its development and evaluation.

This principle should be evident in the ensuing discussion, although we focus more on the issues of comprehension than diversity.

3 Formal Methods in the Safety Critical Systems Life-cycle

Our aim here is to discuss the development process for safety critical systems and to indicate where, in principle, formal methods can be applied beneficially. It is hoped that this general discussion will become more clear and concrete when we discuss and illustrate particular formal techniques in appendix A.

3.1 The Software Life Cycle

We give here a brief overview of the nature and scope of the software life-cycle. A fuller description of life-cycle concepts and the important concepts of process design can be found in [[MR91]].

The software 'life cycle' is concerned with the development of software from initial concepts through delivery, use, and so-called maintenance. It is helpful to produce a generic model of the life cycle in order to have a basis for discussing different software development paradigms. Therefore we base our model on an abstract view of the activities carried out in software development and maintenance.

The first observation which we make is that, except for trivial systems, it is not possible to proceed directly from the initial concepts to executable software. Instead a number of intermediate system specifications are produced, eg requirements specifications. We refer to these using the generic term *descriptions.*

In general development proceeds from concepts, through requirements, etc. and one description is developed by some intellectual or automated process from the preceding description or representations. We refer to this process as a *transformation* although there is no implication that this is a purely automatable process and *synthesis* would perhaps be a better term.

In an ideal world the transformations would yield a sequence of descriptions, resulting in executable programs which satisfied their requirements and the initial concepts. In practice errors and infelicities are discovered during development (and maintenance) which cause iteration, ie repetition of the current transformation or rework of earlier representations. We use the term *Verification and Validation (V&V)* for the checking activities which may lead to iteration. We have already indicated the distinction between these terms above so it seems unnecessary to repeat any discussion here, but it is relevant to consider a distinction between forms of verification in the context of formal methods.

It is common to use the term *formal verification* to mean verification based on the concepts of mathematical proof. More strictly it means proofs where all the detail of the mathematical argument are presented. Clearly this is a form of analytical reasoning.

We can have very great confidence in the correctness (with respect to the specification) of a formally verified system, but the cost of gaining this confidence is very high (at the current state of the art, see below). Consequently the use of formal verification would only be justified where the cost of system failure is very high, eg in safety critical systems. Also the successful use of formal verification is contingent on proper tool support and this affects our views on assurance as the proof tools tend to be complex.

An alternative style of verification known as the *rigorous approach* [[Jon86]] involves the use of much less detailed proofs, or arguments, and 'obvious' truths would be accepted without any requirement to present an explicit argument in a rigorous proof. With the rigorous approach much of the benefit of formal proofs is gained at a much lower cost. It is probable that future, large scale, software development projects will be based on the rigorous approach.

3.2 Typical Development Stages

As indicated above there are many different approaches to software development adopted in industry. The following 'typical' model is intended to encapsulate the differing nature of the information being worked with at different stages in software development, without making commitment to any particular development methodology. It is intended that the model encompasses most real safety critical systems developments, ie we have erred towards including stages which might not always be employed.

Five stages are identified in addition to the concepts 'stage', see below:

1. *requirements specification* — description of the system and its operational environment, particularly stressing the interface between the system and environment;

2. *system specification* — an 'external view' of the system to be produced describing the system inputs, the system outputs and their relationships without describing internal system structure;

3. *architectural design* — a high level, internal view, of the structure of the system as it is to be produced — 'the grand plan' of the system like the architecture of a building;

4. *detailed design* — details of algorithms and data structures needed to implement the system;

5. *implementation* — the program source code (and the executable images).

The first two, Requirements Analysis and System Specification, are in the domain of requirements and this is usually summed up as representing *what* the customer or user wants. The remaining three are in the design domain and this is usually summed up as representing *how* the system developer intends to satisfy the requirements. In practice there may well be multiple stages of detailed design. We leave more detailed descriptions of the life cycle stages to the subsequent sections, but make some observations on the distinctions between the stages.

The what/how dichotomy is rather simplistic and, in practice, it is perfectly legitimate for customers or users to specify 'how' something should be done, eg to specify an algorithm. Similarly the system developer may have valid views on requirements — arising from a knowledge of similar systems or of implementation costs. In general it is more reasonable to say that requirements and design specifications will contain varying proportions of 'what' and 'how' information, and that the levels of description really represent degrees of commitment to implementation strategies [[DM90]]. In particular, for safety critical systems, requirements may place stringent constraints on system architecture in order to achieve some degree of fault tolerance and the what/how distinction is a fairly poor guideline to the distinction between requirements specification and design documents.

For the sake of clarity the following discussion takes a fairly 'pure' view of each of these stages of system development, but it should be borne in mind that any level of description may contain information which we might think of as being primarily related to one of the other levels.

3.3 The Role of Formal Methods in the Software Development Process

We discuss each of the above five stages in the development process and describe in more detail the characteristics of the descriptions and the role that formal methods can play in representing, producing and checking the description. To simplify discussion, we use the term 'target system' to describe the system being specified and implemented in cases where there might otherwise be ambiguity.

3.3.1 Requirements Analysis

Requirements Analysis is the first stage of the development process concerned with documenting the user's or customer's perceived needs by 'transformation' from the (by definition undocumented) initial concepts. The distinguishing characteristic of requirements analysis is that it is primarily an information gathering exercise which can only be validated, not verified (except for internal consistency).

The results of requirements analysis should describe both the *system* and the *environment* in which it operates. This is the case for two reasons:

1. the environment may change, impacting the functionality required of the system

2. the boundary of the system is not known *a priori* .

It is hard to bound precisely that part of the environment which should be considered in requirements analysis, but it should cover at least those systems, individuals, etc. which interact directly with the target system. In the case of safety critical systems the environment model should cover sources of threats to the system and other systems or equipments in which hazards could arise due to failure in the target system. The need to represent the environment means that requirements descriptions must be able to represent concurrency explicitly (because the system and processes in the environment operate concurrently).

In requirements analysis it must be possible to describe *non-computable* systems. This is both because users may ask for unrealisable systems and it is desirable to be able to record their requests exactly, and because it must be possible to record partial requirements, or requirements based on the assumption of infinite resources, which may arise as part of the information gathering process.

The results of requirements analysis are the primary basis for communication with the user and customer. For this reason it is desirable that the representation should be as precise as possible, eg formal. It is also necessary that requirements be intelligible to the customers as one of the primary forms of validation is review with the customer. However it is rare for users to be educated to understand the necessary formalisms. Consequently it seems that formal techniques either cannot be used at this stage, or if they are used some interpretation of the formalism is required for communication with the customer. For example it would be possible to use techniques of animation, specification execution, or derivation of properties by proof techniques in validation of requirements. In this latter case we might wish to prove that no sequence of operations which could be undertaken by the system (if it satisfied its specification) could lead to it (and the environment) entering an unsafe state. Animation is mandated by DefStan 00-55 [[MoD91]].

Technically requirements analysis methods need to deal with causality, eg 'when this event occurs in the environment the system must perform the following actions', and other properties such as behaviour of the system under hardware failure conditions. One of the key differences between 'normal' and safety critical systems is the need to be able to deal with causality in the presence of failure, and this is the reason that techniques such as failure modes effects analysis and fault tree analysis are used at this stage in safety critical systems developments.

There are few formal methods oriented towards requirements although the work of the Alvey FOREST project[[MKJ86, PF86]] is noteworthy as it deals with issues such as formally representing causality and giving guidelines for requirements capture.

Some recent work has been developed to represent timeliness requirements for safety-critical systems [[SAK90, SdLA91]]. The separation between safety and mission (functionality) is suggested for the formal analysis of requirements. This separation has been used for nuclear systems [[PvSK88]] and railway control systems [[CV91]]. Saeed uses Timed History Logic as a formal model for requirements specification.

The use of formal methods in the Requirements phase has added the possibility of animation to the already noted advantages of unambiguity, completeness and consistency [[JL89]]. Notations become more complete, addressing not only functionality but also non-functional requirements such as timing. However, they have not yet been able to combine power with expressiveness and intuitivity and there is still a long way to go to make the notations presentable to the user without (substantial) loss of precision.

3.3.2 System Specification

System Specification is still in the requirements domain, ie it is primarily concerned with what the system should do, not how it does it, although this is not always an easy distinction to make in practice (see below). The primary distinction between this and the previous stage is that it describes only the system, not the environment, and it gives precise definitions of the system interfaces. In practice the System Specification may be an enriched subset of the requirement specification and it should encompass both the system interfaces and its functionality.

In the contractual model of the life cycle the System Specification would be the basis of the contract for the development team. The implicit requirement for precision suggests that the specifications produced should be formal. Further the need to specify what not how suggests that it would be desirable to use algebraic specification techniques, ie techniques where the behaviour of a system is specified implicitly by equations relating inputs to outputs [[Zil74]].

Algebraic specification techniques have been widely applied to small examples but there is little evidence, as yet, that they are suitable for specifying large systems. In an algebraic approach we are forced, in some cases (for example, to establish the existence of an object in a database by reasoning about the sequence of inputs, eg creates and deletes, to the system), to be rather more obscure and cumbersome than the model oriented approach. There is a conflict between the theoretical attractiveness of algebraic approaches and their apparent practical limitations. However, the more operational techniques may compromise design freedom.

There is another important issue related to System Specification which can be illustrated by example. It is possible in an avionics system that some interfaces, eg to radar subsystems, would be specified very precisely during requirements, eg down to the level of the meanings of bits at the interface. However interfaces to other devices, eg a head-up display, may be known in terms of the information to be displayed but not in terms of the data formats, etc. Defining these formats is a design exercise which should involve human factors experts. In producing a System Specification the interface definition would have to be made precise so it will inevitably contain design information. The extent to which the System Specification will (implicitly) contain design information will depend on the nature of the system being built (recall our general comment above about the relationships between the different levels of specification).

The System Specification should be verified against the requirements. In practice this will probably be an informal exercise. Since design information may have been added it is also desirable that it is validated against the

initial concepts. It is possible that techniques of animation or specification execution [[CG87, HI88]] can be used in validation although, as pointed out above, System Specification may not initially contain enough information to allow execution of all aspects of the specification. For safety critical systems, further failure analysis may be appropriate, especially if it is possible that new failure modes can be deduced from the System Specification which were not apparent at the requirements stage.

There seem to be two possible ways in which formal techniques can evolve to become more applicable for this stage in the software development process. First algebraic techniques can be developed so that they are applicable to large scale systems. This will almost inevitably involve schemes for modularising specifications. Second it may be possible to find ways of applying the more operational techniques so that they don't unduly compromise design freedom.

In the more operational perspective it is worth mentioning here the more recent work of Harel [[HLNP90]] and Pnueli [[KP91]] on the specification of *reactive* systems. The Statechart approach with time constraints (Timed Statecharts) is a semantically well founded proposal for the specification of the behaviour of a system that interacts with an environment. It has at the same time the notable advantage of presenting a visual formalism and being amenable to animation.

P. Zave [[Zav82, Zav84]] has also proposed an operational approach to specification in a language called PAISLey (Process-Oriented, Applicative and Interpretable Specification Language), where she argues in favour of explicitly modelling concurrency.

3.3.3 Architectural Design

The Architectural Design describes the system interfaces, functionality and structure as the designers intend to implement it. The architecture is distinct from the previous stage in that it describes system structure and how the functionality will be achieved as well as what functionality is required. The level of detail contained in such a specification will vary from project to project. However it is not the level of detail which characterises the architectural design, but the fact that this is the first description of the system which is produced primarily from the developer's, rather than the user's, point of view.

Many different ways of producing formal specifications have been proposed, however the concept of architecture outlined above seems to match closely the ideas of *model-oriented* specifications and *process algebras*. We should refer here the extensions that have been made to model-oriented specifications in order to increase their structure. This work has been closely related to object-oriented extensions to the existing notations [[Log91]]. Arguably an 'ideal' approach would use a process algebra for specifying concurrent structure and communication but employ model-oriented specifications to state the behaviour of the operations engaged in by the processes.

A primary characteristic of the transformation from System Specification to Architecture is that it may not be structure preserving. In other words the structure of the design may have to be different from that of the requirement. This change in structure may be necessitated so that the system performs sufficiently quickly, so that the customer can afford it, or perhaps so that it has the appropriate fault-tolerance characteristics.

Ignoring, for the moment, the fact that software may not function correctly we can consider the effect of reliability requirements on architecture. If the reliability requirements can be met by a single (simplex) processor (because the available processor chips are of adequate reliability) then the architecture may follow closely the structure of the requirements with one 'design function' for each 'requirements function'. However, if this is not the case, then redundancy may have to be used thereby causing replication of function and introduction of new functions, eg for fault detection and system reconfiguration. In this case more than one design function would map to a function in the requirements and there would be functions which had no (direct) requirements counterpart at all. If we add timing requirements then we may find further changes in structure due to the fact that no one processor can keep up with the data coming from a sensor. Thus the limitations of current hardware technology are a primary factor in determining the design, but there are many other issues such as reliability, failure behaviour, timing behaviour, and so on. We can draw a number of points from this observation.

First, we have given non-functional reasons for the change in structure. In other words non-functional requirements such as performance, cost and reliability *drive* the design process. This is significant because formal specifications do not, for the most part, enable this non-functional information to be recorded. There are, of course, exceptions to this and some of the specification logics deal specifically with timing.

Second, many formal methods support a concept known as refinement (see for example [[Jon89]]), which enables us to define and verify the correctness of the relationships between two formal descriptions of the same system. However the published refinement techniques are usually too restrictive to admit the sort of structural change identified above, although current research work (see for example [[McD88, MW90, MS91]]) is addressing this problem, amongst others.

Third, we need quite a permissive interpretation of equivalence between the levels of representation. It must be possible to take in to account non-determinism, asynchrony, etc. which would mean, *inter alia*, that the order of the outputs would not be determined entirely by the order of the inputs. This may be particularly relevant where high priority inputs to a system can cause it to change operational mode and therefore 'ignore' other, 'lower priority' inputs. The notion of *behavioural equivalence* introduced in algebraic specification (see for example [[ST84]]) admits at least some of the requisite laxity in the meaning of equivalence but it is still a research issue to determine an appropriate set of refinement rules for dealing with the changes from System Specification to architecture.

It is also necessary to be able to represent concurrency within the Architectural Design. Notations such as Statecharts [[HLNP90, KP91]], already referred to above, are specially amenable to represent not only functionality but also the system interfaces and explicit concurrency. The primary problem associated with applying formal methods at this stage in the life cycle is that there is no method, or notation, which encompasses all of the requirements identified above. At present the would-be user of formal methods must choose the technique which best supports the characteristics which are most critical in his application area or to use an eclectic approach and to find appropriate ways of relating the different formalisms used.

3.3.4 Detailed Design

It is our view that detailed design should proceed from the architecture by the conventional process of (structure preserving) refinement. This is not a universally held view, indeed the phrase 'one man's design is another man's requirement' is often used in the software industry when discussing hierarchical specifications of systems. Given the interpretation of the relationship between requirements and design given above this would mean that the structure of the design could be changed in each representation. In our opinion this is an unhealthy attitude from at least two points of view.

Technically it implies that the architect did not have a complete (adequate) understanding of the system. This is particularly critical if the proposed changes involve modifying the process structure and hence impacting timing, etc. possibly to the extent that the system no longer meets its (non-functional) requirements. Clearly problems with the architecture may be found in detailed design: these should be resolved by updating the architecture, not making low level changes to the overall design.

Managerially it implies that the project is not under adequate control. For example modules common to several subsystems may have been identified for separate implementation and the basis on which this decision was made could be invalidated by allowing changes at this level. Thus even if the re-structuring preserves subsystem interfaces it could have 'knock-on' effects on the rest of the project and invalidate project plans, project resourcing, etc.

This structure preserving view of detailed design is consistent with (capable of being supported by) current refinement techniques (see for example [[Jon89, Mor90]]). The classical refinement techniques apply for sequential systems. Some techniques for dealing with concurrent systems, eg CCS [[Mil80, Mil89]], support hierarchical decomposition of systems which is akin to refinement. So far as we are aware there is no satisfactory formalism for dealing with the simultaneous refinement of both the concurrent and sequential aspects of a system. Again, in practice, it seems that in order to use formal methods for *all aspects* of detailed design and refinement to this level that it is necessary to take an eclectic approach and to work out on an *ad hoc* basis how to relate the different forms of specification.

3.3.5 Implementation

There has been considerable work on formal treatment of the final stage of development, that is formally relating a program to a low level specification. Techniques include the so-called 'constructive' approach, eg [[Bac86]] and program verification environments, eg Gypsy [[Goo84]]. The constructive techniques are methods based on the idea of deriving the program from low level specifications, and are intended to be applied manually. The verification environments are based on similar mathematical bases [[Hoa69]] to the constructive techniques but typically are more concerned with giving automated assistance to proof of correspondence between a program and a specification. Techniques for formal implementation are most well developed for sequential programs, but some work has been carried out for concurrent programs. The techniques are expensive to use and most of their uses to date have been in highly critical systems where the cost of failure justified the expense of applying the techniques in development. A considerable improvement in productivity using these techniques will be necessary before they can become more widely used.

The majority of these techniques are suited to the development of sequential programs, or at least programs which terminate. However many critical applications where the use of these formal verification techniques would be justified on economic grounds are continuously running programs, monitoring the state of some (physical)

process and taking the necessary remedial actions if the process is becoming dangerous, eg monitoring and controlling the flow of steel through a steel mill. Improvements in techniques for handling concurrency and continuously running programs will be necessary to handle this class of programs in a satisfactory manner.

Weaker forms of verification may be valuable under some circumstances. For example tools such as Malpas [[Bra84]] can carry out various analyses on programs, and these can be used to validate or verify the program. Capabilities of the tools include analysing control and data flow for undesirable features and establishment of the information flow in the program so it can be compared against the specification. More recently developed, the SPARK toolset [[CJM+90]] facilitates the formal proof of complete programs. It consists of a strictly defined subset of the Ada language, augmented by formal annotations, and a set of accompanying tools.

4 Strengths and Weaknesses of Formal Methods

In the introduction we made a number of comments regarding the strengths and weaknesses, or limitations of formal methods. We now return to these issues and endeavour to substantiate them as far as possible.

4.1 Strengths

We identified in the introduction a number of (purported) benefits of using formal methods. Our aim here is to amplify these points and to provide a justification for our views based, as far as possible, on the insight gleaned from the examples given in the appendix.

We asserted in the introduction that the benefits of using formal methods for specification included precision, abstraction, conciseness and manipulability. We address these points, and a few subsidiary issues, dealing with them first as issues of principle then assessing how close current methods come to these ideals.

Some of the points made below are not clear cut. To avoid circumlocution we state the positive view here and explain any contrary views in section 4.2 below.

4.1.1 Strengths — in principle

Specifications are primarily media for communication. That is they are intended to convey information from the producer of the specification to the reader, eg from the specifier of a module to the implementor. Alternatively they can be viewed as a means for documenting agreements, ie the specifier and implementor agree that the specification defines the interface to the module which is to be built. This is still a form of communication although it implies different degrees of responsibility for producing and verifying the document. A communication medium should be (or facilitate specifications which are) clear and unambiguous. This is not equivalent to saying that they are precise, abstract or concise, but there are relationships between these five properties as we will now show.

Ambiguity is easily dealt with. Formal notations are simply ‘sugared mathematics’ and hence they have have an unambiguous meaning, that of the underlying mathematical structures. More accurately the more sophisticated mathematical notions are built on more primitive notions, eg sets and propositional logic, and this means that there is a well defined *interpretation* of the formal notations and this is enough,in principle, to ensure consistent interpretation of specifications. We can now focus on the issues of precision and clarity.

Formal specifications are, or can be, very precise definitions because the semantics of the notations are well-defined and those of other media, eg English, are not. Other notations, eg those used by structured methods, are also precise but they are less expressive, eg showing structure not functionality, so formal methods give more *useful* precision than other approaches to specification. The direct benefit of the precision is that it reduces, or even eliminates, the risk of ambiguity and misinterpretation of specifications. Thus precision is a property of formal methods (or notations) and it is a major contributor to the production of unambiguous specifications. It should also be pointed out that this precision has a major pragmatic benefit in reviews — it is often possible to have very detailed and very constructive reviews when they are based on formal methods because there is no argument about what has been said, only about whether or not what has been said is what should have been said. In other words precision aids validation as well as communication.

The nature of the abstractions made possible by use of appropriate formalisms should be clear from the examples given in the appendix. Abstraction is one of our primary intellectual weapons for coping with complexity and it aids clarity by ‘drawing away from’ details which are not germane to our interests.

Clarity also arises from conciseness. As we indicated above formal notations vary in their ability to represent concepts concisely but, hopefully, they can be used to produce very compact descriptions. More importantly they can be much more compact than equally clear natural language descriptions whilst (normally) being more precise. To some extent this is borne out by the examples in the appendix (compare the length of the specifications with the

length of their prose explanations) but obviously the examples are a little biased by the fact that it was necessary to give a more tutorial level of description than would normally be the case.

The properties of abstraction, precision and conciseness all contribute to clarity. Good structure also contributes to clarity. In principle there is no reason why formal methods shouldn't yield good structure, but this doesn't seem to be an inherent property of the formalisms. This is perhaps an area where the structured methods are more effective.

In the introduction we stated that a valuable property of formal specifications is that they are manipulable, that is there are well defined rules for analysing and perhaps transforming formal specifications. This property can be used to show consistency of specifications and to derive important consequences of specifications, eg that processes can't deadlock or that a trip system is obliged to drop the control rods if the temperatures sensed go outside the valid range. Thus manipulability also aids in validation and it gives further abstractions — the derived properties — which can also help make specifications clearer.

In general it is possible to represent the mapping between a specification and the corresponding program within a formal framework. Obviously a very important aspect of manipulability, which we haven't been able to illustrate, is the possibility of verifying that the implementation, or at least the source code, satisfies the specification. More generally it is possible to reduce the verification of the mapping between levels of specification and between specifications and programs to a matter for formal proof. Thus, in principle, formal methods can offer very high confidence that the programs correspond to their specifications.

Finally it should be noted that formal methods are, in effect, a *lingua franca* — they will be (should be) interpreted the same way by readers of different backgrounds whether the distinctions are between their mother tongues or their professional disciplines. This truly is a property we require of a language for communication.

4.1.2 Strengths — in practice

It is interesting to consider the extent to which the above strengths are realised in practice. In section 4.2 we discuss weaknesses so our aim here is not to be directly critical but simply to observe which of the above supposed strengths are manifest in practice. The simple answer is all, to some extent!

Formal methods are perhaps most effective as a form of communication and for agreeing and documenting (design) decisions. The properties relating to ambiguity, clarity and so on are not fully substantiatable (see below) but, nonetheless, they do offer an effective medium for communication — between cognoscenti.

These observations are borne out by industrial experience. The use of formal methods in industry is not widespread but, where they have been applied, the evidence is encouraging. It is always difficult to make valid comparative analyses of the effectiveness of software development technology but, for example, IBM Hursley report a reduction in development costs of 9% through the use of Z on CICS [[Phi90]] and a significant improvement in fault rate, although the formally specified version of the product is not yet on full release. In the context of safety critical systems probably the most notable examples of the use of formal methods are by Rolls Royce and Associates and by RSRE on VIPER. In both cases significant quality benefits were attributed to the use of formal methods.

Thus there is relatively little evidence about the use of formal methods on real industrial projects of any nature, and even less on those involving safety critical software. Nonetheless what evidence there is indicates that the strengths discussed above are found in practice, albeit with some limitations. The biggest limitation, in principle, probably relates to the issue of ambiguity. The biggest problem in practice relates to manipulability, largely due to the paucity of effective tools. We return to these two points below.

4.2 Weaknesses

Unfortunately the existing formal methods do not fully live up to the ideal described above. This is mainly due to the state of development of current methods and their support tools, but there are also some issues of principle which run counter to those set out above, or which at least indicate limits to what they mean in practice for formal software development.

4.2.1 Weaknesses — in principle

The most fundamental weakness, or limitation, relates to the problem of specification validation to which we alluded earlier. We may be able to carry out development from the specification with 'mathematical certainty' but we will always have doubts about the veracity of the initial specification.

Clearly it is extremely valuable to remove doubts associated with software development but, unfortunately, most evidence suggests that the primary source of (significant) software errors is the specification — and safety

critical systems are, if anything, more prone to this sort of problem [[Lev86]]. At best this means that the mathematics, of itself, is insufficient to assure safety. Perhaps more significantly we are now faced with a value judgement about the level of effort we should put into formal development as against the effort we should place on means of validating the top level specification. It should be noted that we can use proof techniques to assist in validation, eg by deriving safety properties from a specification, but this simply reduces the 'gap' between formalisms and the 'real world', and doesn't eliminate it. Thus we know that we cannot simply rely on formalism to achieve and demonstrate safety. We will return to this general issue from a more pragmatic perspective after considering ambiguity and the nature of safety properties.

Another major, although less clear cut, limitation is to do with interpretation of specifications. Formal specifications do not just have an interpretation in terms of the underlying mathematics, they are also interpreted by software engineers in terms of a computational model and by system users in terms of a model of the use of the system in its operational environment. The issue of ambiguity then becomes not one of the existence of a unique model for the specification in the underlying logic but of compatibility of interpretations made in different domains by individuals with differing backgrounds and knowledge. Formal specifications are still less ambiguous than most prose, but they cannot be said to be free of ambiguity in any absolute sense as they are open to interpretation. This weakens, but does not negate, this strength of formal methods.

Another fundamental issue is that so-called 'non-functional' requirements and properties such as safety and security cannot be adequately articulated within a first order framework. This is a somewhat subtle technical point which is best illustrated by example. Consider the requirement for a system .to tolerate single point failures. At the level of system architecture, this may be interpreted to mean the failure of single processor/memory units. At the level of software module specification this may be treated as failure of a procedure invocation, and at a lower level it may be interpreted as the failure of a single logic gate or transistor. In other words the requirement is re-interpreted in terms of the relevant abstractions at each stage in the development process. Thus we view properties such as safety (which may encompass notions of fault-tolerance) as being higher-order in that they are really specifications which apply to other specifications.

In order to link formal specifications to the 'real world' and to guide the interpretation of the specifications we give prose descriptions of the basic entities specified and other fundamental notions. In a prose specification we always have to work with such informal descriptions. With a formal specification we can work largely within a analytical framework, subject to the need to re-interpret parts of the specification such as the notion of fault, once we have established the primary links between our specifications and the 'real world', so there is reduced scope for errors of misinterpretation. Thus the true limit of formal methods with respect to ambiguity and precision is that they can only reduce the scope for misinterpretation and other failings of specifications, not eliminate them. In practice, there are usually ways around such problems of principle.

We next address another issue of principle, which was not addressed under the heading of strengths above, and which has some practical ramifications. Once we realise that there is no such notion as absolute safety we have to recognise that we are primarily concerned with gaining assurance, or confidence, in safety not a guarantee. As we indicated in section 2.1 assurance arises from comprehension and diversity both of (or in) the product and the process. If we carry out formal proofs as well as producing formal specifications then we are producing artifacts of considerable complexity — in other words the proofs themselves are highly complex and difficult to understand. This leads to the question — does the use of formal proofs increase or decrease our comprehension and assurance in a software system?

It is hard to answer this question fairly from the point of view of principle because it is difficult not to be influenced by the capability of current program verification tools, so we defer discussion of this point. There is one further issue of principle, however, regarding formal proofs which we should raise. Certain properties of specifications and programs, eg whether or not they halt, are formally undecidable. This means that it is impossible to write a program, eg a theorem prover, that can decide (calculate) whether or not the undecidable property holds, eg that the program will halt. It is not often that such problems are encountered in practice but it is important to be aware of the perhaps surprising result that there are some properties which simply cannot be proven within a formal framework.

4.2.2 Weaknesses — in practice

There are many weaknesses or limitations of current formal methods. Our aim here is to give a brief survey of the most critical issues and to try to give a fair assessment of the likelihood that these problems will be resolved in the near future. As far as possible the comments build on the insights gained by studying the examples set out above.

The most striking aspect of many specifications is the forbidding symbology and, to a lesser extent, the arcane terminology. The mathematical abstractions embodied in notations such as Z and timed CCS facilitate brevity

and precision, but they do not necessarily contribute to clarity. Indeed there are many who would argue that the objectives of clarity and precision (or clarity and conciseness) are fundamentally opposed. In part this is an educational issue which we will return to below but there seems to be some substance in this criticism as even seasoned users of formal methods often have difficulty in reading someone else's specifications, at least until they get used to the style. In the authors' view this is because, in practice, we rely a good deal on the informal interpretation of the specifications, not their interpretations in terms of the underlying logic, in order to gain comprehension.

A somewhat related issue is that there is a high 'guff to stuff' ratio in many formal specifications. In other words it is often necessary to set out a lot of basic background mathematics which has no direct bearing on the problem in hand before we can directly specify the system of interest. In our examples this perhaps is most apparent with the Z specifications, although the author believes that this is a property of the type of problem specified, not the the Z notation itself. This problem is also clearly manifest with verification environments such as m-EVES [[CKM+87]] where it is often necessary to prove lots of elementary mathematical theorems in order to build a basis on which to reason about the program properties of interest. This directly affects clarity and comprehension as discussed above.

It could be argued that the formal specifications aren't really precise as the notations and semantics for the methods are not particularly well defined (at least in some cases). This is not an entirely fair criticism with the examples chosen but it is certainly the case that there are many variants of Z although there now is a standardisation effort as part of the IFIP founded ZIP project. Also some notations are considerably less well-defined than the examples we have used, so it is not always clear what is meant by a formal specification in practice, although they can, in principle, be made precise. In the case of Z we have the ability to extend the language, eg by adding new operators and there is no way of guaranteeing that these syntactic extensions are valid semantically as the language is currently defined; although, it would be possible to insist on proofs of soundness as have been provided for the Z mathematical toolkit. Thus current formal techniques are less well-defined than they might be, and there are some difficult compromises between expressive power, flexibility and precision of definition.

Although the above problems are to a large extent practical issues, it is our view they will not be solved in the short run, although it is to be expected that technical progress will eventually yield reusable specification libraries and more 'user friendly' notations, eg by linking formal and structured methods. It is also to be expected that formal methods will 'stabilise' and the quality of their semantic and syntactic definitions will improve (there is already evidence for this, eg there are moves to standardise VDM and Z[2] which are two of the leading model based specification approaches).

We have specification languages which are effective at representing functionality and certain aspects of concurrency. They are capable of representing some timing properties and more sophisticated notions such as permission and obligation. However there are limitations. The concept of time is very abstract and it is typically quite difficult to handle absolute clock time within the available specification formalisms (in fact there are considerable philosophical difficulties here especially when we need to deal with time in distributed systems where we cannot guarantee clock synchronisation). There are no well-defined ways of handling faults, or fault-tolerance, although this is an area where there is now some research being undertaken.

A related, and rather stronger point, is that current refinement techniques do not deal with timing and failure behaviour. That is we do not have well-defined rules for carrying out refinement in such a way that we can guarantee that the implementation we produce satisfies the timing and/or failure specifications. As almost all safety critical systems have to satisfy timing requirements and have to achieve safety even in the presence of failures this is a major drawback — although it is much less of a problem in 'mainstream' developments. There seems to be no reason, in principle, why the above problems should not be solved in the reasonably near future although the issues of refinement are quite subtle and it would perhaps be unwise to rely on solutions appearing within the next ten years.

A further major issue is the extent to which we have to trust tools. Clearly it is necessary to trust some of the tools we use, eg compilers and loaders, to some extent. The crux is the extent to which we have to trust complex tools, especially those which may be more complex than our application. In fact it is quite likely that any compilers and theorem provers used will be more complex than the application program. In many circumstances we have some form of independent check on the tool, eg we carry out testing on loaded code which gives an independent check (albeit probably far from exhaustive) on the compiler and loader. However, to a large extent, the tools have to be trusted except insofar as the testing and execution of the application gives an independent check. This is particularly worrying for tools such as theorem provers which are often complex heuristic programs. Proving compilers and theorem provers is a difficult task and certainly beyond the state of the art — although again these

[2] A draft ISO standard for VDM is due in September, and Z is proceeding in the same way.

are problems which are being researched. There is also a recursive problem — to what extent do we trust the tools used to verify the verification tools …Thus the use for formal methods and their support tools reduces certain classes of risk, eg that the specifications are inconsistent, but it does not remove all risks and introduces others, particularly in the area of trust in tools. Again it would seem unwise to rely on having solutions to these problems within a decade, if not longer in this case.

Finally we should not forget education and training. It is clear that few practising software engineers have the necessary skills to use formal methods. Perhaps more significantly there are few engineers with both the application domain knowledge necessary to help validate the specifications and the skills to write or read them, and this exacerbates the validation problem. It is relatively easy to give engineers a level of understanding of formal specifications which will enable them to read the specifications with confidence, but it requires considerable skill and experience to write good specifications. Much of the skill in fact lies in finding good abstractions, and simple understanding of the notation is far from adequate to guarantee the production of good specifications. Unfortunately principles of developing abstractions is not, as yet, something that even the formal methods experts know how to teach. However it is perhaps relatively easy to overcome this problem if industry are willing to make the investment in staff time for education and training.

4.3 Summary

It is hopefully clear that there are benefits from the use of formal methods and that some of the theoretical benefits are borne out in practice. However there are limitations, in principle, to what can be achieved with formal methods. At present, however, there are many more limitations reflecting immaturity of the techniques themselves and inadequacies of the support tools than there are philosophical problems. The difficult question which arises from this analysis is 'to what extent should formal methods form part of the development method for developing safety critical systems given their strengths and limitations?'. We address this point in our conclusions.

5 Conclusions

Our main aim here is to draw the discussion to a close by substantiating our claim about formal methods being both under-used and over-sold and to consider when and to what extent it is appropriate to use formal methods in the development of safety critical systems.

5.1 When and how to apply formal methods

Given the above discussions it should be clear that we are now entering the realm of value judgements. There is simply not enough information on which to base an objective evaluation of the relative contribution of formal methods, and other technologies, to the software and system development process. The following therefore represent our views based on a mixture of experience and assumptions about the prevalent classes of errors made in system development. It is worth noting, however, that there would be considerable benefit in carrying out experiments where different techniques were used to develop the same system to gain at least some evidence on which comparative judgements of method effectiveness could be based.

We would advocate the presence of formal methods throughout the several phases of the software life cycle (see section 3). There is no unified methodology that can be proposed for the whole development; we would use formal methods to produce top-level specifications for systems, but carry out development by a systematic application of stepwise refinement (informal variety) supplemented by formal refinement where there are adequate techniques.

In the phase of Requirements Analysis both the environment and the system are described, first building a model of the real world and then specifying the model of the computer system. The capture of the requirements is a vital stage and it is advisable that at least a set of well established guidelines would be followed. The representation of cause-effects relationships and non functional requirements such as time, resources, …, should be done in a formal framework where from the subsequent development can be achieved mainly by enrichment. Safety should be explicitly treated here dealing with the presence of failure. We would advise that when it comes to the definition of the model of the computing system, still as part of the requirements, it should be stated formally namely relations between inputs and outputs preferably in some notation that could be easily animated. It is against this model that the correctness of the final program is verified.

In the Design phases we would use an eclectic approach to specification. For example we would use a notation such as Timed Statecharts to represent concurrent and communication structure but specify the effects of the individual actions in another formalism such as Z; here, we would also advise the use of modularity, taking

probably a more object oriented approach such as ObjectZ. We would define a set of transformation rules that would allow the verification of the preservation of behaviour as structure and detailed functionality are added. Refinement as design becomes more detailed should be carried out in a semi-formal way. We would also derive a number of theorems, eg stating that the system won't deadlock, or giving a top-level statement of safety policy, but probably would reason about these (putative) theorems informally. We would use animation and simulation techniques, and methods such as Real Time Logic [[JM86]] to analyse timing properties. We would also link the formal techniques, so far as possible, to standard safety techniques, eg fault tree analysis. It would seem quite possible to apply such techniques in a manner analogous to the use of fault trees on programs [[LH83]].

When implementation is considered, we would link the specifications to techniques for schedulability analysis [[AB90, ABRW91, TBW92]] and program timing analysis [[ZBN92]]. We would use code verification techniques, eg SPARK, for the most critical code.

In summary we would supplement existing good practices with the use of formal specifications in order to gain clarity in top level specifications, to aid consistency checking of specifications and to assist in validation through derivation of key properties from the specifications.

5.2 Claim and counter-claim

Many formal methods protagonists clearly appreciate and clearly articulate the limits in principle and in practice associated with formal methods. Unfortunately, however, there are many counter-examples to this good professional practice — although much of the evidence is somewhat anecdotal. Nonetheless there clearly are occasions where unsubstantiated claims are made and, for example, the limitations of current techniques in terms of their expressive power or the capabilities of the support tools are 'glossed over'. Perhaps the best recent example of this is the claims made for VIPER, a formally specified microprocessor, where recent analysis has shown that the several claims made about the development were in excess of what had actually been achieved [[Ben90]].

It is perhaps also worth noting that the theoretical problems mentioned above also affect real system developments. As long ago as 1976 Gerhart and and Yelowitz [[GY76]] pointed out cases where formally verified programs had failed. In the examples cited the problems were that inappropriate proofs had been carried out, not that the proofs themselves were flawed.

On the other hand, many 'opponents' of formal methods say that the techniques are fundamentally flawed, or have no relevance, or …Again it is hard to separate fact from anecdote but some major textbooks on software engineering, eg [[MB87]], argue quite strongly that the techniques are still research topics so they cannot (even should not) be applied in industry, and that they have intrinsic limitations, essentially because of the problems of verifying refinements. There are already (limited) counter-examples to the first point. The second issue is much more substantive, however the key issue is not the substantiveness of the point but judging the extent to which the observed limitations actually matter in practice. In our view the limitations don't affect the value of formal specifications *per se* as a documentation and communication medium. However the issue of verifying refinements is a valid objection — but one that says we need to supplement proofs of refinement with other checks, not that the approach is fundamentally flawed. Nonetheless it is clear that we do not yet have adequate refinement techniques and that this is still a difficult research topic.

It would be easy to re-open the whole debate re: use and relevance — and we don't wish to do this. We hope to have now produced enough evidence to show that formal methods can be used effectively in industry. Since their use has been limited to date, our assertion about the benefits of wider use seems to be clearly true! The examples given above show that the techniques are sometimes over-sold and it would appear to be very easy to over-state their value. The theoretical benefits are very great and fairly clear, but the limitations are far more subtle and so it is rather more difficult to articulate them clearly and accurately. Also there is a temptation in trying to stimulate the use of formal methods to stress their value and to 'skate over' the limitations. This may not be deliberate over-selling but it has a similar effect. Thus we stand by the assertion that *formal methods are both over-sold and under-used*, but recognise that this is a simplification of a complex situation.

6 Provenance

This paper is based on a chapter to appear in 'Safety Aspects of Computer Control' edited by Phil Bennet to be published by Butterworth Heinemann in 1992. The chapter contains more examples which we believe substantiate the points made above.

References

[AB90] N. C. Audsley and A. Burns. Scheduling real-time systems. YCS 134, Dept. of Computer Science, University of York, 1990.

[ABRW91] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*. Atlanta, GA, USA, 1991.

[AGCL86] A Abdel-Ghaly, P Y Chan, and B Littlewood. Evaluation of completing software reliability predictions. *IEEE Transactions on Software Engineering*, SE-12(9), 1986.

[ATB+91] N. C. Audsley, K. Tindell, A. Burns, M. F. Richardson, and A. J. Wellings. The drtee artchitecture for distributed hard real-time systems. In *Proceedings 10th IFAC Workshop on Distributed Control Systems*. Semmering, Austria, 1991.

[Bac86] R Backhouse. *Program Construction and Verification*. Prentice Hall International, 1986.

[Ben90] P Bennett. *VIPER: A Perspective.* Centre for Software Engineering, 1990.

[Bra84] BD Bramson. Malvern's program analysers. *RSRE Research Review*, 1984.

[CCM90] S J Clarke, A C Coombes, and J A McDermid. *The Analysis of Safety Arguments in the Specification of a Motor Speed Control Loop*. Dept of Computer Science, University of York, YCS 136, June 1990.

[CG87] D Coleman and RM Gallimore. *Software Engineering Using Executable Specifications.* Macmillan Computer Science Series, 1987.

[CJM+90] B. Carré, T. Jennings, F. Maclennan, P. Farrow, and J. Garnsworthy. *SPARK: the SPADE Ada Kernel (third edition)*. Program Validation Ltd., 1990.

[CKM+87] D Craigen, S Kromodimoeljo, I Meisels, A Neilson, W Pase, and M Saaltink. m-eves: A tool for verifying software. Cp-87-5402-26, I P Sharp Associates Ltd, 1987.

[CV91] V Chandra and M Verma. A fail safe interlocking system for railways. *IEEE Design and Test of Computers*, 8(1):58–66, 1991.

[Dau89] P. Dauchy. Application de la méthode pluss de specification formelle à une fonction du metro de lyon. In *Journée AFCET-INRETS, Conception et Validation des Logiciels de Securité dans les Transports Terrestres*, 1989.

[DM90] J E Dobson and J A McDermid. An investigation into modelling and categorisation of non-functional requirements (for the specification of surface naval command systems). YCS 141, YCS 160, Dept. of Computer Science, University of York, 1990.

[ES86] P D Ezilchelvan and S K Shrivastava. A characterisation of faults in systems. In *Proc. 5th IEEE Int. Symp. Reliability in Distributed Software and Database Systems*, pages 215–222. IEEE Press, Los Angeles, January 1986.

[Fet88] J H Fetzer. Program verification: The very idea. *CACM*, 31(9), 1988.

[Gal87] A. Galton. *Temporal Logics and Their Applications*. Academic Press, 1987.

[GL81] H. Genrich and K. Lautenbach. System modelling with high-level Petri Nets. *Theoretical Computer Science*, 13:109–136, 1981.

[Goo84] D Good. Mechanical proofs about computer programs. Technical Report 41, Institute for Computing Science, The University of Texas at Austin, 1984.

[GT79] J. Goguen and J. Tardo. An introduction to obj: A language for writing and testing software specifications. In *Specification of Reliable Systems*. 1979.

[GY76] Gerhart and Yelowitz. Observations of fallibility in applications of modern programming methodologies. *IEEE Transactions on Software Engineering*, May 1976.

[Hay86] I Hayes, editor. *Specification Case Studies*. Prentice Hall International, 1986.

[HI88] S Hekmatpour and D Ince. *Software Prototyping, Formal Methods and VDM*. Addison Wesley, 1988.

[Hil88] J V Hill. The development of high reliability software - rras experience for safety critical systems. In *Proc. of BCS/IEE SE Conference*. Peter Peregrinus, Liverpool, 1988.

[HLNP90] D. Harel, H. Lachover, A. Naamad, and A. Pnueli. Statemate: A working environment for the development of complex reactive system. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[Hoa69] C A R Hoare. An axiomatic basis for computer programming. *CACM*, 12(10), 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[JL89] M Jaffe and N G Leveson. Completeness, robusteness, and safety in real-time software requirements specification. In *Proc. 11th International Conference on Software Engineering*, pages 302–311. 1989.

[JLHP85] M W Jones-Lee, M Hammerton, and P R Philips. The value of safety: results of a national sample survey. *Economic Journal*, pages 49–72, 1985.

[JM86] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–903, 1986.

[Jon86] CB Jones. *Systematic Software Development Using VDM*. Prentice Hall International, 1986.

[Jon89] C B Jones. Data reification. In John McDermid, editor, *The Theory and Practice of Refinement*. Butterworth Scientific, 1989.

[KMS87] J. Kramer, J. Magee, and M. Sloman. The conic toolkit for building distributed systems. *IEE Proceedings Pt D*, 134(2), March 1987.

[KP91] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 571*, pages 591–620. Springer Verlag, 1991.

[Kro87] F. Kroger. *Temporal Logic of Programs*. Springer-Verlag, 1987.

[Lap86] J-C Laprie. Dependability: A unifying concept for reliable computing and fault tolerance. Technical Report 86.357, LAAS, Toulouse, 1986. To appear in *Resilient Computing Systems* ed T. Anderson, Collins and Wiley.

[Lev86] N G Leveson. Software safety: What, why and how. *Computing Surveys*, 18(2), 1986.

[LH83] N G Leveson and P R Harvey. Analyzing software safety. *Transactions on Software Engineering*, SE-9(9):569–579, 1983.

[Lit89] B Littlewood. Predicting software reliability. *Proc. Trans. Royal Society*, 327, 1989.

[Log91] Logica UK Ltd. Comparative study of object orientation in z. Technical report zip/logica/90/046 issue 3.0, Logica UK Ltd., 1991.

[LSP82] L Lamport, R Shostak, and M Pease. The byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MB87] A Macro and J N Buxton. *The Craft of Software Engineering*. Addison Wesley, 1987.

[McD88] J A McDermid, editor. *Proceedings of Workshop on Theory and Practice of Refinement*. Butterworth Scientific, 1988.

[McD89a] J A McDermid. Assurance in high-integrity software. In C T Sennett, editor, *High-Integrity Software*, pages 226–273. Pitman, 1989.

[McD89b] J A McDermid. Towards assurance measures for high integrity software. In *Proceedings of Reliability'89*. The Institute of Quality Assurance, 1989.

[McD90] John McDermid, editor. *Software Engineer's Reference Book*. Butterworth Scientific, 1990.

[Mil80] R Milner. Calculus of communicating systems. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science No. 92*. Springer-Verlag, 1980.

[Mil89] R Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MKJ86] T S E Maibaum, S Khosla, and P Jeremaes. A modal [action] logic for requirements specification. In P J Brown and D J Barnes, editors, *Software Engineering 86*. Peter Peregrinus, 1986.

[MoD91] MoD. Defence standard 00-55, the procurement of safety critical software in defence equipment. Technical report, Ministry of Defence, 1991.

[Mor90] C Morgan. *Deriving Programs from Specifications*. Prentice Hall International, 1990.

[MR84] J A McDermid and K Ripken. *Life Cycle Support in the Ada Environment*. Cambridge University Press, 1984.

[MR91] J A McDermid and P Rook. Software development process models. *Software Engineers' Reference Book*, 1991.

[MS91] J Morris and R Shaw, editors. *4th Refinement Workshop*. Springer-Verlag, 1991.

[MT89] Faron Moller and Chris Tofts. *A Temporal Calculus of Communicating Systems*. LFCS, December 1989.

[MW90] C Morgan and J Woodcock, editors. *3th Refinement Workshop*. Springer-Verlag, 1990.

[PAK90] D. Parnas, G. Asmis, and J. Kendall. Reviewable development of safety critical software. In *International Conference on Control and Instrumentation in Nuclear Installations*. The Institute of Nuclear Engineers, Glasgow, U.K., 1990.

[Pet77] J. Peterson. Petri Nets. *Computing Surveys*, 9(3):223–252, 1977.

[PF86] C J Potts and A Finkelstein. Structured common sense. In P J Brown and D J Barnes, editors, *Software Engineering 86*. Peter Peregrinus, 1986.

[Phi90] M. Phillips. Cics/esa 3.1 experiences. In *Z User Workshop: Proceedings of the Fourth Annual Z User Meeting*. Springer Verlag, 1990.

[PvSK88] D Parnas, A J van Schouwen, and Shu Po Kwan. Evaluation standards for safety critical software. Technical report, Queens University, Kingston Ontario, 1988.

[Rea79] J Reason. Actions not as planned: The price of automatization. In G Underwood and R Stevens, editors, *Aspects of Consciousness*. Academic Press, 1979.

[Rei85] W. Reisig. Petri Nets with individual tokens. *Theoretical Computer Science*, 41:185–213, 1985.

[RG91] T. Ralson and S. Gerhart. Formal methods: History, practice, trends and prognostics. *American Programmer*, pages 2–14, 1991.

[Roo90] P Rook. Project planning and control. In John McDermid, editor, *Software Engineer's Reference Book*. Butterworth Scientific, 1990.

[SAK90] A Saeed, T Anderson, and M Koutny. A formal model for safety-critical computing systems. In *Proceedings IFAC Workshop SAFECOMP '90*, pages 1–6. 1990.

[SdLA91] A Saeed, R de Lemos, and T Anderson. The role of formal methods in the requirements analysis of safety-critical systems: a train example. In *Proceedings of the 21st Symposium on Fault-Tolerant Computing*, pages 478–485. 1991.

[Spi89] J M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1989.

[ST84] D Sanella and A Tarlecki. *On Observational Equivalence and Algebraic Specification*. Department of Computer Science, University of Edinburgh, 1984.

[TBW92] K. Tindell, A. Burns, and A. Wellings. Allocating real-time tasks (an np-hard problem made easy). *Real Time Systems*, 1992. to appear.

[Vos80] K. Voss. Using Predicate/Transition-Nets to model and analyze distributed database systems. *IEEE Transactions on Software Engineering*, SE-6(6):539–544, 1980.

[Wit69] L Wittgenstein. *On Certainty*. Blackwell, 1969.

[Zav82] P Zave. An operational approach to requirements expression for embedded systems. *Transactions on Software Engineering*, 8(3), 1982.

[Zav84] P. Zave. The operational versus the conventional approach to software development. *Coomunications of the ACM*, 27(2):104–118, 1984.

[ZBN92] N. Zhang, A. Burns, and M. Nicholson. Analysing assembler code for program execution time estimation. In *Spirits Workshop*. 1992.

[Zil74] S Zilles. Algebraic specification of data types. Technical Report 11, Project MAC, Massachussets Institute of Technology, 1974.

A Examples of Formal Methods

The aim in this section is to give a very brief overview of the nature of different types of formal methods in order to illustrate their characteristics and to try to substantiate some of the general points made above. Due to limitations on space the analysis is inevitably somewhat superficial so references are given to texts which give more comprehensive tutorial treatments of the methods discussed.

A.1 Model Oriented Specification

The Z specification language is based on set theory and first order predicate calculus. A distinguishing feature of Z is the use of schemas and the schema calculus. Schemas are 'modules' of specifications and the schema calculus gives a way of linking the modules to build up complex specifications from simple parts in a clear and elegant manner. Z was originated by J-R Abrial and has subsequently been developed by a number of staff at the Programming Research Group in Oxford. Some examples of the use of the language can be found in [[Hay86]] and a more definitive discussion of the language is given by [[Spi89]].

We will assume that the reader is familiar with the Z notation and will concentrate on a detailed example which shows the specification of some safety relevant properties.

A.1.1 Safety example

Our intention is to show the behaviour for a thresholding device such as might be used in temperature monitoring where it is necessary to compare the values from a number of temperature sensors, to reject values which are out of tolerance and to calculate an average of the values which are within tolerance. Such a function might be useful in many situations, eg process monitoring, but it is not based on any specific system or device.

We first introduce some basic definitions for representing sensor properties:

$$[Sensor]$$

The parachuted type Sensor represents the set of all sensors known about in the monitoring system.

$$\begin{array}{|l} upper, lower, bound, spread : \mathbb{N} \\ \hline lower < upper \\ spread < upper - lower \\ bound < spread \end{array}$$

The data items *upper* and *lower* represent the limits on legal values for the sensors: any values outside the range *lower..upper* indicate that the sensor has failed. The item *bound* is a limit on the difference between two successive values from a sensor representing the maximum allowable rate of change of value reported by the sensor. If any pair of successive values from a sensor are different by more than this bound then this will also be taken as evidence that the sensor has failed. Finally, *spread* represents the allowable divergence between any two functioning sensors. If some values do disagree by more than the allowed spread then their values are ignored, but the sensors are not assumed to have failed (this is intended to deal with cases where noise, etc. may affect values temporarily). The constraints represent the natural relationships amongst these data items. In practice we would need to specify the exact values to be used.

We now define a number of data types corresponding to the range of allowable values, the rate of change of sensor values and the coherence of the data values from the complete set of sensors. These are simply used as results from functions which evaluate the above checks on data validity. The first is used for checks on range:

$$status ::= legal \mid illegal$$

The second is concerned with allowable rates of change of sensor value:

$$rate ::= sensible \mid fast$$

The third is used for assessing data coherence:

$$coherence ::= ok \mid out$$

We are now in a position to define functions which evaluate the checks on data validity identified above. The choice of types for the functions is determined by convenience in representing state, see below. The function *valid* evaluates the range check on data validity and assigns the value legal or illegal to the result as appropriate:

$$
\begin{array}{|l}
valid : \mathbb{N} \rightarrow status \\
\hline
\forall n : \mathbb{N} \bullet \\
\quad (n \geq lower \land n \leq upper \Rightarrow validn = legal) \land \\
\quad (n < lower \lor n > upper \Rightarrow validn = illegal)
\end{array}
$$

We have used implication here dealing with each case separately. As the terms before the implication are mutually exclusive there is no ambiguity in the definition of the function.

The function for evaluating legal rate transitions is very similar to the check on absolute sensor value, but clearly needs to check pairs of values:

$$
\begin{array}{|l}
rate\_ok : (\mathbb{N} \times \mathbb{N}) \rightarrow rate \\
\hline
\forall n1, n2 : \mathbb{N} \bullet \\
\quad ((n1 - n2 \leq bound \Rightarrow rate\_ok(n1, n2) = sensible) \land \\
\quad (n1 - n2 > bound \Rightarrow rate\_ok(n1, n2) = fast))
\end{array}
$$

The coherence of a set of values is determined in a similar way, but here we use an equivalence between the function delivering *ok* and the condition when the data set is acceptable — in this way the behaviour of the function when the data is not coherent is defined *implicitly* as the only possibility is for it to deliver the value out, signifying that the values are incoherent.

$$
\begin{array}{|l}
coherent : \mathrm{seq}\,\mathbb{N} \rightarrow coherence \\
\hline
\forall s : \mathrm{seq}\,\mathbb{N} \bullet \\
\quad (\forall id1 : doms \bullet \\
\qquad \forall id2 : doms \bullet s\ id1 - s\ id2 \leq spread) \Leftrightarrow \\
\qquad coherent\ s = ok
\end{array}
$$

However it will not be enough to check coherence and we will have to find a sequence representing those values which are coherent. In doing this we may need to discard mappings from a sequence which contains incoherent values to create one containing only coherent values. However simply discarding arbitrary values might render the result an illegal sequence, eg the domain might be 1, 2, 4 which is illegal as 3 is missing (remember that sequences map from an initial segment of the natural numbers). We therefore need a function to turn arbitrary pairs of numbers into a sequence:

$$
\begin{array}{|l}
mk\_seq : (\mathbb{N} \pfun \mathbb{N}) \rightarrow \mathrm{seq}\,\mathbb{N} \\
\hline
\forall pairs : \mathbb{N} \pfun \mathbb{N} \bullet \\
\quad (\exists map, res : \mathrm{seq}\,\mathbb{N} \mid map \fatsemi pairs = res \land \#res = \#pairs \bullet \\
\qquad mk\_seq\ pairs = res)
\end{array}
$$

The above function, *mk_seq*, has the required property as the mapping sequence, *map*, converts pairs to a sequence and the constraints on the size of the result constrains map not to discard any elements of the function *pairs*. We can now use this function in calculating a sequence of coherent sensor values:

$$
\begin{array}{|l}
co\_seq : \mathrm{seq}\,\mathbb{N} \rightarrow \mathrm{seq}\,\mathbb{N} \\
\hline
\forall s : \mathrm{seq}\,\mathbb{N} \mid \#s > 0 \bullet \\
\quad (\exists s1 : \mathbb{N} \pfun \mathbb{N} \mid coherent(mk\_seq\ s1) = ok \land s1 \subset s \bullet \\
\qquad (\forall s2 : \mathbb{N} \pfun \mathbb{N} \mid \\
\qquad\quad coherent(mk\_seq\ s2) = ok \land s2 \subset s \bullet \\
\qquad\qquad (\#s2 \leq \#s1)) \Leftrightarrow co\_seqs = mk\_seq\ s1)
\end{array}
$$

The function finds the biggest subset of the sequence given as a parameter which is coherent (or one of them if there is more than one of the same size). This is done by ensuring (via the third quantifier) that any other coherent subset is no bigger than the one already found. If there is more than one coherent set of the same size then an arbitrary one will be chosen. Note that since a data value is always coherent with itself the function will, at worst, deliver a sequence of only one element. In this case, and with equal size sets with more than one element, the function is non-deterministic and we do not know which element(s) it will pick (this seems to be reasonable as we have no way of knowing which is the 'best' value if there is no agreement between the values). This specification

is not entirely straightforward, but this is probably a good illustration of the value of formal methods — it is very easy to see how an implementor given only an informal specification might implement such a function incorrectly.

We now have a rather simpler function which calculates the average value from a sequence. Since the values are integers the average will only be approximate. We have chosen to specify the bounds on legal average values rather than to indicate that the average should be rounded up or rounded down. This leaves freedom to the system designers and implementors. The definition uses a function *sum* (we omit its definition here because it is straightforward) that computes the sum of the elements of a sequence.

$$
\begin{array}{|l}
\mathit{average} : \operatorname{seq}\mathbb{N} \to \mathbb{N} \\
\hline
\forall\, \mathit{sens} : \operatorname{seq}\mathbb{N} \bullet \\
(((\#\mathit{sens}) * (\mathit{average}\ \mathit{sens})) < ((\mathit{sum}\ \mathit{sens}) + (\#\mathit{sens})) \land \\
((\#\mathit{sens}) * (\mathit{average}\ \mathit{sens})) > ((\mathit{sum}\ \mathit{sens}) - (\#\mathit{sens})))
\end{array}
$$

We have now completed the preliminaries and can define the system itself by introducing the state and some operations on the state.

We introduce an object to represent the sensors in the system. If we were wishing to produce a complete specification we would need to deal with the way in which the sensor values changed but for our present purposes the intention is that the function sensors represents the current values of the sensors.

$$\mathit{sensors} : \mathit{Sensor} \to \mathbb{N}$$

The state of the computer system checking the sensor values can be broken down into two parts. The parts are treated separately to simplify the specification (see below).

First, we have a pair of functions which contain the latest values read from the sensors and stored in the system (*new_values*) and the previous set of readings (*old_values*). There is no invariant as the only property of interest would relate to the 'freshness' of the data and, within this example, we are ignoring timing (we will return to this point later).

$$
\begin{array}{|l}
\hline
\mathit{SENS\_History} \\
\mathit{old\_values} : \mathit{Sensor} \to \mathbb{N} \\
\mathit{new\_values} : \mathit{Sensor} \to \mathbb{N} \\
\hline
\end{array}
$$

The second part of the state is concerned with the computer's model of which sensors are functioning correctly, and which are not. The set *failed* indicates those sensors which the computer system believes to have failed and *check_set* indicates the current set of values, drawn from the stored sensor values, which the computer is going to use to calculate the average sensor value, ie those that come from working sensors and which are deemed to be coherent.

$$
\begin{array}{|l}
\hline
\mathit{SENS\_State} \\
\mathit{failed} : \mathbb{P}\,\mathit{Sensor} \\
\mathit{check\_set} : \operatorname{seq}\mathbb{N} \\
\hline
\#(\operatorname{dom}\mathit{check\_set}) \leq \#\mathit{Sensor} - \#\mathit{failed} \\
\hline
\end{array}
$$

The invariant states that the number of values to be used as the basis of the check (calculated by an averaging mechanism) can never exceed the number of working sensors. Note that the number might be less than the number of working sensors due to coherence problems.

We can now define the first aspect of the operations to be performed by the system. Here we define the operation which reads the sensor values and updates the (short) history of values retained by the system. The definition is fairly straightforward and we see the value of treating the state in two parts as the *SENS_State* and *SENS_History* change values at different times (in all cases, not just this one).

$$
\begin{array}{|l}
\hline
\mathit{Read\_Sensors} \\
\Delta \mathit{SENS\_History} \\
\Xi \mathit{SENS\_State} \\
\hline
\mathit{new\_values}' = \mathit{sensors} \\
\mathit{old\_values}' = \mathit{new\_values} \\
\hline
\end{array}
$$

We now consider the checks on sensor data validity. We first consider the overall limits on sensor values. The schema calculates which sensors (if any) which have now failed as *new_fail* — despite the name this might include sensors that were previously known to have failed. The set *new_fail* is 'added' to the set *failed*. Changes to *check_set* are not specified — this doesn't matter as we will specify how the value of *check_set* is calculated later.

Check_Limits

$$\Xi SENS\_History$$
$$\Delta SENS\_State$$

$$\exists\, new\_fail : \mathbb{P}\, Sensor \bullet$$
$$\quad \{s : Sensor \mid valid(new\_valuess) = illegal\} = new\_fail \land$$
$$\quad failed' = failed \cup new\_fail$$

Here we say that the set **new_fail** is exactly the set of Sensors for which the function valid yields illegal (this is read rather like a quantified expression). Note that if a Sensor previously deemed to have failed gives a sensible reading we do not automatically reinstate it. This reflects an attitude that a failed sensor may drift and occasionally give legal, but erroneous, values and so its values should be ignored until it is explicitly reinstated. In this specification fragment we do not deal with reinstatement operations.

The rate of changes of the sensor values are calculated in a similar manner.

Check_Rate

$$\Xi SENS\_History$$
$$\Delta SENS\_State$$

$$\exists\, new\_fail : \mathbb{P}\, Sensor \bullet$$
$$\quad \{s : Sensor \mid$$
$$\qquad rate\_ok(old\_values\ s, new\_values\ s) = fast\} = new\_fail \land$$
$$\qquad failed' = failed \cup new\_fail$$

We can now determine the set of values which will be used for the check. Note that we do not discard sensors just because they are in disagreement with others — this allows us to discard noisy readings which probably were caused by noise without discarding the sensor. Again in a full specification we might care to record a history of disagreeing sensors and to discard them after too many disagreements.

Define_Check_Set

$$\Xi SENS\_History$$
$$\Delta SENS\_State$$

$$failed = failed'$$
$$(\exists\, map : \mathrm{seq}\, Sensor;\ values : Sensor \pfun \mathbb{N} \mid$$
$$\quad values = failed \lhd new\_values \land \mathrm{ran}\, map = \mathrm{dom}\, values \bullet$$
$$\qquad check\_set' = co\_seq(map \fatsemi values))$$

The operation for defining the sensor value to be delivered is now straightforward, being defined by calculating the average of the *check_set*. In addition we deliver the size of the *check_set* as a measure of confidence in the accuracy of the value.

Calc_Value

$$\Xi SENS\_History$$
$$\Xi SENS\_State$$
$$val! : \mathbb{N}$$
$$size! : \mathbb{N}$$

$$val! = average\ check\_set$$
$$size! = \#check\_set$$

We can now define the complete operation of a single checking cycle for the system, assuming that the checks are executed periodically. This is done by the following schema calculus expression:

$$Check\_Cycle == Check\_Limits \fatsemi Check\_Rate \fatsemi Define\_Check\_Set \fatsemi Calc\_Value$$

The forward relational composition between schemas is similar to that between functions except that it maps states to states, not results to parameters. Thus the after state of *Check_Limits* becomes the before state of *Check_Rate*, and so on. Note that the ordering of the operations is the same as the order of their definition. This is no accident as it helps to explain their behaviour — but note that it was much easier to understand the operation 'piecemeal' than it would have been if we had presented the complete predicate for the total operation 'in one piece'.

Clearly there are potentially other operations of interest for such a system but, hopefully, the above gives a clear definition of at least some of the requisite functionality, ie the basic checking mechanisms.

A.1.2 Commentary

We will comment in detail on the effectiveness of such specification techniques in section 4, however it is worthwhile drawing out one point here. In systems like the (hypothetical) one described above time is a very important property and we would probably want to specify the frequency with which the sensor values are checked, and the length of time needed to carry out the checks. There is no built in notion of time within Z so there is no pre-defined way of doing this. However it is possible to extend the Z language with notions of time and we could have expressed timing constraints if we so wished.

The next notation which we will consider is much more strongly oriented towards specifying temporal properties of systems and we will return to the general issue of what we can specify formally in sections 4 and 5.

A.2 Logic Specification

As indicated above there are many logics that can be used in specifications. For our purposes it is interesting to illustrate the logic developed as part of the Alvey FOREST project [[MKJ86]] and known as MAL — standing for Modal Action Logic. The logic is *deontic*, that means it includes notions of *permission* and *obligation*. MAL specifications are concerned with agents and actions so it is possible to specify, for example, that some agent is obliged to carry out some action. Coupled with a temporal capability this gives the ability, in principle, to state that some action must be carried out within a given interval. This is intuitively appealing as it is close to the basic notions of safety in many cases, eg nuclear trips and other shutdown systems. For the sake of simplicity we only consider simple deontic specifications here and do not address the temporal issues.

The available specification logics are very different in form, although all embody the capability of making inferences about (permitted) behaviour from the basis of what has been specified. Thus the following example should be viewed as being illustrative, not representative.

A.2.1 Simple MAL Specifications

MAL is a layered logic, that is it is built up by adding more sophisticated logical frameworks over a basis of first order predicate calculus (the same underlying basis as found in Z). The layers and their uses are:

1. first order predicate logic for specifying the static properties of data and other entities being modelled;
2. a modal logic for expressing the effects of performing operations;
3. a deontic logic for expressing permission and obligation for carrying out actions;
4. action combinators for constructing larger actions from smaller ones;
5. a temporal logic for expressing timing constraints.

Our simple examples will largely be concerned with the first three layers.

Assuming that the reader is now familiar with the simple first order logic concepts through the treatment of Z we can start to explain the second layer, the action logic. In the action logic we can specify axioms of the form:

$$precondition \Rightarrow [action, agent]postcondition$$

This is very similar to the Z concepts except that there is an explicit identification of the agent which engages in some action. The axiom means that, if the precondition holds and the agent carries out the action then the postcondition holds. A benefit of the logic is that we can make deductions about logical possibilities.

Even the simple modal basis allows us to express interesting properties and to deduce relevant facts about sequences of operations. However the deontic component offers much greater expressive power. The two basic constructions are:

$$obl(action, agent)$$
$$per(action, agent)$$

The permission operator, *per*, simply says that the agent may do the action, whereas the obligation operator, *obl*, says that the agent must do the identified action next (although there is no time limit without the temporal component).

Having given this elementary introduction to the basic MAL concepts (excluding operation combination and timing) we can now give a simple example of a MAL specification.

A.2.2 An Example MAL Specification

The specification is structured into sections introducing agents, data types including types for the predicates used in the specifications, and variables which also include definition of the actions which can be undertaken by the agents. There is a specification checking and proof system for MAL and our example is presented in the syntax used by the MAL tools so that we can also illustrate the use of one of the tools. However, it should be stressed that this is only a partial specification intended for pedagogical purposes, not to give a complete problem specification.

The specification is intended to represent the structure of agents and the actions of the agents for a triple modular redundant implementation of a trip system where each of the triplicated channels reads input from six temperature sensors. The output from the three channels goes via a voter to a simplex actuator. In MAL we have chosen to model each of the basic hardware components as an agent — this is the natural approach as the hardware components are the only entities which can engage in actions. It is intended that the example be viewed as defining a computational structure in which the threshold calculations described in Z in the previous sections might be appropriate, ie they might represent the functionality implemented in the channels.

In MAL we first introduce the basic entities for the specifications, ie the agents and data items to be manipulated, together with (types of) predicates which represent the actions engaged in by the agents. The is also identification of other predicates which simply represent properties of the system.

We first introduce four types (sorts in FOREST's terminology) for agents.

AGENT

$Sensor, Channel, Voter, Actuator$

These agents, or rather agent types, represent the four major units in the trip system. The connections between these components will become apparent through the axioms presented earlier.

The data section now introduces two basic data types representing the main data elements that pass between the hardware components (agents) and defines the set of sensors and channels, together with the voter and actuator. We have chosen to have six sensors, *S1-S6*, although this is a rather arbitrary decision (choosing a different number would not have affected the example in a significant way). We also define two predicates representing 'calculations' carried out by the system, viz: *in_limits* and *majority*, but only give their types, in the sense of stating the data over which they are defined, rather than stating their properties in predicate calculus. These predicates are, however, conceptually similar to the operations defined in the Z specification shown above. The three predicates: *available, assessed* and *all_assessed* are necessary to specify data flow through the components of the system and various synchronisation properties. Finally the predicates: *reading, assess, arbitrate, reset* and *closedown* define actions which can be undertaken by the agents.

DATA

$temp, threshold;$
$S1, S2, S3, S4, S5, S6 \rightarrow Sensor;$
$C1, C2, C3 \rightarrow Channel;$
$V \rightarrow Voter;$
$A \rightarrow Actuator;$
$available : Sensor \times temp;$
$assessed : Channel \times threshold;$
$in\_limits : temp \times temp \times temp \times temp \times temp \times temp;$
$signal : threshold;$
$majority : threshold \times threshold \times threshold;$
$all\_assessed :;$
$(Sensor)$ $reading : temp;$
$(Channel)$ $assess : temp \times temp \times temp \times temp \times temp \times temp \times threshold;$
$(Voter)$ $arbitrate : threshold \times threshold \times threshold;$
$(Voter)$ $reset;$
$(Actuator)$ $closedown;$

The predicates are intended to have intuitively obvious interpretations. *Available* indicates the availability of a new reading from the temperature sensor. *Assessed* indicates that a channel has made an assessment and has a threshold value (perhaps indicating that the temperature is outside the allowed limits) available. Both are true when data is available. The predicate *all_assessed* is true when all of the channels have made an assessment, ie when assessed is true for each channel. These predicates are necessary to define the synchronisation and flow of control between the various system components (agents).

In_limits is a predicate representing an evaluation over six temperature values to assess whether or not they are within the specified limits — this is, in effect, the predicate evaluated by each channel. It is true when the temperatures are outside the permitted range. *Signal* is true when an out of range temperature set is signalled from the channel to the voter. *Majority* is the analogue of the predicate *in_limits* evaluated by the voter.

The action *reading* delivers a temperature value from a sensor. *Assess* evaluates a set of six temperature readings and determines whether or not they (according to some averaging calculation) exceed the allowed threshold value — and signal a threshold value if this is the case. *Arbitrate* is a similar function to assess dealing with the threshold signals coming from the three channels and closedown represents the action of shutting down the reactor, eg dropping the rods. Finally *reset* enables the system to start reading temperature values again; it is slightly arbitrary that reset is deemed to be an action of the voter, but this reflects a view that once the voter receives the inputs from the channels the previous values are no longer needed. In practice a rather looser synchronisation may be appropriate.

We now introduce variables which enable us to state the axioms and define the semantics of the operations in which the agent types can engage. The temperature and threshold values with a numerical component represent the outputs from the sensors and from the channels respectively. The identifiers introduced in the data section are also available for use in the axioms defining the system behaviour and clearly refer to parts of the physical system.

VARIABLES
$s : \mathit{Sensor}$,
$c : \mathit{Channel}$,
$t, t1, t2, t3, t4, t5, t6 : \mathit{temp}$,
$l, l1, l2, l3 : \mathit{threshold}$;
END

We can now specify the axioms which define the required behaviour of the system. The basic aim is to show the flow of data and control through the system, culminating in defining when the reactor is closed down. The axioms fall naturally into groups. We first state the axioms in each group then give an interpretation of their meaning:

$/ * \mathit{Axioms\ for\ the\ trip\ system} * /$
$/ * \mathit{Axiom}\ 1 * /$
$\mathit{all\_assessed} \Rightarrow \mathit{obl}(\mathit{reset}, V)$;
$/ * \mathit{Axiom}\ 2 * /$
$[\mathit{reset}, V]!\mathit{all\_assessed}\ \&$
$\quad !\mathit{available}(S1, t1)\ \&\ !\mathit{available}(S2, t2)\ \&\ !\mathit{available}(S3, t3)\ \&$
$\quad !\mathit{available}(S4, t4)\ \&\ !\mathit{available}(S5, t5)\ \&\ !\mathit{available}(S6, t6)\ \&$
$\quad !\mathit{assessed}(C1, l)\ \&\ !\mathit{assessed}(C2, l)\ \&\ !\mathit{assessed}(C3, l)$;
$/ * \mathit{Axiom}\ 3 * /$
$\mathit{FORALL}\ s : \mathit{Sensor}(\mathit{FORALL}\ t : \mathit{temp}(!\mathit{available}(s, t) \Rightarrow \mathit{obl}(\mathit{reading}(t), s)))$;
$/ * \mathit{Axiom}\ 4 * /$
$\mathit{FORALL}\ s : \mathit{Sensor}(\mathit{FORALL}\ t : \mathit{temp}([\mathit{reading}(t), s]\mathit{available}(s, t)))$;
$/ * \mathit{Axiom}\ 5 * /$
$\mathit{FORALL}\ c : \mathit{Channel}([\mathit{assess}(t1, t2, t3, t4, t5, t6, l), c]\mathit{assessed}(c, l))$;
$/ * \mathit{Axiom}\ 6 * /$
$\mathit{all\_assessed} \leftarrow \mathit{FORALL}\ c : \mathit{Channel}(\mathit{assessed}(c, l))$;

The above group of axioms is largely concerned with sequencing of the actions for the system as a whole. *Axiom 1* says that when the *all_assessed* predicate is true, ie when all channels have assessed the input temperatures, the voter is obliged to carry out the *reset* action. *Axiom 2* says that the consequence of carrying out the *reset* action is that no data is available from the sensors and that the *assessed* predicate reflecting the state of the channels is false for each channel (note: that '!' is used for $\neg$).

Axiom 3 says that all the sensors are obliged to read their associated temperatures when their output is not available. *Axiom 4* says that after a sensor has engaged in the reading action the predicate available is true for the

associated datum, indicating that it may be used by three channels carrying out the assessment. *Axiom 5* represents a similar condition to *Axiom 3* for the channels, and *Axiom 6* says that *all_assessed* is true when all the channels have made their assessments.

None of the above axioms are very remarkable — they simply define the 'natural' sequencing of operations through the system. We can now consider the axioms that represent the channel behaviour:

$$
\begin{array}{l}
/* \textit{Axiom } 7 */ \\
\textit{EXISTS } t1 : \textit{temp}(\textit{EXISTS } t2 : \textit{temp}(\textit{EXISTS } t3 : \textit{temp}(\\
\textit{EXISTS } t4 : \textit{temp}(\textit{EXISTS } t5 : \textit{temp}(\textit{EXISTS } t6 : \textit{temp}(\\
\quad \textit{FORALLc} : \textit{Channel}(\textit{EXISTS } l : \textit{threshold}(\\
\quad\quad \textit{available}(S1, t1) \ \& \ \textit{available}(S2, t2) \ \& \ \textit{available}(S3, t3) \ \& \\
\quad\quad \textit{available}(S4, t4) \ \& \ \textit{available}(S5, t5) \ \& \ \textit{available}(S6, t6) \ \& \\
\quad\quad !\textit{assessed}(c, l) \Rightarrow \\
\quad\quad\quad \textit{obl}(\textit{assess}(t1, t2, t3, t4, t5, t6, l), c)))))))));
\end{array}
$$

$$
\begin{array}{l}
/* \textit{Axiom } 8 */ \\
\textit{EXISTS } t1 : \textit{temp}(\textit{EXISTS } t2 : \textit{temp}(\textit{EXISTS } t3 : \textit{temp}(\\
\textit{EXISTS } t4 : \textit{temp}(\textit{EXISTS } t5 : \textit{temp}(\textit{EXISTS } t6 : \textit{temp}(\\
\quad \textit{FORALLc} : \textit{Channel}(\\
\quad\quad \textit{EXISTS } l : \textit{threshold}(\\
\quad\quad\quad !\textit{in\_limits}(t1, t2, t3, t4, t5, t6) \Rightarrow \\
\quad\quad\quad\quad [\textit{assess}(t1, t2, t3, t4, t5, t6, l), c]\textit{signal}(l)))))))));
\end{array}
$$

The axioms here are rather clumsy due to the need to introduce variables for the temperature readings which pass between the sensors and the channels. Unfortunately the MAL checker only allows single variables for each quantified statement, hence the need for the deeply nested existential quantifiers.

Axiom 7 says that when all the sensors have produced data values (temperature readings) then all the channels must assess the values and produce a threshold signal. In practice it would probably be appropriate to specify that the action occurs when a subset of the data is available or after some timeout has occurred. Additionally there may be a need to specify synchronisation between the channels, ie that the channels work in 'lock-step'. For the sake of simplicity we have not addressed such issues.

Axiom 8 states that if the temperature values are not in limits then the threshold value produced by each channel makes the predicate *signal* true, indicating the out of limits temperature values to the voter. It should be noted that we have not said how the predicate *in_limits* is defined so we do not have a full definition of system behaviour.

Finally, we have the axioms defining the operations of the voter and actuator.

$$
\begin{array}{l}
/* \textit{Axiom } 9 */ \\
\textit{EXISTS } l1 : \textit{threshold}(\textit{EXISTS } l2 : \textit{threshold}(\textit{EXISTS } l3 : \textit{threshold} \\
\quad (\textit{assessed}(c1, l1) \ \& \ \textit{assessed}(c2, l2) \ \& \ \textit{assessed}(c3, l3) \\
\quad\quad \Rightarrow \textit{obl}(\textit{arbitrate}(l1, l2, l3), V)))); \\
/* \textit{Axiom } 10 */ \\
\textit{EXISTS } l1 : \textit{threshold}(\textit{EXISTS } l2 : \textit{threshold}(\\
\quad \textit{signal}(l1) \ \& \ \textit{signal}(l2) \ \& \ l1! = l2 \Rightarrow \\
\quad\quad [\textit{arbitrate}(l1, l2, l3), V]\textit{obl}(\textit{closedown}, A)));
\end{array}
$$

Axiom 9 says that the voter is obliged to carry out an arbitration when all the channels have produced values for assessment. Note that we cannot use the predicate *all_assessed* because we wish to identify that the values *l1*, *l2*, and *l3* are actually used as a basis of the arbitration, ie we are identifying the flow of data from the channels to the voter.

Finally, *Axiom 10* says that if any two of the three channels indicate that the temperatures are outside their set limits then the *closedown* action must occur. The specification here is a little artificial as the redundancy and voting is only useful if the channels might 'see' different temperature values (perhaps due to synchronisation problems) or the channels may fail. Again for simplicity in illustrating the use of MAL we have not included such details here.

In principle we should prove that the specification has certain consistency properties, eg that the it does not require one agent to carry out two actions at once (the semantics of obligation is that the agent must do the obliged action next). Also we can derive properties of interest from the specification — for instance it ought to be possible to show that the temperature values going out of range implies that the actuator is obliged to carry out the

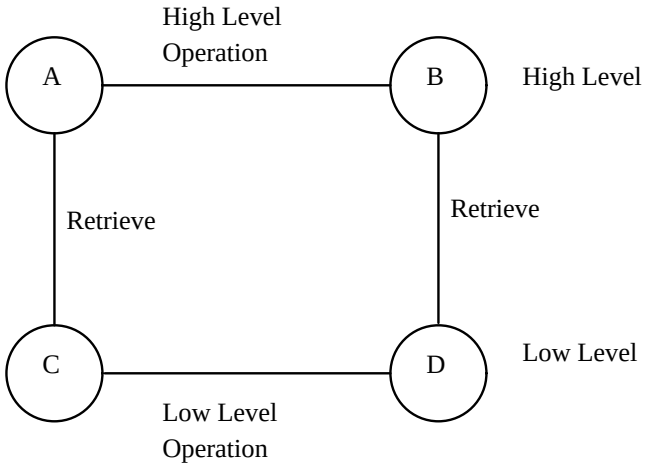

Figure 1: Relationship Between Before and After States

closedown action. The FOREST project has developed some tools, including a proof assistant, for investigating such properties.

With the MAL approach it is worth stressing that we have not only been able to specify required behaviour but, using an animator, we can shown that the system has the expected behaviour in defined circumstances. Thus simulation (and other forms of 'animation') can be an aid to validation of specifications.

A.3 Refinement

Space does not permit us to illustrate a complete refinement here so our intention is to give a more detailed, but not too technical, discussion of the nature of refinement in order to clarify the concept. Our description essentially deals with refinement in the context of model based specification — conceptually similar but technically different approaches are used with other formalisms, eg algebraic specifications.

Refinement covers both guidelines on how to proceed from a high level to a low level specification, and rules for verifying (checking) that this has been done in a consistent manner. It is normal to specify both data which will be stored within a computer system and operations which will modify or transform the data. Thus refinement rules have to deal both with refining data, and with refining operations.

With data objects, the primary requirement for the verification rules is to show that all data which can be unambiguously represented at the high level can similarly be represented at the low level. This is usually referred to as *adequacy*. For example a high level specification may include the concept of a set, and a lower level specification may choose to *implement* the set as a list. It is normal to define a function or relation which maps the values between the two levels. Demonstration of adequacy thus means showing that the relation or function gives an unambiguous mapping between the levels. In our (somewhat simplified) example this amounts to showing that every set can be represented as a list, and *vice versa* for every list that can be generated as the representation of a set.

The function or relation between the levels is given different names in different methods, but it is perhaps most commonly called a *retrieve* function as it can be thought of as retrieving the high level values from their low level representation. In general there will not be a one to one mapping between the levels, and it may be possible to represent *more* values at the low level than at the high level. For example integers in the range 1 to 10 in a specification might be represented by full (machine processable) integers in a program or lower level specification. Further values at the high level may be represented in more than one way at the low level — indeed this is the case in our simple set example.

With functions/operations the requirement is to show that the operations at each level *do the same thing* — albeit after allowing for mapping between the data objects at each level. This is usually referred to as *satisfaction* . The concept of satisfaction can most readily be illustrated by considering a diagram relating states before and after an operation.

Imagine starting with a low level value, *C*, and mapping it to a high level value *A* before applying the high level operation to arrive at value *B*. It would also be possible to carry out the low level operation first, then to map from *D* to *B*. Satisfaction requires that each route leads to the establishment of the same value at *B*.

There are in fact many different definitions of refinement, although many of them are conceptually similar (but

not identical) to the form illustrated above.

In practice refinement rules typically incorporate a set of *proof obligations* which are criteria which must be met if a refinement is to be valid — more strictly the obligations are theorems which have to be proven to show adequacy and satisfaction.

There are, in general, differences in amount of detail between two levels of specification, so verifying the proof obligations cannot show that the specifications are equivalent — merely that they are non-contradictory. This is essentially the point we were making earlier when we were drawing the distinction between the pairs: verification/validation and synthetic/analytic reasoning. More significantly there is a considerable amount of freedom in defining a set of refinement rules, eg in the way they treat non-determinism, and this has led to many sets of refinement rules being developed, each with its own strengths and weaknesses. This does not mean that some techniques are right, and that others are wrong, rather that they have different areas of applicability.

We have illustrated the concepts of refinement in the context of model-oriented specification. With the other approaches to formal specification the technical details of refinement are different from that of model-oriented specification, but the spirit is the same — verifying that we are adding detail, or otherwise enriching specifications, in a manner which is consistent with the initial specification.

Our brief discussion has also focussed largely on the verification aspects of refinement, and not on the guidelines for proceeding from a high level to a low level specification. Typically these guidelines will (or should) cover issues of functional decomposition, and also consider non-functional properties of systems. That is, the guidelines should recognise that non-functional issues such as performance, reliability, and so on, can **drive** the refinement process. Unfortunately current refinement approaches do not deal adequately with such issues so, for example, there are no refinement rules which deal adequately with fault tolerance — an approach would need to show that the fault models plus fault recovery mechanisms at one level 'satisfied' the fault models at the next higher level. This remains an area of research.

A.4 Summary and Comparison of Approaches

There are a wide variety of types of formal methods, each with different characteristics which means that generalisations about formal methods may be more misleading than helpful. It is also rather difficult to appreciate what the methods are like in use from simple definitions and descriptions — by way of analogy, consider how difficult it is to appreciate the utility of a programming language without trying it out on a few problems. This is why we have taken the trouble to give fairly extensive examples of three rather different types of formal method. We are now in a position to make some comparisons, although we steer clear of value judgements regarding utility as this is the province of the next section.

First, we can now see clearly that the two methods enable us to do quite different things. Z enabled us to give quite detailed specifications of the required behaviour of the actions to be carried out in the system, but was rather poor at modelling communication and has no way of representing concurrency. In contrast MAL is much clearer about system structure, including potential for parallel execution, and communication although they it is relatively weak at defining functionality. MAL allows us to make statements about timing behaviour, whereas Z does not.

Some of the above differences are partially a reflection of the way in which we have used the notations. For example it is possible to specify timing in Z [[CCM90]] but we believe we have accurately characterised the 'natural' way to use the core specification languages in each case. Thus we must conclude that different methods have quite different expressive powers.

Second, we believe that it is quite difficult to use the techniques outside their natural domains. This does not mean to say it is impossible, as we indicated above it is possible to extend the techniques to deal with additional properties of systems but it is not entirely straightforward — for example adding a deontic component to Z would be quite difficult, especially when it came to defining the semantics for the extended notation. However since the methods illustrate different facets of systems then they can be used together — assuming we can map adequately between the notations. Thus we believe that it is both possible and beneficial to use an eclectic approach to specification, although this is rarely, if ever, done in practice.

Third, the mathematics, although valuable for its precision, does not stand on its own. In the example it was essential to use prose to define what it was that the specifications were meant to relate to — 'in the real world'. It is always necessary to support formal specifications with prose and, without this, we have no way of knowing what the specifications mean. More technically we know what they mean in terms of the underlying logic, but we don't know what they mean in relation to the systems we hope to build. This is a general property of formal approaches, not just a characteristic of our examples, but one which we hope is adequately borne out by the examples.

Fourth, there is considerable difference in the conciseness or verbosity of the notations. Again this is partly an effect of the examples chosen and the way the problems have been addressed but. This is important as conciseness

influences intelligibility, although there is not a simple relation. Extremely terse and extremely verbose notations may be equally hard to read and, ideally, we require concise notations so we do not have much to read, but which are still as easy to read as ordinary english prose. This is, of course, a difficult compromise to achieve — and we will leave the reader to draw his own conclusions about which, if any, of the three notations used above satisfy this requirement.

Finally, it is important to stress the point that the methods are genuinely different in their capabilities and any generalisations about formal methods (other than this one!) may be quite misleading and inappropriate to some particular class of method.