THE ACHIEVEMENT AND ASSESSMENT OF SAFETY IN SYSTEMS CONTAINING SOFTWARE

A. BALL SRD, UKAEA
M. H. BUTTERFIELD AEEW, UKAEA
C. J. DALE SRD, UKAEA

ABSTRACT

In order to establish confidence in the safe operation of a reactor protection system, there is a need to establish, as far as it is possible, that

(i) the algorithms used are correct
(ii) the system is a correct implementation of the algorithms
and (iii) the hardware is sufficiently reliable.

This paper concentrates principally on the second of these, as it applies to the software aspect of the more accurate and complex trip functions to be performed by modern reactor protection systems.

In order to engineer safety into software, there is a need to use a development strategy which will stand a high chance of achieving a correct implementation of the trip algorithms. This paper describes three broad methodologies by which it is possible to enhance the integrity of software: fault avoidance, fault tolerance and fault removal. Fault avoidance is concerned with making the software as fault free as possible by appropriate choice of specification, design and implementation methods. A fault tolerant strategy may be advisable in many safety critical applications, in order to guard against residual faults present in the software of the installed system. Fault detection and removal techniques are used to remove as many faults as possible of those introduced during software development.

The paper also discusses safety and reliability assessment as it applies to software, outlining the various approaches available. Finally, there is an outline of a research project underway in the UKAEA which is intended to assess methods for developing and testing safety and protection systems involving software.

1. INTRODUCTION AND STATEMENT OF THE PROBLEM

In order to ensure that the risks from nuclear plant compare favourably with other risks of everyday activities, their protection systems are designed to satisfy quantifiable reliability requirements. Indeed targets are laid down in the UK by the CEGB (ref. 1): these state, inter alia, that

(a) The total frequency of all accidents leading to a large uncontrolled release of radioactivity to the environment resulting from some or all of the protective systems and barriers being breached or failed should be less than 10^{-6} per reactor year.

(b) The frequency of any single accident leading to an uncontrolled release should be less than 10^{-7} per reactor year.

The accident frequencies stated in (a) and (b) above are defined as being the product of the initiating fault frequency and the probability of failure of the safety system to control the accident.

407

Thus, from the above targets and a knowledge of the frequency of the initiating events it is possible to arrive at design targets for the safety systems. However, ref. 1 stipulates that due to common-mode failure (CMF), the failure probability which can be claimed for a redundant but non-diverse system should not be better than 10^{-5} per demand. Even this level of reliability requires justification: higher levels of reliability cannot be claimed without applying effective defences against CMF (ref. 2). For hardwired systems this must include the use of fail-safe techniques, diversity of design, diversity of parameter measurement etc. Similar considerations also apply to computer-based safety systems, but must also embrace the software.

This paper is concerned with methods of system design and testing, with emphasis on the software aspect. Our course of research is also presented; this is intended to compare the forms of system architecture and design methods which minimise the likelihood of software faults in reactor protection systems. It is also intended to evaluate methods of testing both for assessment of system reliability and for detection and removal of residual faults.

For the purposes of this paper, it is necessary to define certain terms on which there is no general agreement as to definition. The term error will be used in two senses: it may refer to an incorrect action by a human being, or it may refer to an incorrect state of computation during processing. A fault in the software is the result of a human error, and will lead to a processing error when certain input values are encountered. A failure occurs at the system level and may be caused by a processing error if the system is unable to recover from that processing error by fault tolerant means.

2. STATE OF THE ART

Whereas the very large majority of nuclear reactors in the world rely on hardwired protection systems, there are strong incentives to adopt computer-based protection systems either wholly or in part. These are

(a) In cases where very large numbers of sensor signals need to be economically processed e.g., in sub-assembly temperature protection systems.

(b) In cases where sensor signals need to be processed by highly complex algorithms e.g., for DNB protection with scope for more efficient use of the plant i.e., increased output.

With regard to (a) examples are the TRTC minicomputer-based system in the French fast reactors Phenix and Super-Phenix (refs. 3 and 4) and the UK microcomputer-based ISAT system (refs. 5 and 6).

For (b) the French microcomputer-based safety system SPIN (refs. 7 and 8) has been introduced into the PALUEL reactors and Westinghouse are offering WIPS, also a microcomputer-based safety system, for the proposed PWR for Sizewell 'B' in the UK (reference 9).

The Project on Diverse Software (PODS, ref. 10) constituted an international exercise within the Halden project in which teams from UK (CEGB), Finland (VTT) and Norway (Halden) implemented a representative protection system, the structure of which was defined by the Safety and Reliability Directorate (SRD) of the UKAEA. Each system was subjected to rigorous acceptance tests and corrected so that no faults were apparent at that stage. The three were then compared 'back-to-back' at Halden. This process revealed further faults which are currently being analysed.

Many researchers in this field emphasise the need to validate the software specification (e.g., refs. 21, 22 and 25), and this is now receiving particular attention. A particularly difficult area is that of relating measures of testing thoroughness (e.g., ref. 11) to measures of operational reliability. It is possible that methods could be developed to estimate reliability from comparisons of the potential faults in the software and those removed by testing.

The philosophy of choice of test spectrum is discussed in Sections 5 and 6. In order to assess system reliability various choices exist representing operating conditions or alternatively the way in which the trip condition is approached. Obviously a safety protection system should not retain any known faults and correcting them as they arise necessitates repeating all tests since it is not possible to be certain that the correction process has not created a fault elsewhere.

3. FAULT AVOIDANCE

Fault avoidance is concerned with making software as fault free as possible by appropriate choice of development methods. It is essential that this principle be applied at all stages of development, and not just during the programming itself, because a large proportion of faults can be expected to have their origin early in the life cycle. For example, Potier et al (ref. 12) report a study in which more than 60% of all faults originated prior to the implementation (or coding) stage.

Since software faults arise from human errors, the control of factors which affect the occurrence of human errors will reduce the fault content of software. According to Rzevski (ref. 13), there are five classes of factors which affect the occurrence of human errors: these are discussed below. The choice of development methods which assist control of these factors can be expected to produce software which is relatively fault free.

System Complexity. The inability of humans to deal with over-complex entities without committing numerous mistakes is well documented, for instance by Miller (ref. 14). There is a number of software design methodologies which are all aimed at containing the complexity of software. Many of these organise the software in a hierarchy of reasonably independent software modules (e.g., Yourdon and Constantine (ref. 15), Myers (ref. 16)). An alternative approach is due to Jackson (ref. 17). There is thus a range of methods available for control of system complexity, which are most readily applied at the design stage.

Task Complexity. Experience shows that the frequency of human errors increases if tasks are too simple (and thus boring), or too complex. This topic has had very little coverage in the literature, and tends to receive insufficient attention from software development managers. A useful rule of thumb which can be applied at the implementation stage is that no individual should be expected to deal with more than one module at a time. In general, tasks should be clearly defined in terms of inputs, outputs and activities to be performed.

Resources. The quality and characteristics of the resources made available for system development can have a significant effect on the occurrence of human errors. Resources include languages, guidelines, manuals, standards, computer aids and tools applicable to all phases of software development. There is a very large number of tools and other resources available, and it is an important managerial task to select the best and most appropriate tools for the job in hand.

The language which is chosen for a particular application may affect considerably the avoidance of faults. It should be borne in mind that it is now possible to express specifications and designs, as well as programs, in formal languages.

Guidelines and standards are very powerful aids to the organisation and management of software production. Company standards are often preferable to national or international ones, because they can be tailored to the particular organisation and can more readily be modified and updated.

There has in recent years been an explosion of activity in the field of computer aids and tools for software production. The problems here for managers are posed not only by the sheer number of tools and the difficulty of deciding which are the appropriate ones, but also by the extreme difficulty of assembling an integrated tool set to deal with the whole of the software development. It is to be hoped that future developments will alleviate these problems.

Human Factors. Inevitably, the occurrence of human errors is influenced by the characteristics of the personnel involved in software development. While there is a large body of knowledge on the relationship between human factors and the frequency of errors by operators of machinery, the effects of human factors on design errors such as those which lead to software faults have not been studied extensively. There is evidence (ref. 13), however, that the occurrence of human errors is minimised if software production personnel are equipped with knowledge and skills related to systematic design methods, and if their attitude is egoless, thorough, and self disciplined. It is also an important skill to be able to identify simple solutions to complex problems. Thus the control of human factors is by selection and training of personnel.

Environmental Factors. Reliability of software can be affected by the physical, social and psychological aspects of the environment in which systems are created. Examples of possible influencing factors are noise level, team morale, and criteria for promotion. An important reference in this area is the book by Weinberg (ref. 18).

The choice of development methods in the way outlined is not enough in itself. It is also necessary to implement appropriate quality assurance and project control activities, and to ensure that the project is thoroughly planned at the outset, including such aspects as the testing philosophy to be adopted. Only in this way can there be any assurance that the introduction of faults into the software during its development has been kept to a minimum.

There is currently no development method which can be guaranteed to produce fault free software; nor is there any assessment method which can guarantee that a given piece of software is fault free. A good deal of research is going on in these areas, but for the moment it is essential to test software, both to remove faults and to gain confidence in the level of the reliability. In safety-related and other areas of application where reliability is of great importance, it may be the case that the risk of residual faults in a delivered piece of software is too high. If this is the case, it is possible to implement fault tolerant software, in order to reduce the risk that these residual faults will lead to an unsafe situation.

4. FAULT TOLERANCE

This section reviews methods for promoting high reliability in systems that incorporate software whilst acknowledging the limited reliability of individual programs.

Although it is possible for any given program to be free from faults, establishing this as a fact is as yet not possible. Therefore, it is worthwhile to examine ways of protecting systems against the effects of faults that might exist.

Redundant Hardware with Common Software. Here the individual elements or channels of a system are replicated and system behaviour decided by majority voting on individual channel behaviour. In this case the individual channels each contain identical software, so that if each piece of software receives the same input, all channels will give the same output. In particular, if a software fault leads to failure in one channel, then all channels will fail simultaneously. Any argument for software fault tolerance due to redundant hardware therefore depends upon the various pieces of software receiving different inputs, and the degree of reassurance which can be obtained in this way is very limited, as will be shown.

With this arrangement it is possible to argue that provided redundant sensors are used, the input signals representing each parameter will almost certainly differ for each channel. In fact, for a series of 12-bit digital input signals corresponding to analogue parameters, the least significant 3 or even 4 bits are likely to differ due to differences in sensor accuracy, resolution and operating characteristics, and for all intents and purposes can be regarded as random. Thus, even for a relatively simple system with only a few such measured values, there will be a wide variation in program inputs for the same parameter values. Hence, it is argued, any software fault that causes one channel to fail is very unlikely to affect the other channels at the same time. If such a fault did exist, so the argument runs, its effects would be quite gross, so that any reasonable test procedure should bring it to light.

It is considered that this argument, although superficially appealing, is untenable. It pre-supposes that software faults are sensitive to the absolute bit patterns of inputs, whereas very many faults are likely to cause failures for wide ranges of inputs or groups of inputs. The claim that faults with such gross effects can be expected to reveal themselves early during testing cannot be relied upon since with even a simple system that uses only 5 x 12-bit inputs, the total number of combinations is of the order of 10^{18}, so a fault that appears gross by causing failures for a very wide range of input combinations can still be a relatively obscure one, which may not be revealed by testing.

In general, it is fair to say that this kind of fault tolerance protects against only hardware failures.

Software Diversity. Software diversity, also known as a n-version programming (ref. 19), would appear to present a way of achieving truly independent software. With a 3-channel protection system employing different programs in each channel and a 2-out-of-3 majority voting arrangement, software faults would seem unlikely to affect 2 or more programs for the same input conditions since each program would contain different faults. However, there are limitations to this reasoning associated with the degree of diversity that can be achieved in practice. There must be some common ground, which includes at least the task specification. If this is a very detailed document, defining

411

the task precisely, then it is likely to contain faults. The person or team writing the specification may know exactly what is wanted, but in expressing these demands may create ambiguities, inconsistencies, or simply write down the wrong thing, or they may have an incorrect view of what the software ought to do. Even with no human errors in these processes, typographical errors may well occur and slip through uncorrected, especially where numerical data are concerned. Faults created in any of these ways are likely to propagate through all the different pieces of software, and for these faults negate the apparent benefits of diversity.

There are other common factors, even with independent programming teams methods of thinking and working may be very similar, especially for highly skilled programmers, so the algorithms embodied in the finished programs may resemble each other quite closely. This, together with the fact that error proneness itself is a particularly human characteristic, with susceptibilities to fault generating mechanisms common to all, degrades somewhat the hoped for independence.

Software diversity represents a powerful means of protecting systems against the dangerous effects of software faults, but to achieve it to an acceptable degree very careful controls must be applied. Our experience is that special care must be taken at the task specification stage. If more than one person is familiar with the requirements, there may be merit in each one producing a preliminary version, without consultation with the others, and defining only <u>what</u> the software is to do, but not <u>how</u> it should do it. Then, by reviewing each others' work, a consensus view may be arrived at, and a final version developed. It is essential to commence the development of the software itself with a single specification, common to each piece of software. Any diversity of specification would be likely to lead to a situation where the various programs do different jobs and are incompatible.

Each separate team should apply the currently accepted techniques for generating reliable software, notably effective quality assurance procedures and proper documentation, structured design, internal design reviews and development testing.

It is worth noting that the number of diverse programs need not be the same as the number of channels. For example, two or more diverse programs may be implemented within a single channel, with voting taking place to determine what the channel output should be. In this way it is possible to make all channels identical, and still take advantage of both hardware redundancy (between channels) and software diversity (within channels). This arrangement may lead to cost savings, since procurement and spares handling procedures may be simpler.

<u>Other Software Design Features</u>. Several other features can be incorporated within individual programs to help in recognising the effects of processing errors, which are caused by interactions between internal data and faults in the program, and thereby make the software fail safe. Such features include functional diversity (also known as the use of validatory functions), where a program performs cross-checks on internal algorithms by diverse means, perhaps to a lower degree of accuracy as a way of detecting gross processing errors. The detection of a processing error must be followed by a suitable error recovery process, the intention of which is to return the system to an error free state. In a multi-channel system using diverse software, error recovery may consist simply of awaiting the next input signal. In a single channel system without diverse software, error recovery would sometimes necessitate a reactor shutdown in order to remain fail safe. In any

412

case, all such events should be automatically logged for investigation. A particularly sophisticated form of internal functional diversity is the recovery block (ref. 20).

In order to contain the effects of processing errors, each process, task, sub-task, module or sub-system of the software should validate its inputs, before commencing processing, and then at completion establish criteria showing that its outputs are correct, possibly by use of functional diversity within the process concerned. There should also be protection against the software looping, either indefinitely or for a large number of times, due to a software fault. This is best achieved by ensuring that at specific time intervals there must be a specific output.

Another worthwhile technique is software self-testing, which again relies on some form of diverse software incorporated in a control program which continually exercises algorithms in the program under test in order to check their behaviour against that expected.

5. FAULT REMOVAL

As has already been pointed out, fault avoidance techniques used in isolation cannot be guaranteed to produce sufficiently fault free software. Thus, whether or not a strategy of software fault tolerance has been adopted, it is essential to carry out fault removal activities in addition. Historically, fault removal consisted of testing the software in a relatively ad hoc manner. More recently, it has been recognised that the longer a fault remains, the more expensive it is to eliminate - Boehm (ref. 21) reports that faults are typically 100 times more expensive to correct in the maintenance phase on large projects than in the requirements phase (ignoring the consequential costs which may be associated with an operational failure). The result of this observation is that fault removal activities are now applied at all stages of the software life cycle, in addition to testing towards the end of development. These activities include such things as design walk throughs and peer reviews, as well as inspections, which are described below.

Inspections. Substantial improvements in programming quality and productivity can be obtained through the use of formal inspections of design and of code. The chief objective of the inspection process is to find faults. For the sake of clarity, the inspection process will be outlined as it would be applied between design and implementation. As Fagan (ref. 22) points out, inspections can be applied throughout the development cycle of software.

The design inspection team should be led by a moderator, and include the person who produced the design, the person who will implement that design, and the person who will eventually test the program produced. Preliminary to the inspection itself there is an overview meeting at which the designer describes the overall area being addressed and the specific area he has designed in detail. This is followed by individual preparation so that each member of the team has an understanding of the design prior to the inspection meeting. The inspection meeting itself consists of the implementor describing how he will code the design. Other members of the team raise questions during the implementor's discourse, which are pursued to the point of a fault being identified. After the inspection meeting all faults noted are removed by the designer. It is the moderator's responsibility to ensure that all problems have been resolved.

The details of the above description will inevitably vary for application of inspections to other stages of the software life cycle, but the underlying principle of organising a formal detailed discussion between interested parties with the objective of identifying faults is applicable throughout the software life-cycle.

Testing for fault removal. Software testing comprises two distinct activities. The first is fault detection, concerned with finding software faults to allow their removal, and the second is reliability testing, to demonstrate an acceptably low failure frequency or unavailability during use. These processes are not aspects of the same thing as is commonly thought, but are separate in both purpose and design. The efficiency of the fault detection process is related to its ability to find the faults that are present, its aim is to make the software fail. The function of reliability testing on the other hand is to provide an indication of the reliability that can be expected of the software during its operational life. Any software is likely to contain residual faults even after the most stringent production processes and penetrating fault detection testing, therefore, for safety critical applications, an assurance is required, quantified if possible, that the probability or frequency of system failure due to their presence is acceptably low. Reliability testing is intended to provide this assurance, and will be discussed later in this paper.

Development testing comprises those test procedures applied by the development team themselves to their own programs. If a properly structured design methodology has been used, then the individual modules should be testable individually. A well-designed module will incorporate features to facilitate its testing, so that as many (possibly all) combinations of inputs as practicable can be exercised to verify correct functioning. There is direct merit in simplifying module tasks as much as possible since the simpler the task the more comprehensive can the module test be made, and the less chance there is of carrying faults forward to later development stages.

Attention should be given to the ranges of input variables that the modules are to deal with. It should be recognised that processing errors external to the module may cause an input range boundary to be violated, so the module test procedures should deliberately incorporate such values to check for an appropriate response. The process demonstrates the robustness or otherwise of the modules - their ability to cope safely with external error effects.

Testing can be administered by use of a driver or a test harness, as appropriate. A driver is a piece of software that permits the setting up of input conditions, calls the module under test as a subroutine, and accepts the outputs for storage, display, or further analysis as appropriate. A test harness is a software system which enables a module, sub-system or complete system to be executed in an environment different from its design environment (e.g., to execute microprocessor software on a mainframe). The test harness will provide an equivalent interface to external peripherals and other software components, and may well have other functions relating to the recording and analysis of the testing.

For simple modules, it is often sufficient for the driver merely to display outputs in numeric or graphic form, with direct inspection for correctness. For more complex or higher level modules a more carefully designed test procedure may be required, with simulation of the module task by part of the test harness to allow comparison of module response with that expected. In such cases, the module simulation software incorporated in the test harness should be written by a different person than the one who wrote the module itself, so as to provide diversity in the test function.

At some stage the software development will be deemed complete by the development team, and ready for a more formal testing strategy as a complete program. At this stage sets of input test data will be selected for the program to deal with, and the response checked against that expected. A test harness can be used to administer the tests, and possibly to execute the same task as the program in a diverse manner, comparing and reporting on the two sets of outputs. If diverse programs are to be employed, then these can usefully be tested at the same time with the same inputs. Comparison of output can reveal all types of fault except those due to common mode error effects present in all the diverse systems.

Black box and glass box testing are broad alternative testing strategies which differ in the way in which input test data are selected. Black box testing treats the program as just that - a black box of unknown content. Inputs are selected on some random basis, or deliberately, to exercise specific features of the test specification. The strength of this form of testing lies in the fact that it is independent of the way the task is implemented. The black box need not in theory even contain software. Providing the task is implemented correctly, then the contents of the box are of no concern.

Glass box testing acknowledges not only the fact that the box contains software but also takes note of its structure. Input data sets are chosen to exercise the various paths through the program. It is rarely practicable to exercise every possible unique path from input to output since there are likely to be far too many, but with a well-designed procedure it is possible to exercise a sensible subset. The lowest level is to ensure that every statement is executed at least once, the next level is to ensure that every branch is executed at least once. Proceeding in this systematic way ensures substantial classes of fault are eliminated (ref. 23).

Manual black box testing strategies use input data sets which are deliberately chosen to check specific task features. Functional tests are aimed at producing all the various responses, not necessarily all combinations of responses since these would usually be excessive, but all the separate forms of response for as many different circumstances as are directly relevant, to show that all the intended transfer functions have been properly incorporated. Boundary value tests check operation at and around the input conditions that cause changes of state of outputs, and at input boundaries where these are significant, since these areas are especially prone to error. Invalid data checks deliberately supply out-of-range values for inputs to check the robustness of the program. Simulation tests supply input sequences that correspond to expected input behaviour during normal operation. Other forms of manual tests can be devised to test special features, for example the timing of certain operations may be an important aspect of proper functioning.

Random testing strategies are also of black box type, but whereas the manual tests are selected by inspection and therefore severely restricted in number, random testing is automatable and can supply large quantities of test cases. There are various forms of random strategies, perhaps the most obvious being full range uniform randomness of all inputs, where all the input values are allowed to vary independently over their full range, both valid and invalid. In practice this is usually very inefficient since generally whole batches of input combinations cause similar operation. For example, if one input is a 'Power On' signal which (say) sets all outputs to some pre-determined state regardless of all other input conditions, then half of the (random) tests can be expected to exercise the software in precisely the same manner, thus greatly reducing the value of a large proportion of the tests. A better strategy in many cases is to set certain inputs to fixed values while allowing others to vary at random for a set number of tests, after which the fixed inputs are set to new fixed values and the tests continued.

Equivalence partitioning is a more useful method of random testing which avoids the problem illustrated above. Here the full domain (valid and invalid) of each input is split into a finite number of ranges, all input values in a particular range being judged equivalent in testing the program. These ranges will not generally be of equal length, some will contain many values, when say all values up to some high level trip point are expected to have the same effect, whereas others will have few values, for instance between an alarm and a trip, when these lie relatively close together. Then equal weighting is given to each of the partitions in selecting random values, so the testing can be made more efficient.

Boundary value testing loads the selection of test data to around the edges of both the input and output equivalence partitions (not necessarily at the same time). Special consideration is needed in defining the loading of inputs to produce outputs around their equivalence partitions, but experience shows that this technique is an especially powerful one for revealing faults in software.

If it is desired to test equally thoroughly over a particular range of values, for instance when carrying out equivalence partitioning, then it is appropriate to generate random test values from a uniform distribution. The Gaussian distribution, on the other hand, is appropriate for boundary value testing, since it leads to a clustering of values around a given point.

Whatever distribution is used, it is important to record either the complete set of test data, or sufficient information to enable the set to be regenerated, as well as the manual test input values. This will enable the repetition of any test sequence, which will be necessary after each fault repair session since further faults are often introduced during repair, and it is important to know that program performance (as monitored by its response to the above tests) has not been degraded.

Path testing uses the glass box technique to exercise paths and sub-paths to some pre-determined level. The task is well suited to computer implementation and programs are available to handle it. One such test program (ref. 11) takes sets of test data and runs the program under supervision. The test program monitors the effectiveness of the test in terms of the paths exercised, and reports accordingly. Different sets of data are chosen, often by devising special cases, to exercise all statements, then all branches, then sub-branches (all secondary branches after some primary branch) to whatever level is desired.

The test program is used in conjunction with a static analysis program that is applied first. This performs a detailed scrutiny of the statements comprising the source program, both to highlight and report on defects in its structure, and to format the code in a special way that makes it accessible to the test program described above.

4. ASSESSMENT

Traditional methods for safety and reliability assessment are in many cases not readily applicable to software, despite their being perfectly sound for hardware assessment. There are two main aspects to the safety assessment of software: assessment of the process of creating the software and assessment of the software product itself. Ideally, safety assessment could be done in

416

terms of the product alone, by thorough testing of the software (making use of the software structure, for example, by ensuring that all paths are tested) to establish confidence in the correctness of its implementation. This is not in general practicable due to the complexity of the software, the enormous variety of possible input data, and the stringent requirements imposed by the need for safety. Even if it were possible to demonstrate software integrity, and so carry out a perfectly sound assessment of the software, it would not be desirable to do this alone: an assessment of the process, aimed at ensuring that the development of the software is proceeding successfully towards a safe and reliable product, can highlight problem areas and bring about their resolution much earlier than the retrospective assessment of the product alone. It is therefore necessary to carry out assessments of both process and product, and it must be recognised that neither of these assessments is capable of guaranteeing safe operation.

Assessment of the process. There is currently no available technique for quantitatively assessing software reliability based on a knowledge of the process alone - indeed, it is only recently that qualitative methods have become available. Daniels (ref. 24) describes an assessment methodology based upon a checklist of questions relating to various activities throughout the software development cycle. The purpose of the checklists is to provide a stimulus to critical appraisal of all aspects of the system rather than to lay down specific requirements. The approach can be used to assess the quality of a particular software development process from the point of view of the likely reliability of the final product, and can readily be integrated with software quality assurance activities.

The usefulness of assessing the process is twofold. Firstly, the assessment can be applied very early in the development process, and can thus be used to guide the development and to give an early warning of problem areas. Secondly, since product assessment is imperfect (for reasons discussed below), the process assessment can be useful in improving judgement as to whether a given product has reliability of an adequate level; this is especially true of safety systems, since the high reliability requirements often associated with such systems are very difficult to demonstrate by testing alone.

Assessment of the product. Traditionally, assessment of the reliability of software is carried out by way of testing, and it is this aspect which will be concentrated on below. It is worth pointing out, however, that there is an increasing number of software tools which can be used to assess the quality of the product, including such things as code analysers (e.g., ref. 11). Such tools are probably most useful for the task of fault removal, but their potential in assisting product assessment should not be ignored. In addition, a great deal of research is currently going on into formal methods of software development. An example of this work is the idea of program proofs, where a mathematical proof is carried out (manually or semi-automatically) to demonstrate that a given piece of software is correct with respect to its specification. Probably the most highly developed formal method is the Vienna Development Method (VDM), (ref. 25), originally developed by IBM. VDM employs a formal mathematical notation, and the basic procedure is to write a rigorous specification of the software which is then developed into a derived correct program. The final stage is to examine the performance of this program and improve its efficiency where necessary.

These methods have not yet matured to the extent that they can be advocated for general use, but the potential benefits of their use in the future are enormous, both from the point of view of fault removal and reliability and safety assessment. However rapidly these methods are developed, it is difficult to foresee a time when testing will not be utilised as a final demonstration of software reliability. It is only when the testing stage is reached that the performance of the software can be compared with the requirements: formal methods can only hope to prove correctness with respect to a specification, and that specification may itself be an inaccurate reflection of the requirements. The remainder of this section concentrates on the use of testing as a means of assessing reliability.

Reliability testing. As has been pointed out, this procedure is not primarily intended to find faults, but rather to give an assurance that during use the failure frequency or system unavailability due to residual faults is acceptably low. To provide this interpretation an operating environment that simulates the input conditions expected during use is required. If several different operating modes are to be experienced, then separate tests should be devised to simulate each, to provide reliability measures for each mode. For example, in the case of a reactor protection system, the plant can be expected to operate for the vast majority of the time at steady full or near full output when trip margins are large. The safety role of the protection system only comes into play when trip conditions arise, so that testing the software for safety should concentrate on the inputs to be expected when trip conditions arise. Such a system has another implicit requirement, which is not to trip when trip conditions do not arise (i.e., to reduce false alarms). Reliability testing for this requirement should concentrate on those inputs to be expected during normal operation. It is worth emphasising that a test plan which simply mimics the overall operating conditions of the plant would in this case provide almost no safety assurance.

Any failure that is observed during reliability testing should be traced back to the underlying fault and this repaired, unless it can be shown beyond doubt that no dangerous effects can result from the presence of the fault. Any fault so found and repaired will necessitate a repeat of all tests (fault detection and reliability) up to the point of failure, to discover any new faults introduced by the repair. This is not an onerous task if the possible need for it has been anticipated and all test data has been recorded in such a way that the testing can be repeated automatically.

Choice of test data which simulates the conditions expected to be experienced during the operational life of the program implies a detailed knowledge of plant behaviour. For a complex plant such as a reactor it is difficult, if not impossible, to select the appropriate distribution of input conditions manually. Limited groups of inputs for certain expected modes of operation will be identifiable but these will be far from comprehensive.

What is recommended is to engineer a plant model in software (if a simulator already exists then this might be adaptable) incorporating as many behavioural interactions as is practicable, and use this to generate test data. Then all expected normal, abnormal and dangerous circumstances can be simulated, and the corresponding test data applied to the program either immediately or by preference via an intermediate data storage file (to permit repeat testing after error repair). The better the plant model, then the more representative will be the test data generated. The relationship between infidelity in the plant model and degradation in value of the test data is not known; intuitively it would seem that providing the model did not incorporate gross inaccuracies or errors, then the test results should retain an acceptable level of validity for the real plant.

If diverse software is to be used, then there is no reason why the reliability testing process should not continue through life, with some external system to recognise significant output disparities and record the corresponding plant conditions.

Reliability quantification is often necessary in order to ascertain whether specified reliability requirements have been met.

If a reliability figure for software is demanded then it is vitally important that the figure should be valid. Indeed, this is stating the obvious, but it is very easy to generate a number: any test will yield some or no faults in a given number of trials, so by applying whatever confidence factor seems reasonable, a fault frequency or unavailability can be derived. The danger is that when a number becomes associated with a piece of software it may unduly influence the decision as to whether or not the software is adequate for use, when in fact the number in question may be of low accuracy and based upon assumptions which are unsound. Therefore, the greatest caution is called for in calculating and using such a number.

The basis of quantification should be the results of a reliability testing procedure which applies data from the operational input domain, not an error detection procedure which uses quite different input domains. If any results other than of reliability testing are used, then the derived figure will be valid only for the particular input domain used during the tests, which might well bear no resemblance at all to the operational input domain.

The bare data from which any reliability figure is derived may well take the form "k failures observed in N trials", the value N being the number of trials that exercise the safety function out of a large sample of input conditions from the operational input domain. It is of course the case that for a safety critical application such as a protection system each of the k failures must be investigated and the corresponding fault removed. However, for the purposes of calculating a reliability figure with an appropriate level of statistical confidence, the k failures must be retained. This is because a further N random trials may well reveal further faults, and the "k failures observed in N trials" may still be the performance level to be expected from those areas of the input domain which were not covered by the original N trials. In this way a suitably pessimistic figure will be generated for the expected number of failures in N demands on the safety function of the program.

There will be occasions when the figure obtained by using "k failures observed in N trials" is inadequate, but it is felt that the removal of the faults leading to those failures has improved the software reliability across the whole input domain. The only sound way of demonstrating this statistically is to carry out a further set of tests to exercise the safety function of the (corrected) software. It may then be possible to base the assessment on data of the form "No failures observed in M trials", where M is the number of inputs used in the new testing, and a much higher level of reliability can be claimed. On the other hand, the additional testing may simply confirm fears that there are further faults which were not exposed by the original reliability demonstration test comprising N input sets.

From the figure generated as above can be deduced an unavailability figure per demand, which is the pertinent reliability parameter for a protection system. For safety critical applications other than protection, other more suitable figures can be derived such as failure frequency or reliability in the mathematical sense - the probability of functioning correctly for a specific period of time under specific conditions.

The figure so generated should be treated as only one of a number of guides as to the adequacy of the software, others including the development methods and QA procedures used, design philosophy adopted, and performance during the fault removal testing phase. The reason for this is not only the statistical nature of the conclusions drawn and the inevitable uncertainty therefore associated with them, but also the enormous difficulties associated with making the testing as representative of use as possible, and the possible sensitivity of the statistical methods used to the assumption that testing is indeed representative of use.

Reliability growth modelling of software is an area of research which has received a great deal of attention, as a result of which a number of statistical models have been put forward to quantify the growth in reliability of a piece of software as faults are detected and repaired (refs. 26-30). The times of the occurrence of each failure are recorded and the current and future failure rates derived from these data and the model itself. The value of these models as currently developed lies in making economic decisions such as those related to the best time to release software for commercial purposes, not in predicting the reliability of software for safety critical applications. Work is continuing in this field, however, and may bear fruit for safety applications in due course, especially when diverse software is used.

The same comments as made earlier also apply to this technique, that the data used to quantify the reliability should be taken from reliability testing or (as is usually the case for commercial software) from actual operating experience, and not from any of the fault removal testing procedures.

7. RESEARCH IN UKAEA

A programme of research is in hand in the UKAEA aimed at assessing methods for developing and testing safety and protection systems involving software. Codes of practice have been drafted (e.g., refs. 31-33) concerning the structuring of the software design, documentation, in-built test facilities, use of test harnesses, stages of testing etc. Also specific methods of testing software are being proposed.

Our intermediate objectives are to assess which items in the codes of practice are of particular significance, to assess the different forms of system architecture (e.g., diversity and redundancy), to evaluate the relative power and applicability of the different test methods and to pursue the idea of quantifying the coverage achieved by a given set of tests. This endeavour will, on one hand, provide us with a tool-kit for assessing any system offered and this tool-kit would include appropriate questioning on design procedures, evaluation of tests carried out or scheduled and proposals for further tests as deemed necessary. On the other hand it will also enable us to achieve the most effective system designs.

The approach adopted for this research is to specify a protection system which is in fact hypothesised but realistic and based on a requirement to trip on low DNBR (Departure from Nucleate Boiling Ratio). This specification is being subjected to scrutiny by teams within UKAEA, CEGB and elsewhere, where codes and methodologies for the purpose of testing specifications have been evolved (refs. 34 and 35). Considerable emphasis is placed on this stage as a result of the PODS exercise referred to earlier.

The specification will be implemented 'in-house' in FORTRAN on a PDP 11/44 computer. This will be carried out as far as reasonably possible according to existing codes of practice. A careful log recording progress in developing the software is being kept. This shows time spent and progress, all test records, corrections, modifications etc; much of this will be recorded automatically at later stages.

In parallel with this in-house implementation, a contract is being placed with a software house to implement the same specification in PASCAL. This will be carried out according to high standards of commercial practice with agreed forms of testing at prescribed stages.

Both pieces of software will be subjected to extensive and varied 'acceptance testing'. This will include functional testing and several forms of structural testing, and the testing carried out will be subjected to an evaluation to assess the test coverage which has been achieved. Following this extensive back to back testing would be applied using various forms of statistical distribution in the conditions. These will include actual operating conditions, boundary or limiting conditions, and conditions anticipated at the time of causing a trip. It is important to test the system under severe operating conditions, from many points of view it is the form of circumstance most likely to be required for the system to operate.

8. CONCLUDING REMARKS

This paper has discussed a number of software development methodologies which can be use for the production of reliable software for systems important to safety, and has outlined UKAEA research aimed at assessing the efficacy of various development and testing methods with particular reference to reactor protection systems. Although the use of the methodologies described will no doubt impact beneficially upon the reliability of such systems, the current state of the art is such that the production of fault free software cannot be guaranteed; neither is there any assessment method which can guarantee software to be fault free.

In these circumstances there is a need to implement fault tolerant software in cases where failure of the safety system could be catastrophic, while recognising that in the future improved technology for software development may eliminate this need.

9. REFERENCES

1. R. R. Matthews. "Current CEGB Safety Policy and Criteria" IAEA International Conference on Nuclear Power Experience, Vienna, September 1982. IAEA-CN-42/126.

2. A. J. Bourne et al. "Defences Against Common-mode Failures in Redundancy Systems". UKAEA Report, SRD 196, January 1981.

3. J. L. Gourden et al. "The Evolution of the Core Monitoring and Protection System for Fast Reactors". IAEA International Conference on Nuclear Power Plant Control and Instrumentation, Cannes, France, 24-28 April 1978.

4. M. Josue et al. "Data Processing and Data Collection for Super Phenix". IAEA International Conference on Nuclear Power Plant Control and Instrumentation, Cannes, France, 24-28 April 1978.

5. G. J. Vaughan et al. "Sub-assembly Accident Protection Instrumentation Systems". International Topical Meeting on LMFBR Safety and Related Design and Operational Experience, Lyon, France, 19-23 July 1982.

6. A. B. Keats. "Failsafe Criteria for Computer-based Reactor Protection Systems". Nuclear Energy 1980, 19 December, No. 6, 423-428.

7. J. L. Savorin et al. "An Integrated Digital Protection System (SPIN)". IAEA International Conference on the Control and Instrumentation of Nuclear Power Stations, Cannes, France, 24-28 April 1978.

8. L. Carpentier. "New Surveillance and Control Techniques and their Application to Nuclear Power Plant with Particular Reference to 1300 MW". RGN-1980-No 3 May, June.

9. J. Bruno and J. B. Reid. "A Westinghouse Designed Distributed Microprocessor Based Protection and Control System". Proc IAEA Specialist Meeting on Distributed Systems for Nuclear Power Plant Control & Instrumentation. May 1980. Chalk River. pp101-118.

10. M. Barnes et al. "Project on Diverse Software - An Experiment in Software Reliability". Paper submitted to SAFECOMP '85.

11. M. A. Hennell, D. Hedley and I. J. Riddell. "The LDRA Software Testbeds: Their Roles and Capabilities". Proceedings of IEEE Soft Fair '83 Conference, Arlington, Virginia, July 1983.

12. D. Potier, J. L. Albin, R. Ferreol and A. Bilodeau. "Experimenting with Computer Software Complexity and Reliability", Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing, 1982, p381-389.

13. G. Rzevski. "Recent Advances in Software Reliability Methods", Proceedings of the 3rd National Reliability Conference, 1981.

14. G. A. Miller. "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information", Psychological Review, March 1956, p81-97.

15. E. Yourdon and L. L. Constantine. "Structural Design: Fundamentals of a Discipline of Computer Program and Systems Design", Prentice Hall, 1979.

16. G. J. Myers. "Composite/Structured Design", Van Nostrand, 1978.

17. M. A. Jackson. "System Development", Prentice Hall, 1983.

18. G. M. Weinberg. "The Psychology of Computer Programming", Van Nostrand, 1971.

19. A. Avizienis and L. Chen. "On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution", Proceedings of COMPSAC, 1977, p149-155.

20. B. Randell. "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, 1975, p220-232.

21. B. W. Boehm. "Software Engineering Economics", Prentice Hall, 1981.

22. M. E. Fagan. "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, 1976, p182-211.

23. M. R. Woodward, D. Hedley and M. A. Hennell. "Experience with Path Analysis and Testing of Programs", IEEE Transactions on Software Engineering, Vol. 6, No. 3, pp278-286, May 1980.

24. B. K. Daniels. "Software Reliability", Reliability Engineering, 1983, p199-234.

25. D. Bjorner and C. B. Jones. "Formal Specification and Software Development", Prentice Hall, 1982.

26. Z. Jelinski and P. B. Moranda. "Software Reliability Research", in Statistical Computer Performance Evaluation, W. Freiberger (Ed), Academic Press, 1972, p465-484.

27. B. Littlewood. "A Bayesian Differential Debugging Model for Software Reliability", Proceedings of Workshop on Quantitative Software Models, 1979, p170-181.

28. B. Littlewood and J. L. Verrall. "A Bayesian Reliability Growth Model for Computer Software", Applied Statistics, 1973, p332-346.

29. J. D. Musa. "A Theory of Software Reliability and its Application", IEEE Transactions on Software Engineering, 1975, p312-327.

30. C. J. Dale. "Software Reliability Evaluation Methods", British Aerospace Dynamics Group Report ST26750, 1982.

31. "Software Development and Maintenance Guidelines". EPRI report EL3089, 1984.

32. "Code of Practice for Documentation of Computer-based-Systems", BS5515, British Standards Institution, 1978.

33. "Guidelines for Lifecycle Validation, Verification, and Testing of Computer Software", FIPS PUB 101, US National Bureau of Standards, 1983.

34. W. J. Quirk. "The Automatic Analysis of Formal Real-time System Specifications". AERE - R9046. August 1978.

35. M. H. Gilbert, W. J. Quirk, R. P. J. Winsborrow and A. Langsford. "Applications of SPECK to the Design of Digital Hardware Systems. AERE - R10257. July 1981.

DISCUSSION

A. ROULSTONE :

(1)There are a wide range of measures available to ensure reliable software. There is a need to make rational choices about these methods. Do you advise the expendition of time on specifications rather than diversity ?

(2)There are two views about software reliability : some believe you can derive a numerical value ; others not. Which side of the argument attracts you ?

M. BUTTERFIELD :

(1)The advantages potentially available by use of diversity are limited by common mode factors such as the specification. It is therefore necessary to ensure that the specification is of high integrity if the benefits of diversity are to be maximised. Our feeling is that the two activities must go hand in hand. An important aspect of our research is to investigate the "rate of exchange" between the two.

(2)Since there is a need to quantify system reliability (etc...) and since systems contain software, the ability to satisfy the above mentioned need is limited by the ability to quantify software reliability. We feel, therefore, that it is important (i) to use whatever relevant measures are available, whilst being cognisant of there limitation, and (ii) to encourage and carry out further research enhance our currently limited abilities in this important area. It is certainly the case that current abilities enable the quantitative assessment of testing effectiveness in a relative sense, and we see possibilities that this may be extendable to the evaluation of reliability, but this is a difficult research area and the results are by no means certain.

A. POUJOL :

What kind of specification methods do you intend to use in your studies ?

M. BUTTERFIELD :

Principaly, algebra plus natural language. However it is intended to submit the specification to analysis with respect to completeness and unambiguity, using tools which can handle natural language specification. It is also intended to submit the specification to analysis by tools such as SPECK (Ref.34-35), by translating the natural language specification into an equivalent formal representation. In these ways it is hoped that we shall achieve a specification with a high level of integrity.

F. GANGEMI :

Were the errors attributed to the specifications traceable to wrong specification or to different interpretations by various people of the SPECK ?

M. BUTTERFIELD :

Specification errors consisted of ambiguities, inconsistencies, and loose ends. Most came to light during inspection by the different parties, but some only came to light during the final (back to back) testing phase, when it became clear that the specification had been interpreted in different ways without anyone realising that the ambiguity or inconsistency was actually present. These errors would not have been discovered at all but for the implementation in diverse software.