NPL
National Physical Laboratory

NPL Report
DEM-ES 014

Software Support for Metrology Best Practice Guide No. 1

Validation of Software in Measurement Systems

Brian Wichmann,
Graeme Parkin and Robin Barker

NOT RESTRICTED

January 2007

dti
FUNDED BY THE DTI

National Physical Laboratory | Hampton Road | Teddington | Middlesex | United Kingdom | TW11 0LW

Switchboard 020 8977 3222 | **NPL Helpline** 020 8943 6880 | **Fax** 020 8943 6458 | **www.npl.co.uk**

Software Support for Metrology
Best Practice Guide No. 1

Validation of Software in Measurement Systems

Brian Wichmann
with Graeme Parkin and Robin Barker
Mathematics and Scientific Computing Group

January 2007

ABSTRACT

The increasing use of software within measurement systems implies that the validation of such software must be considered. This Guide addresses the validation both from the perspective of the user and the supplier. The complete Guide consists of a Management Overview and a Technical Application together with consideration of its use within safety systems.

Version 2.2

ISSN 1744–0475

National Physical Laboratory,
Hampton Road, Teddington, Middlesex, United Kingdom TW11 0LW

We gratefully acknowledge the financial support of the UK Department of Trade and
Industry (National Measurement System Directorate)

Approved on behalf of the Managing Director, NPL by Jonathan Williams,
Knowledge Leader for the Electrical and Software team

Preface

The use of software in measurement systems has dramatically increased in the last few years, making many devices easier to use, more reliable *and* more accurate. However, the hidden complexity within the software is a potential source of undetected errors. Both users and suppliers of such systems must be aware of the risks involved and take appropriate precautions. Such risks may have safety implications for some uses of measurement systems. Safety implications are handled in this Guide by reference to the generic safety standard IEC 61508. Readers are warned that safety is a systems issue, which implies that this Guide (which only handles software) can only provide part of a solution to the validation of measurement software within safety systems.

In this Guide, we consider the implications of the use of software in measurement systems. Such software could be embedded within the measurement system, be provided by the system supplier but be separate from the measurement system, or could be developed separately to work with one or many types of measurement system. The key issue is to preserve the integrity of the measurement process.

Although users of measurement systems have a different perspective than the suppliers of measurement systems, just one Guide is provided. The reason for this is that the user needs to understand what the supplier can reasonably provide to demonstrate the integrity of the measurements. Equally, the supplier needs to be aware of the legitimate concerns of the users in order to provide assurance in a manner that can be accepted by the users. In a competitive market, a consistent approach to quality assurance for measurement software is required.

This Guide is provided in two parts. The first part covers the Management Aspects of the area and defines a framework for an approach to quality assurance. This framework is based upon a software risk assessment, which is then used to determine the most appropriate validation techniques to be applied. For safety applications, the software risk assessment approach has been designed to conform to IEC 61508.

The second part of the Guide provides the technical details of the validation techniques recommended. This material is supported by many references, research papers and related documents that can assist in the application of the techniques.

For convenience, the complete material of the Guide is also available in electronic format. The complete Guide is available from the SS*f*M website
`http://punlications.npl.co.uk/npl_web/pdf/dem_es14.pdf`, and includes *the technical part* and *the appendices*. The Management part, without *the technical part* or *the appendices*, is also available from the SS*f*M website.

This Guide is based upon a previous guide that was published under the first Software Support for Metrology programme (1999–2001). However, this new edition has a wider scope to embrace the requirements of the IEC 61508 safety standard. It also takes into account all the comments made on the previous edition. Note that the highest safety integrity level of IEC 61508 (SIL4) is not covered in this Guide, although this is planned for a subsequent edition.

This is version 2.2 of the Guide that has been revised following further comments and has been reprinted as an NPL report.

Acknowledgements

- Thanks to Bernard Chorley for his support and help with this project. Brian Bibb (Elektra Oncology Systems Ltd) provided some useful references that greatly assisted this work.
- Comments are acknowledged from: Kevin Freer (BNFL), Peter Vaswani (UKAS), John Wilson (BMTA), Brian Bibb, John Stockton, Bill Lyons (Claude Lyons Ltd), Richard Mellish (Medical Devices Agency).
- Detailed input has been obtained from Adelard on issues related to the certification of measurement systems to the safety standard IEC 61508.
- Thanks also to Roger Stillman of SIRA for appendix B.1.
- Examples were provided by Ben Hughes (NPL), Nick Fletcher (NPL) and Phil Cosgriff (Pilgrim Health NHS Trust).
- Maurice Cox and Peter Harris (both NPL) contributed to the first edition of the Guide on numerical analysis and numerical software testing.

Software Support for Metrology programme

This Guide was produced under the *Software Support for Metrology* programme 2004–2007, which is managed on behalf of the DTI by the National Physical Laboratory. NPL is the UK's National Measurement Institute and as such is a centre of excellence for measurement standards and related science and technology.

For more information on the *Software Support for Metrology* programme, contact the helpline on 020 8943 7100, e-mail: ssfm@npl.co.uk, or by post to National Physical Laboratory, Hampton Road, Teddington, Middlesex, United Kingdom TW11 0LW.

Contents

List of Tables

List of Figures

Part I

Management Aspects

Chapter 1

Management Overview

Almost all of the current generation of measurement systems contain a significant amount of software. Since it is hard to quantify the reliability or quality of such software, two questions immediately arise:

- As a *user* of such a system, how can I be assured that the software is of a sufficient standard to justify its use?
- As a *supplier* of such software, what validation techniques should I use, and how can I assure my users of the quality of the resulting software?

This Guide addresses these two questions. The intended readership comprises those responsible for software in measurement systems and those using such software. Since progress depends on both the user and supplier, this Guide merges both views, but distinguishes them by reference to *user* and *supplier* when necessary.

Software written as a research project or to demonstrate the feasibility of a new form of measurement is excluded from the scope of this Guide.

The Guide surveys current good practice in software engineering and relates this practice to applications involving measurement systems. Known pitfalls are illustrated with suggested means of avoiding them.

Many measurement systems are used as subsystems of safety related systems. In order to avoid any accident, such systems need to be seen to be appropriate for the specific application — in this Guide this issue is handled by validation against the generic safety standard [*IEC 61508*], see chapter 6.

The general approach is a five-stage process (see figure 1.1) as follows:

1. An analysis of the physical process upon which the measurement system is based, see chapter 3.
2. A software risk assessment based upon a model of a measurement system with its software, see chapter 4. In the context of a measurement system used in a safety application, the software risk assessment takes into account the safety integrity requirement of the total system as defined in *IEC 61508*, see chapter 6.
3. An assessment of reliability required of the software, based upon the software risk assessment (called the **Measurement Software Index** (MSI)), see chapter 5.
4. Guidance on the software engineering methods to be employed, as determined by the **Measurement Software Index**, see chapter 7.

5. Report produced on the validation.

In addition to the above, a clear understanding of the standards and legal requirements is clearly needed, see chapter 2.

It must be emphasised that there is no simple universal method (silver bullet) for producing correct software and therefore skill, sound technical judgement and care are required. Moreover, if it is essential for the quality of the software to be demonstrated to a third party, then convincing evidence is needed which should be planned by the *supplier* as an integral part of the software development process.

To aid in the application of this Guide, detailed technical material and checklists are provided in part II.

To avoid complexity in the wording of this Guide, it is phrased as if the software is already in existence. It is clear that use could be made of the Guide during the development, but it is left to the reader to formulate its use in that context. We are not implying retrospective validation which cannot be recommended but rather a properly planned development process in which the validation steps are as simple as possible due to their requirements being envisaged earlier in the life-cycle.

No consideration has been given here of the possibility of standardising this material, or obtaining some formal status for it. However, in the context of *IEC 61508*, it is planned that a certification process will be provided for measurement subsystems based, in part, upon this Guide.

The use of software either within or in conjunction with a measurement system can provide additional functions in a very cost-effective manner. Moreover, some measurement systems cannot function without software. Hence, it is not surprising that there is an exponential growth of software in this area. Unfortunately these changes can give rise to problems in ensuring that the software is of an appropriate quality.

The problem with software is largely one of unexpected complexity. Software embedded with a measurement system could be inside just one ROM chip and yet consist of 1 Mbyte of software. Software of such a size is well beyond that for which one can attain something approaching 100% confidence. This implies that one has to accept that there is a possibility of errors occurring in such software.

Even if a measurement system is not used for a safety application, the techniques devised in the safety arena to obtain high assurance are of interest to see what can be achieved. For general advice in the safety area, see *HF-Guide*.

An example area in which very high standards are required in software production is that for airborne flight-critical software. The costs for producing such software can easily be one man-day per machine instruction — obviously too demanding for almost all software within (non-safety) measurement systems. Hence the main objective behind this Guide is to strike a balance between development cost and the proven quality of the software, taking into account the application.

The main approach taken here is one of *software risk assessment* as a means of determining the most appropriate level of rigour (and by implication, cost) that should be applied in a specific context. This approach is modified slightly in the context of safety systems to take into account the *safety integrity* objectives in *IEC 61508*.

A major problem to be faced with software is that the failure modes are quite different from those of a simple instrument without software. An example of this is that of the non-linear behaviour of software, in contrast to simple measuring devices — see **Software is non-linear** in section A.1.

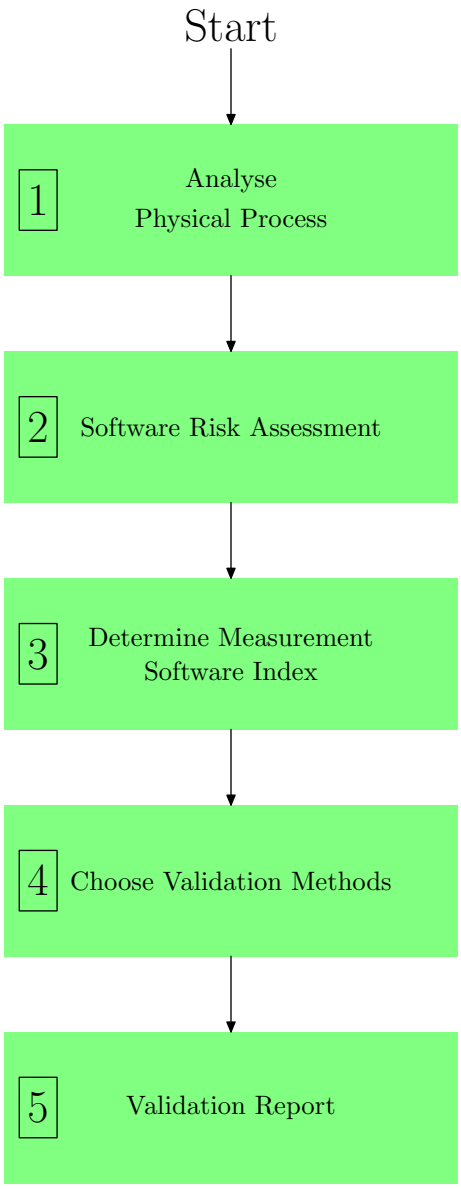

Figure 1.1: Steps in Validation

Summary

Managers of supplier organisations need to be aware of the vital parameters of their measurement software — the complexity and novelty of the application together with the Measurement Software Index. This Index will determine the rigour and hence the cost of the validation procedures required.

Those users requiring high assurance software, perhaps due to the use in a safety context, should read Part I so that the most appropriate means of obtaining the assurance can be found.

The rest of Part I considers the various aspects of the software which have a direct bearing upon the above issues, and hence the likely cost of the development.

Chapter 2

Standards and Legal Requirements

There are a number of standards that specify requirements for software in measurement systems, which are collected together here with an indication of the issues to be covered by this Guide. The more relevant standards appear first. Standards are important since they are an agreement between the *user* and *supplier* communities.

ISO/IEC 17025. This standard [*ISO/IEC 17025*] for testing and calibration laboratories, replaces both *ISO Guide 25* and *EN 45001*. The vital paragraph on computers is as follows:

> *5.4.7.2 When computers or automated equipment are used for the acquisition, processing, recording, reporting, storage or retrieval of test or calibration data, the laboratory shall ensure that:*
>
> *a) computer software developed by the user is documented in sufficient detail and suitably validated as being adequate for use;*
>
> *b) procedures are established and implemented for protecting the data; such procedures shall include, but not be limited to, integrity and confidentiality of data entry or collection, data storage, data transmission and data processing;*
>
> *c) computers and automated equipment are maintained to ensure proper functioning and are provided with the environmental and operating conditions necessary to maintain the integrity of test and calibration data.*
>
> *NOTE Commercial off the shelf software, (e.g., word processing, database and statistical programs) in general use within its designed application range may be considered sufficiently validated. However, laboratory software configuration/modifications should be validated as in 5.4.7.2a.*

One can see that the main emphasis is upon software developed by the user rather than the software embedded within an instrument or that within standard packages. Although the major software risk may well be with user-developed software, it should not be assumed that other software is without any software risk. It would be reasonable to make no checks on word processing software, but just inspect the resulting documents. The same is not true of spreadsheets packages in which unstable (or less than ideal) algorithms are known to exist but which cannot be so easily detected [*CISE 25/99*]. Similarly, simple embedded software should not be a serious software

risk [*SmartInstrument*]. However, complex software is bound to be a software risk since one cannot expect all possible uses to have been checked by the supplier.

A technical report [*ISO 13233*] was produced on the interpretation of accreditation requirements for software testing based upon *ISO Guide 25*, the predecessor to *ISO/IEC 17025*; however this interpretation does not specifically deal with software in measurement systems: unfortunately, there are no plans to revise this interpretation document.

Legal Metrology. The WELMEC documents summarise the position for such applications [*WELMEC–2.3, WELMEC–7.1*]. The measuring instruments in this class include weighing machines, gas / water / electricity meters, breath analysers and exhaust gas analysers. The requirements here derive from the EU Directive 90/384/EEC, which has three relevant parts as follows:

1. Annex I, No 8.1: *Design and construction of the instruments shall be such that the instruments will preserve their metrological qualities when properly used and installed, and when used in an environment for which they are intended....*
2. Annex I, No 8.5: *The instruments shall have no characteristics likely to facilitate fraudulent use, whereas possibilities for unintentional misuse shall be minimal. Components that may not be dismantled or adjusted by the user shall be secured against such actions.*
3. Annex II, No 1.7: *The applicant shall keep the notified body that has issued the EC type-approval certificate informed of any modification to the approved type....*

Clearly, only a subset of these requirements is relevant to the software of measurement systems in general. The WELMEC Guide derives three specific requirements for software from the above Directives, as follows:

1. Section 3.1: *The legally relevant software shall be protected against intentional changes with common software tools.*
2. Section 3.2: *Interfaces between the legally relevant software and the software parts not subject to legal control shall be protective.*
3. Section 3.3: *There must be a software identification, comprising the legally relevant program parts and parameters, which is capable of being confirmed at verification.*

In the context of measurement systems not within the ambit of legal requirements, there are two important principles to be noted from the above:

- The handling of the basic measurement data should be of demonstrably high integrity.
- The software should be properly identified (this arises from configuration control with *ISO 9001* [*ISO 9001*], in any case, but there is no requirement that *ISO 9001* is applied to such systems).

IEC 601–1–4. The standard covers the software in medical devices [*IEC 601*] and is used both in Europe to support a Directive and by the FDA in the USA [*FDA*]. The standard is based upon risk assessment with the software engineering based upon *ISO 9000–3*. The flavour of the standard can be judged from a few key extracts, with those parts relevant to this Guide being:

1. Section 52.204.3.1.2: *Hazards shall be identified for all reasonably foreseeable circumstances including: normal use; incorrect use.*
2. Section 52.204.3.1.6: *Matters considered shall include, as appropriate: compatibility of system components, including hardware and software; user interface, including command language, warning and error messages; accuracy of translation of text used in the user interface and "instructions for use"; data protection from human intentional or unintentional causes; risk/benefit criteria; third party software.*
3. Section 52.204.4.4: *Risk control methods shall be directed at the cause of the hazard (e.g. by reducing its likelihood) or by introducing protective measures which operate when the cause of the hazard is present, or both, using the following priority: inherent safe design; protective measures including alarms; adequate user information on the residual risk.*
4. Section 52.207.3: *Where appropriate the specification shall include requirements for: allocation of risk control measures to subsystems and components of the system; redundancy; diversity; failure rates and modes of components; diagnostic coverage; common cause failures; systematic failures; test interval and duration; maintainability; protection from human intentional or unintentional causes.*
5. Section 52.208.2: *Where appropriate, requirements shall be specified for: software development methods; electronic hardware; computer aided software engineering (CASE) tools; sensors; human-system interface; energy sources; environmental conditions; programming language; third party software.*

It can be seen that this standard is mainly system-oriented and does not have very much to state about the software issues. However, the key message is that the level of criticality of the software must be assessed, and the best engineering solution may well be to ensure the software is not very critical. This standard covers only instruments that are on-line to the patient as opposed to those used to analyse specimens from a patient (say). Not all medical applications could be regarded as "measurement systems", and therefore the relevance of this guide needs to be considered.

IEC 61508. This standard is a generic standard for all safety-critical applications [*IEC 61508*], which allows for many methods of compliance. A catalogue is provided in Part 3 [*IEC 61508–3*] of the standard, which handles the software issues. This catalogue is used here as a means of selecting specific methods that may be appropriate in some contexts. This Guide follows the same approach and specific recommendations are given when a measurement system is to be used in a safety context. Guidelines for use within the motor industry have been developed from this standard [*MISRA*].

The conclusion for this Guide is that this standard is only directly relevant when the measurement system is used in safety applications, but could be applied in specific contexts. The catalogue of techniques provides a reference point to a wide variety of software engineering methods. For an analysis of this standard for accreditation and certification, see *UKAS-61508*.

For a detailed research study of assessing instruments for safety application by means of a worked example, see the SMART Reliability study [*SmartInstrument*]. This study was based upon a predecessor of this Guide, namely NPL Good Practice Guide No. 5, on Software in Scientific Instruments, but enhanced to reflect the safety requirements. The issue of the qualification of SMART instruments in safety applications is noted as a research topic in a Health and Safety Commission report [*HSC*].

In this Guide, appendix B.4.3 considers the detailed requirements of this standard, as applied to software (Part 3). Those who require adherence to this standard therefore need to study that appendix.

DO-178B. This is the standard [*DO-178B*] for civil avionics safety-critical software. It is not directly relevant. However, if an instrument were flight-critical, then any software contained within it would need to comply with this standard. In practice, instruments are replicated using diverse technologies and hence are not often flight-critical. This standard is very comprehensive, and specifies an entire software development process, including details on the exact amount of testing to be applied.

The conclusion for this Guide is that this standard is only relevant for very high-risk contexts in which it is thought appropriate to apply the most demanding software engineering techniques. These demanding techniques are relevant for safety systems with the highest safety integrity requirements in *IEC 61508*, which corresponds to Measurement Software Index 4 (see section 5.2, on page 28). This standard can be taken as an ideal goal, not however achievable in practice, due to resource constraints.

The conclusion from this survey of the standards is that most of their requirements on software are broadly similar and that aiming to meet all the requirements is a reasonable way of proceeding.

In the context of safety systems, high safety integrity can be required which then imposes very significant costs for the development and validation. These costs are very evident from the Class A and B systems from *DO-178B*. Hence, if a measurement system is required to meet the most demanding levels of *DO-178B*, then that cannot be expected of an instrument designed to satisfy the other standards mentioned here. Thus this Guide aims to provide advice on producing software which will satisfy any of these standards, including everything but the most demanding level for safety systems. It is planned that this exclusion will be removed in the next edition of this Guide, as the SS*f*M programme for 2004–2007 [*SSfM–3*] will work on this issue.

Chapter 3

From Physics to Measurement

A measuring device is based, of course, on physical properties. However, the underlying physics can be very simple and well-established, or complex, or not well-established. The software may need to model a complex physical process and therefore require a correspondingly demanding validation process.

3.1 Specification of the physical processes

The issues that arise from the modelling of physical processes have been considered in the Modelling Theme [*SSfM-Model*] of the SS*f*M programme [*SSfM–2*] but are summarised here.

In a perfect world, building a mathematical model and solving it would be a well-defined process. It is the fact that the model is an approximation of the real world that causes difficulties.

Two questions have to be addressed (essentially by the *supplier*, but the *user* needs to consider this also to appreciate the software risk involved):

- Does the software specification provide a faithful implementation of the physics underlying the operation of the measurement system?
- Does the software implement the specification correctly?

In particular, the first question can be quite difficult to answer and requires input from an expert metrologist (physicist, chemist, biotechnologist, materials scientist or engineer; rather than mathematician or software engineer).

3.2 Producing the model

Some of the issues to be considered:

1. The use of a mathematical model to provide the conceptual interface between the metrology and the software. The model must include descriptions of
   - the measured quantities,
   - the errors in the measured quantities,
   - the measurement result(s),

- the relationship between measured quantities and measurement result(s).

2. There is a wide range of complexities of mathematical model. However, the model is generally required to be sufficiently complex to reflect accurately the physical situation but also sufficiently simple for the software computations based on the model to be feasible.

3. An estimate of the measurement result is obtained by solving the model in some sense, taking into account the defined error structure. Again, the complexity of this process can vary widely.

4. Approximations and assumptions are often made when writing down the mathematical model. Models are generally approximate (certainly for empirical models but also for physical models), and model validation is a means to understand and quantify any modelling error. Assumptions are also made about the errors in the measured quantities (particularly with regard to the underlying probability distributions) that then affect the solution process and the accuracy of the measurement result. Metrics such as *bias* and *efficiency*[1] are often used to measure and compare the performance of different solution algorithms that reflect different sets of assumptions.

5. The procedure for establishing the internal data, such as calibration constants, might be considerably more complicated that the procedure implemented by the measurement system's software.

6. The need to consider the affect of the control software (see figure 3.1). For example, the control software might make the model more complicated by introducing additional measurement error.

3.3 Validating the implementation of the model

Having established a software specification, the second question is generally more clear-cut. It is the subject of the SS*f*M numerical software testing project [*CISE 25/99*], although there are perhaps some particular issues that arise for software that is embedded within a measurement system:

- The requirements on the software, for example, numerical accuracy requirements, are typically dictated by the physical application.
- The software can be tested in isolation from control software and the basic instrument.
- In common with other SS*f*M projects, when model validation is required, we advocate the use of Monte-Carlo simulation to validate the mathematical model.

We have several categories of measurement system technology in this area:

Validated physics[2]: This means that the physical model is widely understood and used. Any discrepancy in practice between the model and the observations are within the

[1] These two measures relate to the skewness and spread (respectively) of the resulting measurement caused by the error in the raw measurement data, for which an uncertainty estimate is produced; see the Guide on Uncertainty [*BPG-Uncertainties*].

[2] "Physics" is specified here, but these categories could equally apply to chemistry, biotechnology, materials science or engineering — whatever is the scientific or technical basis for the metrology.

tolerance of the accuracy specified for the measurement system. Obviously, it should be possible to quote a reference to the physical model in the reviewed literature. (Only this category would be suitable for safety applications.)

Physics^2 can be modelled: The physics can be modelled, but there is no widely agreed literature supporting such a model as the basis of a measurement system. Hence, for high assurance measurement, it is necessary to validate the model being used.

See the relevant part of the SS*f*M programme [*SSfM–2*] for research and information in this area [*SSfM-Model*]: *BPG-Model* for discrete modelling and model validation, and *CMSC 29/03* for model validation in continuous modelling.

Underlying model uncertain (pragmatic approach): In some situations, there is no precise physical model that can be expressed mathematically. A pragmatic approach may be needed, such as calibration at a number of points together with interpolation when in use. The implications of such a method needs to be understood. For instance, it is necessary to ensure that any non-linear effects are properly taken into account. (See section 3.7 for a problem in this area.)

Physics^2 being researched: In this situation, even the model is still to be determined. Measurement systems in this category are not included within the scope of the Guide.

The *supplier* should be able to inform the *user* which of the above categories applies to a specific measurement system.

Given a measurement system as a black-box, then the physical process being used might not be apparent. Obviously, in that case, the measurement system can only be tested as a whole and therefore the question of the quality of the software does not arise directly. This implies that any errors that may be present in the software could only be observed by appropriate use of the measurement system.

3.4 A measurement system model

In order to provide a framework for the discussion of the software for a measurement system, we present here a simple model. The components of the model in figure 3.1 are as follows:

Basic instrument. The basic instrument contains no software. It performs functions according to the control logic and provides output. This instrument itself is outside the scope of this Guide, but it is essential that the properties of the instrument are understood in order to undertake an effective appraisal of the software. The basic instrument contains sensors and appropriate analogue/digital converters.

Control software. This software processes output from the basic instrument for the purpose of undertaking control actions.

Data processing software. This software performs a series of calculations on data from the basic instrument, perhaps in conjunction with the control software, to produce the main output from the measurement system. (In the case of complex measurement systems, like Coordinate Measuring Machines, the data processing software provided can include a programming language to facilitate complex and automatic control.)

Internal data. This data is held internally to the measurement system. A typical example would be calibration data. Another example might be a clock that could be used to "time-out" a calibration.

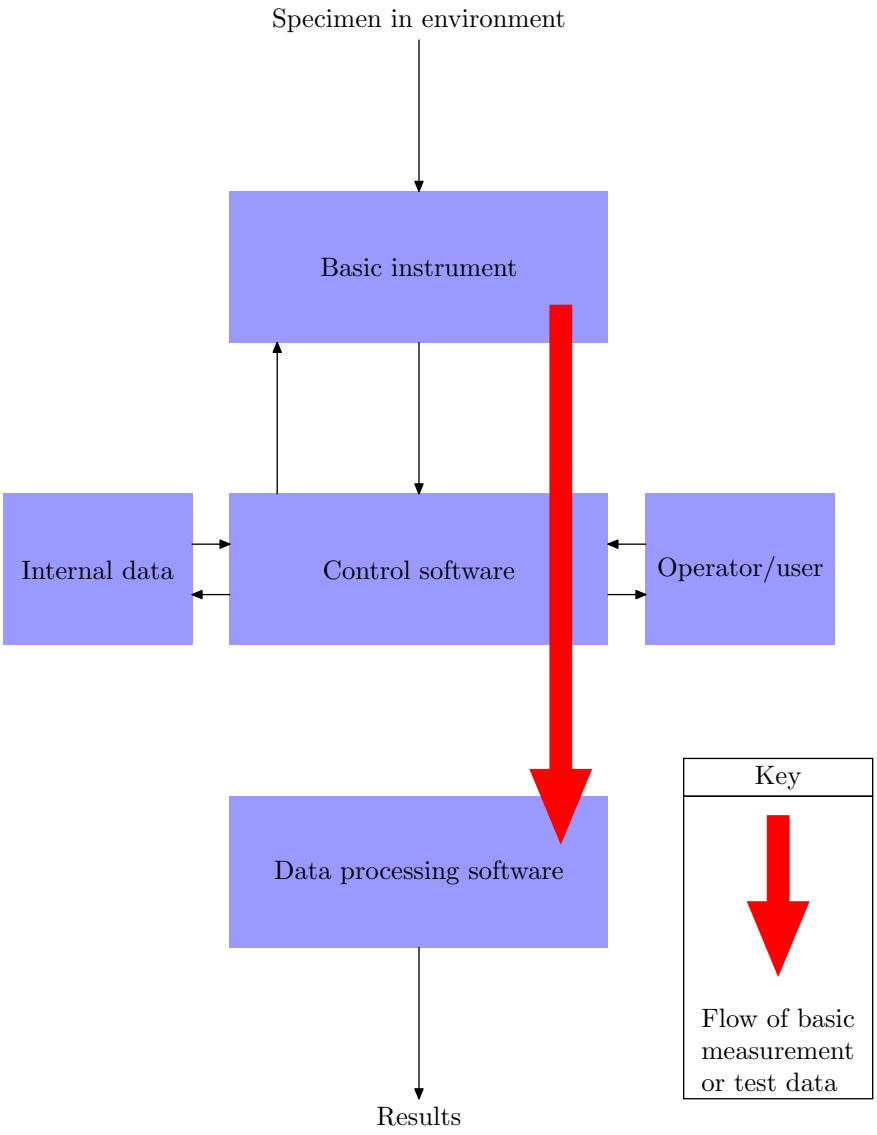

Figure 3.1: A model of a measurement system

Operator/user. In some cases, there is an extended dialogue with the user, which implies that the control function can be quite complex. This dialogue could be automated via some additional software (which therefore could be included or excluded from this model).

In this model, we are not concerned about the location of the software. For instance, the control software could be embedded within the measurement system, but the data processing software could be in a PC or workstation. It is even possible for the subsequent data processing to involve several computers via a Laboratory Information Management System (LIMS). In applying this Guide, you may have a choice in deciding where to draw the line at the bottom of this diagram. For instance, one could decide to include or exclude a LIMS. If the LIMS is considered, then reference *LIMS* on a medical application provides some useful insights, and *BPG-LIMS* has advice of the selection and use of LIMS. The integrity of the production of the test/measurement certification should not be forgotten.

The basic measurement/test data is a key element in this structure. The major requirement is to show that the processing and entire flow of this data has sufficient integrity. Note that in the area of legal metrology, the basic measurement data is converted into money (say, in a petrol pump) and this therefore has the same status as the basic measurement data.

It is important to decide the scope of the measurement system. For instance, in DNA analysis, samples are analysed for the Base-Pairs in the DNA. Subsequently, the Base-Pairs found are compared with other samples or population statistics to prepare a statement for the Courts. In this context, it seems that only the determination of the Base-Pairs could be regarded as measurement system software.

3.5 Abstraction at several levels

A measurement system model can exist at several levels and the appropriate level of abstraction must be chosen to fit the objectives of validation. The different levels are illustrated with reference to a Coordinate Measuring Machine (CMM), see figure 3.2.

The basic machine just produces (x, y, z) coordinates from some artefact. Hence the processing software within the basic instrument is probably quite simple — just performing a small calibration correction to the raw data. On the other hand, the control software could be quite complex if the measurement probe is to be moved round a complex object. One would like to be assured that this complex control software cannot result in false measurements.

Now consider another measurement system that measures artefacts to determine their sphericity (departure from a sphere). This measurement system is actually a CMM with some additional software running on a PC. We now have a measurement system with complex data processing software that needs to be carefully checked if the measurement of the departure from a sphere is to be authenticated.

We can envisage a further level where the sphericity data is processed by visualisation software on the user's PC. The complete system from CMM probe to images on the user's PC is a measurement system and it may be appropriate to validate the system including the visualisation software.

All three examples satisfy the model of a measurement system used in this Guide. However, it is clear that the implications are very different. Hence it is important that the appropriate level of abstraction is chosen for the application in mind.

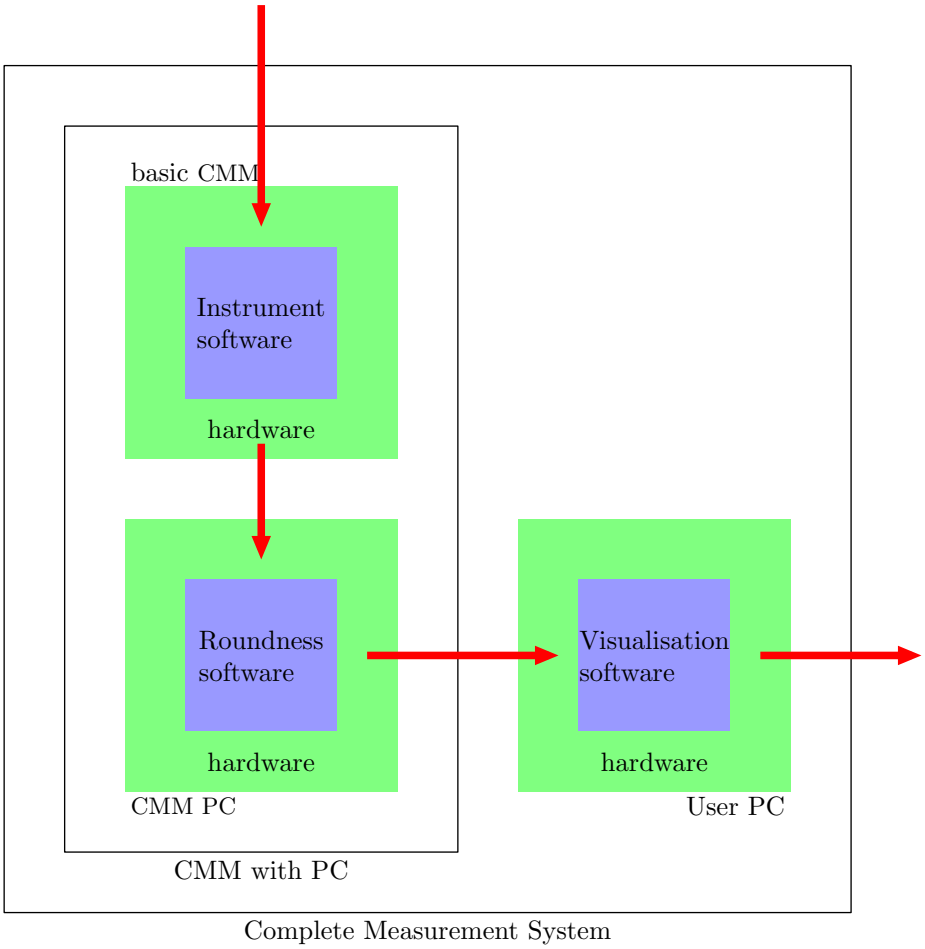

Figure 3.2: Levels of measurement system abstraction

3.6 Example 1: Druck's pressure transducer

As part of the research programme funded by the Nuclear Industry, NPL undertook a study on the safety integrity[3] of SMART instruments [*SmartInstrument*]. This study was specifically to investigate the software aspects of safety integrity, which required that the software design and implementation was analysed. To this end, Druck Ltd assisted NPL in providing full details of the software. NPL was effectively taking the role of the *user* (on behalf of the nuclear industry with a safety application), while Druck was the *supplier*.

The results of this study can be summarised as follows:

1. The physical model used was simple.
2. Modelling the physics within the instrument was numerically sound.
3. The software, written in C, passed all the tests run by NPL.
4. There was no reason why such an instrument, using software, should not be used in a safety application.

NPL had complete visibility of the software, and the results satisfied the specification of the instrument.

Since this was a research exercise, there was no formal assessment of the instrument nor certification to *IEC 61508*.

3.7 Example 2: A vacuum pressure measurement device

NPL has had experience in the past with a pressure measurement device in the vacuum range that works by observing the slowing down of a rotating sphere. For low pressures, the rate of deceleration of the sphere is proportional to the pressure. However, at higher pressures, which is within the range of the device, the linear relationship breaks down. Software within the device compensates for this. However, NPL observed that this compensation was not correct in parts of the range, thus giving rise to a "kink" in the graph of the actual pressure against the observed pressure.

Several lessons can be learned from this example:

1. Compensation for a non-linear response should be specified by the measurement system supplier in case the user wishes to take note of any potential software risk that this might entail.
2. The dangers are clearly greater if high accuracy is required.
3. If the "raw" data from the measurement system is available (in this case, the observed deceleration), then the compensation can be done externally or checked separately.
4. Even if a spot check is made for a single high value, the compensation error might not be detected.

[3] The study uses the term reliability instead of the more correct term of safety integrity.

Chapter 4

Assessing the Software Risks

In this section, we undertake an analysis of a measurement system and its related software, the purpose of which is to make an objective assessment of the likely *software risks* associated with a software error. For a general discussion on risk, see *RiskIssues*. The primary responsibility for the software risk assessment must rest with the *user*, but for the user to undertake this in a meaningful manner, information from the *supplier* is often vital.

4.1 Software Risk factors

The first step in undertaking the software risk assessment is to characterise the measurement system according to aspects that influence the software risk. Those aspects are listed below.

A software risk assessment must be based upon the visible information (one must assume the worst when critical information is not available). Some complex devices may internally record information that is difficult or impossible to access. Examples of such information are the selection of operator options, or low-level data within a complex system. For user-programmable systems, it should be possible to reconstruct the user program from a listing, and repeat the execution from the data recorded from a prior execution.

The *user* needs to study the information provided by the *supplier* to confirm that the software risk assessment is based upon a correct understanding of the measurement system.

4.1.1 Criticality of Usage

It is clear that the usage of some measurement systems is more critical than others. For instance, a medical instrument could be critical to the well-being of a patient. On the other hand, a device to measure noise intensity is probably less critical.

To make an objective assessment of the level of criticality of usage, we need a scale which we give in increasing criticality as: **critical**, **business-critical**, **safety-indirect** and **safety-direct**. (If the system has no degree of criticality, then no specific validation requirements would be appropriate.)

Criticality of usage can be categorised as follows:

Critical. This category is used if correct operation of the measurement system software is considered sufficiently important that appropriate software validation should be undertaken, without being in any of the higher categories described below.

Business Critical. This category applies to that measurement system software for which failure could cause serious financial loss, either directly or indirectly.

Safety-indirect. This category applies to that measurement system software for which failure indirectly puts human life at risk or directly puts human health at risk. Such software is considered safety-critical in the context of *IEC 61508*.

Safety-direct. This category applies to that measurement system software for which failure directly puts human life at risk, and hence is safety-critical.

One of the major problems in this area is that a *supplier* may well be unaware of the criticality of the application. The *user* may well assume that the measurement system is suitable for highly critical applications, while the supplier may well prefer to exclude such usage. For an example of this problem see **Buyer beware!** in appendix A.4. Naturally, for safety systems, critical components *would* require full information from the supplier[1].

4.1.2 Legal requirements

Many measurement systems are used in contexts for which there are specific legal requirements, such as the WELMEC guides [*WELMEC–2.3, WELMEC–7.1*]. In such a context, a measurement system malfunction could have serious consequences.

To make an assessment, the *user* needs to know if there are any specific legal requirements for the measurement system and have a reference to these. (It may be necessary to check what current legislation applies.) For a general-purpose measurement system, the *supplier* may not have produced the measurement system to satisfy the legal requirements of a particular usage.

4.1.3 Impact of complexity of control

The control function of the software can range from being almost non-existent to having substantial complexity. Aspects of the control will be visible to the operator in those cases in which operator options are available. Some control may be invisible, such as a built-in test function to help detect any hardware malfunction.

Many aspects of control are to make the device simpler to use, and protect the operator against misuse which might be feasible otherwise. This type of control is clearly highly advantageous, but it may be unclear if any error in its operating software could produce a false reading. Hence aspects of the complexity of the user-instrument interface are considered here.

Complexity of control is classified as follows:

Very Simple. An example of a very simple control is when the measurement system detects if there is a specimen in place, either by means of a separate detector, or from the basic data measurement reading. The result of this detection is to produce a more helpful display read-out.

Simple. An example here might be temperature control which is undertaken so that temperature variation cannot affect the basic measurement data.

Modest. An example of modest complexity arises if the measurement system takes the operator through a number of stages, ensuring that each stage is satisfactorily complete

[1] In general, if information is not available, assurance at a high level is not possible.

before the next is started. This control can have an indirect effect upon the basic test/measurement data, or a software error could have a significant effect upon that data.

Complex[2]**.** An example of complex control is when the software contributes directly to the functionality of the measurement system. For instance, if the measurement system moves the specimen, and these movements are software controlled and have a direct bearing upon the measurement/test results.

The *supplier* should be able to provide the *user* with the measure of the complexity of control on the above scale.

In assessing the complexity of control as part of the software risk assessment of software in a measurement system, it is necessary to only consider the impact of the control software on the processing of the measurement data. Therefore, the software risk factor to be assessed is not the complexity of the control software *per se*, but the impact of complexity of control.

4.1.4 Complexity of processing of data

In this context, we are concerned with the processing of the raw data from the basic instrument (i.e., the instrument without the software). In the case of software embedded within the measurement system itself, the raw data may not be externally visible. This clearly presents a problem for any independent assessment; however, it should be the case that the nature of the raw data is clear and that the form of processing is well-defined. Calibration during manufacture would typically allow for "raw data" to be displayed in appropriate units. (Subsequent to the calibration during manufacture, the raw data may not be available to the user.)

Complexity of processing data is classified as follows:

Very Simple. In this case, the processing is a linear transformation of the raw data only, with no adjustable calibration taking place.

Simple. Simple non-linear correction terms can be applied here, together with the application of calibration data. A typical example is the application of a small quadratic correction term to a nearly linear instrument which is undertaken to obtain a higher accuracy of measurement.

Modest. Well-known published algorithms are applied to the raw data.

The assumption here is that the algorithms used are numerically stable (and there is evidence for this). For an example of a problem that can arise in this area, see **Numerical instability** in section A.2.

Complex. Anything else.

The *supplier* should be able to provide the *user* with the measure of the complexity of data processing on the above scale.

[2]This term should not be confused with that in complex arithmetic — readers are invited to propose another term should they find it confusing.

4.2 Reliability of software

Measuring the reliability of software is not easy — unless it is very unreliable!

If an item of software has a long history and a lot of users, then there may well be good independent evidence of high reliability. However, most large items of software are continually enhanced making any measurement problematic.

Reliability growth models *Littlewood* and *ReliabilityBounds* can be used to estimate reliability, but would depend upon gathering suitable data which is typically difficult or impossible.

Consider a simple case in which software acts on a per-demand basis and the reliability requirement is better than one failure per thousand demands. Here, one could justify that figure by running two or three thousand statistically valid tests (see 12.12). If no failures occurred, or just one or two which were satisfactorily handled, then such a claim might be sustainable. However, if the requirement was better than 1 in 10,000 or 1 in 100,000, the cost of undertaking such testing would become prohibitive. To reduce the costs, the testing could be biased to the "difficult" test cases, but then the statistical reasoning will be flawed.

An alternative to such a statistical approach to reliability is a formal proof of correctness of the software. In practice, this is hard to achieve and virtually impossible with most measurement software which relies upon floating point arithmetic. Of course, such proofs depend upon the correctness of the compiler and the underlying hardware.

4.3 Some examples

In the case of Druck's pressure transducer (see section 3.6), from the supplier's point of view it is **business-critical**, but from the point of view of its application in the nuclear power plant, it is **safety-direct**. In practice, this mismatch requires that any additional validation needed from safety application would need to be funded by the users. (Indeed, the validation could reveal defects in the design, thus requiring a substantial re-work of the whole system.)

Instruments which are developed for the safety market should be suitable for formal certification, see chapter 6.

There is no doubt that the majority of measurement systems at present perform **very simple** or **simple** processing on the raw data. It would be reasonable to assume the software risks of the software error were quite low, but the example of the vacuum pressure measurement device (see section 3.7) shows that such software risks cannot be totally ignored. Also one needs to beware the trend to increase the complexity of the data processing with each new measurement system.

4.4 A warning from history

From 1985, for a period of 5 years, the FDA had been warning the pharmaceutical industry that it was expecting it not only to validate their processes but also the computer systems that were involved. Unfortunately the pharmaceutical industry did not take this warning on board.

In 1991 the FDA conducted a foreign inspection of pharmaceutical companies in Europe and, for the first time, included computer validation.

At a number of sites the FDA issued a 483 (major non-compliance) against the computer control system of an Autoclave for not being validated. An autoclave is used to sterilise drugs after container filling.

The non-compliances were noted as:

- No formal plans stating the validation requirements (i.e., Validation Plan).
- Specifications were not available to define the intended software operation.
- No qualification protocols[3] defining testing or acceptance criteria.
- Inadequate accuracy checks, input-output checks and alarm testing.
- Inadequate change and version control of software.
- No final review of the evidence demonstrating validation (i.e., Validation Report).

A validation plan was immediately drawn up for the Autoclave and put into action. It covered specification generation, installation qualification (IQ — hardware testing), operational qualification (OQ — software testing), code walk-throughs and supplier audits.

The results were surprising.

1. In the IQ/OQ phases one alarm code was found to be missing, two limits were crossed over and the real time clock was found to be of insufficient accuracy over a period of time to support integration techniques.

2. The more critical areas of code were checked by walking through the program to ensure that it had been correctly coded to function in the way it was designed and that it had been written to the company defined standard.

   The Code Walk-through addressed:

   - Software Design.
   - Adherence to coding standards.
   - Software Logic
   - Absence of redundant code.
   - Critical Algorithms.
   - Software control.
   - Level of code comments/explanation.

   The software was well written but the main algorithm for calculating F_0 (F_0 is defined as the time during which sterilisation is actually performed at 121°C i.e., time of microbial destruction effectiveness) was found to ignore summation of a small area under the curve giving a lower F_0 calculation over the sterilisation period. The result of this was to subject the product to a longer sterilisation period than required.

   The computer manufacturer was informed of the findings and the company responded quickly in rectifying the problems.

3. In setting up the supplier audits for the Autoclave it was discovered that the computer system was built and tested by a computer manufacturer and then application programmed and fitted by the autoclave manufacturer. A decision was taken to audit both companies.

[3]The term "protocol" means a documented procedure with defined acceptance criteria.

The computer manufacturer was situated in a building where the complete hardware and software design process was carried out. The full life cycle was checked out, right through to testing and final shipment packing to Autoclave manufacturer. No problems were encountered and a high degree of confidence was recorded in the manufacture of the computer system.

A visit was then paid to the Autoclave manufacturer and the audit trail started from where the tested computer system was delivered into stores. The stores were at one end of a very large workshop area where approximately 20 autoclaves of various sizes were in different stages of manufacture. The computer system was followed to a small computer workshop, next to the stores, where the correct sequence software was installed for the Autoclave it was to control. From there it was followed to the workshop floor where it was stored under the nearest workbench to the autoclave it was designated for. Building an autoclave, which is essentially a large kettle, is mainly an engineering and plumbing task hence there were many workbenches around for people to use. It was noticed that some of the unpacked, unprotected computer systems were beneath bench vices and in a number of cases metal filings had dropped onto the computer case which had open air vents in it. A number of other major problems were found, one being that there were no written tests or recorded test data. It depended on the person building the autoclave and the test plan he kept in his head.

Once the problems were pointed out to the Autoclave manufacturer, the company responded very quickly and it now has a completely different method of computer integration and test which ensures the computer controlled autoclave is produced to the highest quality.

4.5 Conclusions

The above example has been supplied to NPL by industry as an example of the problems actually encountered. The originators wish to remain anonymous.

The conclusions are clear: a software risk assessment needs to be undertaken as early as feasible and the consequences followed through the entire process of the use of the system.

In the example, the legal requirements were crucial. Even without that, the pharmaceutical supplier needs to be assured that the Autoclave performs the intended function, since a failure could have terrible consequences. In this case, the Autoclave supplier accepted the requirement for an audit and undertook the necessary remedial action. The purchase of off-the-shelf equipment may well present more problems in gaining visibility of the necessary information.

Chapter 5

Software Implications

At this point, the *user* will have most of information needed to make an assessment of the reliability required of the software, taking into account all the software risk factors including the target software risk to be taken. This assessment should be as objective as possible, but is bound to have a modest degree of subjectivity. The result of the assessment is the assignment of the **Measurement Software Index** for the software. The MSI is used to determine which techniques are recommended for validation of the software in the measurement system.

There are further software issues that either affect the software risk factors defined in the previous chapter or may cause the MSI, which is derived from those factors, to be modified. To attempt to resolve these additional issues, they are expressed in the form of open questions. At this stage, we are looking at the measurement system as a black box but assuming that some questions can be asked (and answered) which might not be directly apparent from the measurement system. The underlying reasoning behind the questions is to assess the affects of the software risk factors involved. If some key questions cannot be answered, then clearly any assessment must be incomplete.

5.1 Software issues

Having gone through the previous section of this Guide, these additional questions should be answered. The answers may change the software risk factors or these may change the MSI derived from the software risk factors; the answers may also suggest changes to the selection of validation techniques recommended for the MSI. For each question the effect on the software risk factors, or on the MSI, is stated in *sloping text*.

1. What degree of confidence can be obtained in the measurement system merely by performing "end-to-end" tests,[1] i.e., using the measurement system with specimens of known characteristics?

   Such tests just regard the entire measurement system as a black-box and effectively ignore that software is involved. To answer this leading question you need to take into account the software risk factors noted in section 4.1. For instance, if complex software is being used which uses unpublished algorithms, then high confidence cannot be established.

[1] The term "end-to-end" is used for testing the measurement system as a whole — readers are invited to propose another term should they find this confusing.

As a *user*, you may have no visibility of the software and hence this could be the only form of testing that could be undertaken. This will clearly limit the assurance that can be obtained, at least if the software alone could cause the measurement system to malfunction.

As a *supplier*, you may need to provide some additional information on the measurement system for those users who require high assurance.

(Note that this type of testing is distinct from conventional black-box testing of the software since the software is only exercised in conjunction with the basic instrument.)

If a high degree of confidence can be obtained in the measurement system by performing "end-to-end" tests, then the assessment of the complexity of processing and (impact of) complexity of control can be reduced.

2. In the case in which the processing of the basic data is **modest** or **complex**, can the raw data be extracted so that an independent check on the software can be applied?

   Consider a case in which a Coordinate Measuring Machine (CMM) is being used to determine the degree of departure of an artefact from a perfect sphere. The basic CMM will provide the x, y, z coordinates of points on the surface of the artefact, and additional software will determine the departure from the sphere. For high assurance, it is essential that the additional software should be tested in isolation, preferably using Numerical Reference Results (see section 12.18).

   If the raw data can be extracted so that an independent check on the software can be applied, then the assessment of the complexity of processing can be reduced.

3. Has essentially the same software for the data processing been applied to a similar measurement system for which reliability data is available? (Note that there is a degree of subjective judgement here which implies that the question should be considered by someone who is suitably qualified.)

   The *supplier* may regard such information as commercially sensitive and hence is not provided to the user, or is only available under a non-disclosure agreement.

   If the same software for the data processing has been applied to a measurement system for which reliability data is available, then this reduces the overall software risks associated with the software and the MSI can be reduced.

4. For this measurement system, is there a record available of all software errors located? Under what terms, if any, can this information be made available?

   The *user* could reasonably request this information and the *supplier* should be able to supply it, perhaps under some non-disclosure agreement. Note that the existence of such a log is a requirement of quality management systems, like *ISO 9001*.

   If there is a record of software errors then this reduces the overall software risks associated with the software and the MSI can be reduced.

5. To what extent can the complexity in the control software result in false measurement/test data being produced?

   The *user* is likely to have great difficult in answering this question while the *supplier* may be confident of the measurement system, but finds it difficult to provide objective evidence to support that confidence.

   If the complexity in the control software result in false measurement/test data being produced, then the software risk factor "impact of complexity of control" will be as

high as the complexity of control (as explained in section 4.1, item 4.1.3). Conversely, if complex control software cannot result in false measurement/test data, then the "impact of complexity of control" will be lower.

6. If the operator interface is complex, can this be assessed against the documentation? How important is operator training in this?

   In this case the *user* is in an ideal position to check the documentation independently. The *supplier* would no doubt welcome constructive feedback.

   If the operator interface is complex then this complexity should be included in assessing the complexity of control; if the complex operator interface can affect the measurement/test data then the "impact of complexity of control" will be high (i.e., modest or complex).

5.2 Computing the Measurement Software Index

The MSI is derived from the three software risks factors, once they have been assessed taking account of the questions above. The Index is a scale from 0 to 4, where increasing MSI corresponds to software of either increasing software complexity or increasing criticality of usage or both. As the MSI increases, so should the software risk of errors be reduced due the application of more rigorous software engineering techniques, which is the topic of chapter 7. We present a definition of the MSI by means of a table (table 5.1) where the axes correspond to the software risk factors of "Usage", "Processing" and "Control". If the software is also conform to IEC 61508 then see section 6.

Criticality of Usage	Software complexity				
	Complexity of Processing	Impact of Complexity of Control			
		Very Simple	Simple	Modest	Complex
Critical	Very Simple	0	0	1	2
	Simple	0	1	1	2
	Modest	1	1	2	2
	Complex	2	2	2	2
Business Critical	Very Simple	0	1	1	2
	Simple	1	1	2	2
	Modest	1	2	2	2
	Complex	2	2	2	3
Safety-indirect	Very Simple	1	1	2	2
	Simple	1	2	2	3
	Modest	2	2	3	3
	Complex	2	3	3	3
Safety-direct	Very Simple	2	2	2	3
	Simple	2	2	2	3
	Modest	2	2	3	4
	Complex	3	3	4	4

Table 5.1: Measurement Software Index, as a function of the software risk factors

The significance of the definition can be illustrated as follows:

Measurement Software Index 0. Each software risk factor is almost as small as possible: no significant problems are expected.

Measurement Software Index 1. Either at least one software risk factor is significant or more than one software risk factor is not as small as possible. This Index can arise with a system undertaking measurement in the safety context (e.g. potentially life-critical) when conformity to *IEC 61508* is required.

Measurement Software Index 2. When any one software risk factor is as large as possible then the MSI is at least 2. This Index can arise with a system undertaking measurement in a life-critical safety context, if the other software risk factors are not too great.

Measurement Software Index 3. When all software risk factors are almost as large as possible then the MSI is at least 3. This Index will often arise with a system undertaking measurements in the safety context when conformity to *IEC 61508* is required.

Measurement Software Index 4. This Index is reserved for safety systems that require the highest possible assurance. This Index is not fully covered in this edition of the Guide.

Once the *user* has determined the MSI, this figure should be agreed, if possible, with the *supplier*. The Index may be modified by consideration of some of the open questions, see section 5.1.

If the MSI is changed for other (business-related) reasons, then the agreed Index and the reason for the change must be recorded. If the MSI is deemed to be inappropriate, it may be necessary to revisit the software risk assessment and reconsider the values assigned to each of the three software risk factors.

Chapter 6

IEC 61508

In this chapter, we consider measurement systems which are part of a safety system which needs to satisfy the requirements of *IEC 61508*. We take this standard since it can be applied to any safety system and provides a framework which can be used for measurement systems. Any measurement subsystem must be shown to be fit-for-purpose within a safety system. It is unlikely that a measurement subsystem will be found to be acceptable unless it was designed with safety in mind in the first place. To aid tracing the relationship between this Guide and the IEC standard, references[1] to part 3 of the standard [*IEC 61508–3*] appear as marginal notes in this Guide.

6.1 Approach to *IEC 61508* for software

IEC 61508 is a *system* standard, whereas in this Guide we only consider software. This implies we can only consider aspects of the software required for the entire system to satisfy *IEC 61508*. For a simple introduction to *IEC 61508* see *FunctionalSafety* and *IGE-SR-15*, which has been produced by professionals in the gas industry.

This Guide does *not* cover the certification of systems for which detailed information about the development is not available. The use of proven-in-use systems is already covered by *IEC 61508* which may provide a means of demonstrating compliance with the standard. Failing that, two reports are available specifically on this topic which should be consulted, see *SOUP–1* and *SOUP–2*.

IEC 61508 defines four Safety Integrity Levels (SIL) and four Software Safety Integrity Levels.

It is necessary to determine the software safety integrity level of the measurement software. This is achieved as follows:

1. Determine the safety functions being performed, or dependent upon, the measurement software;

2. Determine the SIL of the safety functions;

3. If the software measurement system forms a part of the safety-related system then the Software Safety Integrity Level will be the same as the SIL for the relevant safety-function.

[1] These are referenced with an initial "3–" to distinguish them from internal references in this Guide.

SIL	*High demand rate (failures per hour)*	*Low demand rate (failures per demand)*
1	$10^{-6} \leq \ldots < 10^{-5}$	$10^{-2} \leq \ldots < 10^{-1}$
2	$10^{-7} \leq \ldots < 10^{-6}$	$10^{-3} \leq \ldots < 10^{-2}$
3	$10^{-8} \leq \ldots < 10^{-7}$	$10^{-4} \leq \ldots < 10^{-3}$
4	$10^{-9} \leq \ldots < 10^{-8}$	$10^{-5} \leq \ldots < 10^{-4}$

Table 6.1: Safety-integrity level targets

One purpose of this Guide is to contribute to the certification of measurement subsystems under the CASS scheme (see appendix B.1). Formally, this is undertaken by ensuring that the measurement subsystems satisfies the criteria for CASS subsystems specified in *CASS-subsystems*.

For the software issues, there is a vital aspect which should be determined as early as possible. The safety integrity requirement for the subsystem will partially determine the corresponding Measurement Software Index. This in turn will determine the validation requirements and hence the cost of obtaining certification or compliance to *IEC 61508*.

In the same way as with quality, blind conformance with a standard will not suffice. *IEC 61508*, as with other standards, does not include all aspects of every device or item of software within a safety system. Hence it is important that those responsible for safety systems take a broad view of the issues involved and do not just tick off the items listed in this Guide or within *IEC 61508*.

For safety systems, another standard contains useful insights that is particularly relevant to measurement systems. This is the consensus European nuclear regulator view of software issues [*EUR 19265*]. Even if nuclear applications of a measurement system are not envisaged, the document is recommended reading.

The overall approach within the Guide in its application to safety systems is shown in figure 6.1 (which is an extension of figure 1.1). The process is shown as six consecutive steps, but this is a simplification since it may be necessary to repeat an earlier step if a later one
3–7.7.2.2 shows a design problem. Specific issues that need to be addressed in the early design stages are as follows:

Robust design. A conventional instrument design may not be robust enough to cater for
3–7.1.2.3 operator errors or adverse environmental conditions. In consequence, the device may be inappropriate or require major changes.

3–7.2.2.5 **The right SIL.** The safety integrity requirement is critical. It is likely to cost at least twice as much to produce an instrument for the next higher SIL.

However, choosing a SIL that is to low to reduce costs could lead to inadequate validation for the intended usage.

The technological base. The technology used may not permit the validation of the instrument. For instance, an instrument's software may use an operating system which cannot be demonstrated to have the required safety integrity.

Is validation possible? The choice of methods chosen to validate the instrument may be inadequate for the safety integrity required. This problem is particularly difficult for measurement systems at the SIL3–4.

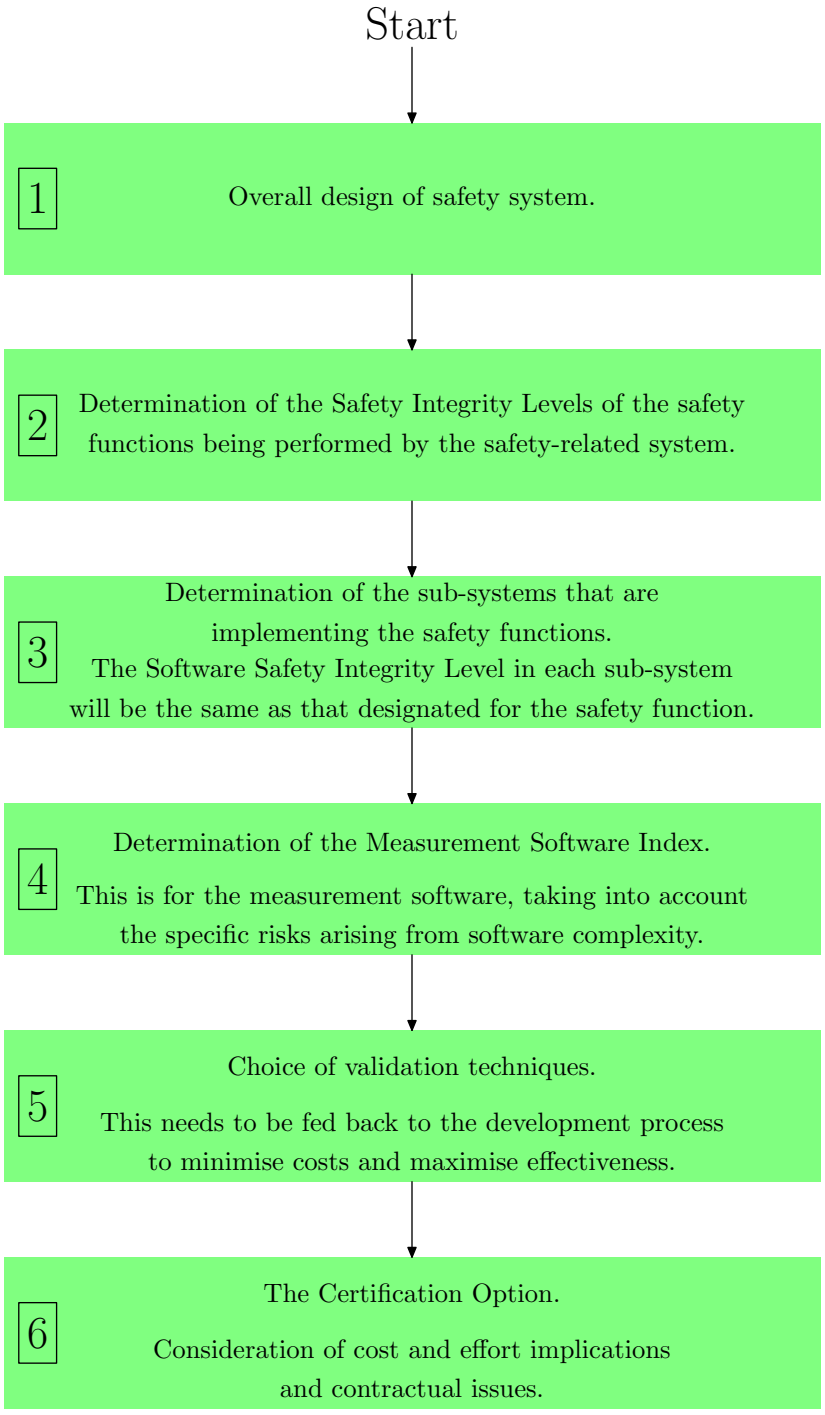

Figure 6.1: Stages in a Safety Validation

The certification option. If the user would like an instrument certified for use within a safety context, then the instrument supplier would need to know what is required and to be able to calculate the implications in terms of effort, cost and timescale.

Lack of evidence. *IEC 61508* requires extensive documentation (see appendix B.2) and it is unlikely that all of this would be available unless carefully planned in advance.

6.2 Measurement Software Index for safety systems

At first sight, it might appear that the SIL for the complete system would determine a safety integrity level for the subsystem and therefore the SIL could be mapped onto a suitable MSI. Using table 6.1, it may be felt that we can equate our MSIs with those of *IEC 61508*. This is not quite correct since the SIL applies to complete systems. In any case, one can envisage a mixed software system as follows:

> Suppose a measurement safety subsystem performs continuous measurement whose safety integrity requirements correspond to SIL2. However, the software actually consists of two parts: a relatively simple and straightforward part of the software which operates continuously providing the necessary measurement data, and also a calibration part which involves complex interaction with the
> 3–7.4.2.8 user. Assuming that the hardware must be restarted before and after calibration (say), then with some additional checks, it can be assumed that the two software components are sufficiently separated to be considered separately for validation. We now have two possibilities according to whether the calibration is a safety function:
>
> **No:** Here, the accuracy required for the safety of the system can be provided without calibration. We still need to ensure that either calibration is never performed or that it cannot reduce the accuracy of the instrument outside the safe limits;
>
> **Yes:** The calibration function must be included in the software validation process. This could be quite demanding.
>
> Note that the supplier would perhaps want to supply a single instrument, while from a validation perspective, another instrument with reduced accuracy and no calibration would be optimal for the safety market.

The software risk-based approach adopted here (see chapter 4), is not the same as the safety integrity approach used in *IEC 61508*. For this reason, the IEC method must be used to determine the SIL of the software first in order to ensure compliance with the IEC standard. Software of SIL 1 or SIL 2 would give criticality of usage of safety-indirect and software of SIL 3 or SIL 4 would give a criticality of usage of safety-direct. The complexity aspects of software risk can then be determined to derive the MSI as described in section 5.2.

The primary purpose of the MSI is to determine the most appropriate validation techniques to be applied. However, in the context of the software part of the standard [*IEC 61508–3*], there is a recommendation that certain techniques should be applied which we follow here (in *the technical part*). Also, the certification process requires objective evidence that certain validation techniques have been applied.

Given the need to certify a specific system, it may be that the case for compliance rests heavily upon the use of a software validation technique which is not handled in this Guide. It may be that certification via the CASS scheme cannot be undertaken or otherwise it may be possible to extend this Guide to cover the crucial technique. Many techniques in *IEC 61508* are to be recommended for software development, but play a much reduced role in software validation. In essence, the software developer may not be able to claim "certification credit" for such techniques. This is not to down-rate such techniques, but merely to avoid having to handle too many techniques which have a reduced bearing on the key issue: "is the system safe?". In other words, if you are using or depending upon validation techniques not covered in the Guide, then clearly the Guide is not going to help you.

Chapter 7

Software validation techniques

Given a **Measurement Software Index**, then the software risks can be mitigated by the application of suitable software validation techniques. If the *supplier* can assure the *user* that appropriate techniques have been applied, then the use of the measurement system in the agreed role can be justified. (For safety applications, the agreement would usually be based upon certification, see chapter 6).

7.1 Appropriate applicable techniques

The most appropriate techniques to apply depend upon the nature of the software, and are considered in detail in part II.

As a *user*, the validation techniques used by the *supplier* may well be unknown. In that case, the Guide can be reviewed by the supplier who can be asked to comment, given the results of the software risk assessment provided by the user.

A number of specific software techniques are described in part II. However, there is no overall framework for software validation techniques. Guidance for general software testing is given in *ISO 13233* which does provide a set of objectives such as repeatability, which could be developed further for measurement software. This would be a useful advance, since currently methods like Numerical Reference Results (see section 12.18) are only recommended in specific contexts like Coordinate Measuring Machines [*ISO 10360–6*].

7.2 Software and quality management

For any MSI, basic practices must be established which could be a direct consequence of the application of *ISO 9001* to software [*ISO 9000–3*] or from the application of other standards. It can be claimed that *ISO 9001* itself requires the application of appropriate (unspecified) software engineering techniques, especially for the higher Indexes. However, *ISO 9000–3* does not even mention many such techniques and hence we take the view that specific techniques should be recommended here, rather than depend upon the general requirements of *ISO 9001*. In the pharmaceutical sector, specific guidance has been produced which is effectively a version of *ISO 9000–3* oriented to that sector [*GAMP*].

In the UK, it is reasonable to expect *suppliers* to be registered to *ISO 9001*, which in the case of software should imply the application of TickIT (although note that the TickIT Scheme has now become a benchmark for any software certification scheme for *ISO 9001*,

rather than a software specific extension to *ISO 9001* registration; see *UKAS-TickIT*). If a supplier is *not* registered, then one lacks an independent audit of their quality system.

In practice, an important requirement of any audit or assessment is the provision of documentary evidence of the development and validation processes. A typical *ISO 9001* audit is based upon sampling and hence may not reveal the absence of documentation for a specific development.

ISO 9001 provides a basic standard for quality management, whereas in practice, companies will continually improve their system if the aim is high quality. In any case, improvements in the light of experience are essentially a requirement of *ISO 9001*. The standard implies a defined life-cycle which is elaborated in *ISO 12207*. For those companies not formally registered to *ISO 9001*, it is necessary that a similar quality management approach is used and evidence for that can be produced.

To conclude, *users* should expect *suppliers* to be registered to *ISO 9001*, or to be able to demonstrate compliance to a similar level of quality management.

Chapter 8

Conclusions

The attempt to answer the two questions posed at the beginning of the Guide is limited by the information available. One must accept that the *user* of a measurement system may not be able to obtain information from the *supplier* to determine if appropriate software engineering practices have been used.

If end-to-end testing cannot be regarded as providing sufficient assurance of the measurement system, then a user must consult the supplier to obtain the assurance that is desired.

It must be accepted that assurance levels higher than the majority of the users of a measurement system require can probably only be achieved by purchase of additional information from the supplier, perhaps even to the extent of developing a special-purpose measurement system.

In the case of measurement subsystems to be used with safety systems, assurance would be provided by adherence to *IEC 61508*. In this case, specific requirements for documentation are significant and, as the safety integrity requirements increase, so do the requirements for more rigorous validation of the software. The natural way for off-the-shelf measurement systems to be provided for the safety market is via a certification scheme since the expense of the validation of the instrument need only be met once, *provided that the use of the instrument is compatible with the scope specified in the certification.*

The approach taken in this Guide places great importance on the MSI. As this Index increases, so do the inevitable costs of validation. In consequence, great care must be taken to determine the appropriate Index — too high will increase the costs unnecessarily, and too low will not provide the required assurance.

Part II

Technical Application

Chapter 9

Understanding the Requirements

9.1 Technical requirements

The main non-technical aspects of the requirements are covered in the first part of this Guide. The purpose of this part is to provide additional information of the software engineering aspects so that high assurance can be provided by a *supplier* in a form which a *user* can appreciate. Hence this part is directed more at the supplier than the user.

In the context of safety applications, we want to specify a system for certification of measurement subsystems to *IEC 61508*. To ease the presentation to those not concerned with safety, much of the detail is placed in a special appendix (see appendix B). We exclude the more specialised area of Programmable Logic Controllers, since another guide is available covering this [*IEE-PLC*].

The general approach in Part I is followed here but with more detail. If high assurance is required, then the detail presented here is necessary. In essence, Part I only handles systems with a MSI of 0 (i.e., just the application of standard quality management).

The steps to achieve successful validation are illustrated in figure 1.1. The following points should be noted:

1. If as a *user*, your assessment of the software risk is higher than that envisaged by the *supplier*, then it may be difficult, or even impossible to obtain the assurance required.

2. As a *user*, you may be able to reduce the software risks by choosing a simpler system with only just sufficient functionality to cover the application.

3. The MSI is vital. Too low and the assurance is not obtained, too high and effort will be wasted.

4. The choice of validation methods will determine the cost.

In the context of certification to *IEC 61508*, particular safety integrity targets are specified (see table 6.1). This implies that the measurement subsystem *hardware* must be assured to satisfy these targets. This topic is outside the scope of this Guide, and readers are recommended to consult *FunctionalSafety* and *IGE-SR-15*. Clearly, if the hardware does not meet the required safety integrity level, the software safety integrity is irrelevant.

9.2 Handling calibration data

In most contexts, the handling of calibration data is a major concern. For instance, nuclear power station safety systems use the measurements from a large number of sensors, each
3–6.2.2 of which must be calibrated. Incorrect data could clearly have safety implications which therefore requires careful control. This Guide does not handle this issue specifically, and therefore readers are recommended to consult *EUR 19265*.

9.3 Ensuring a robust system

In any system in which there is a specific assurance requirement, which includes safety systems, it is important that every aspect of the measurement system should be analysed to ensure the target assurance is actually attained.

Aspects that need to be considered include the following:

1. Potential misuse by the user. This is a specific aspect of usability which is important for systems having a long life, see appendix A.6.
2. Device not installed correctly.
3. Device subject to extreme conditions. In software terms, this implies that the range of input values should be checked, and all values within the acceptable range should be processed adequately.
4. Memory not adequately protected from external interference. In some environments, sum-checks should be applied to volatile memory, for instance.
5. Invalid measurement request — say, an attempt to measure a sample when no sample is present.

Note that a lack of a robust design implies that the measurement request may fail and this, in turn, has implications for the design of the system using the measurement. In some cases, the software may be able to detect a malfunction of the basic measurement hardware, while in others it may be necessary to consider a redesign of the hardware.

Chapter 10

Issues in Software Risk Assessment

Without a software risk assessment one cannot be confident that any effort spent in validation will be worthwhile. Clearly, if one can be assured that the software within a measurement system can have no negative impact on the resulting measurements, then all that is needed is to record the reasons why this conclusion has been reached. On the other hand, if the software is large and complex, and also performs an essential function in producing the output from the measurement system, and that output has a safety implication, then validation is essential.

The software risk assessment should identify the key aspects so that the validation process can target these important aspects. Almost all complex software systems contain a high proportion of less critical code which could be checked by simpler and cheaper methods than the more critical parts. (However, the parts must have sufficient independence, otherwise a gross programming error can easily overwrite critical data.)

Measurement is a numerical process. In consequence, one must be confident that the software processing of the numeric data produces a meaningful result. Techniques are available to ensure the numerical correctness of the computations within the measurement system, which should therefore be applied when the assurance they provide is required. (See Numerical Reference Results, section 12.18, and Numerical Stability, section 12.9.)

10.1 Software Risk factors

The fundamental problem for the *supplier* is that the criticality of the usage of a measurement system may be unknown and, therefore, a specific Index must be chosen. If a *user* of a measurement system wishes to use it in a more critical application than the supplier has envisaged, then additional validation effort may be required, which can involve much analysis work (especially with safety systems). It may prove impossible to obtain the necessary assurance in some circumstances which may therefore require an alternative measurement device. For safety systems, it may be that the most effective solution is a change in the system architecture.

The issues which need to be considered are collected together here as questions. Not all of these will be relevant for a specific measurement system.

1. For the *user*, what is the criticality of usage?

critical / business-critical / safety-indirect / safety-direct

2. Is the *supplier* aware of the criticality of your application?

3. Are there specific legal requirements for the measurement system?

4. What are the consequences of a measurement system malfunction?

5. Is independent evidence needed of the software development process?

6. Does the measurement system require regulatory approval?

7. What is the complexity of control?
   very simple / simple / modest / complex

8. Does the measurement system perform built-in testing? Is this sufficient for this application?

3–7.2.2.7 9. Do the control functions protect the operator from making specific errors? Is this sufficient for this application?

10. Can an error in the control software cause an error in the basic test or measurement data?

11. Is the raw data available from the measurement system?

12. Does the measurement system contain local data, such as that derived from the last calibration?

13. Is the processing of the raw data strictly linear?

14. Is the processing of the raw data a simple non-linear correction?

15. Is the processing of the data restricted to published algorithms?

16. Have the algorithms in use been checked for numerical stability?

17. Would a numerical error, such as division by zero, be detected by the software, or would erroneous results be produced?

    This will typically depend upon the programming system used to produce the software, and can vary from no detection of such errors to elaborate indications of the exact point of failure. If no internal checks are applied, there is a greater risk of a programming error resulting in erroneous results. See also section 11.2.

One should not necessarily regard performing a software risk assessment as a once-only function. Given an initial assessment, some validation work may be undertaken. As part of that validation, more information will be available which may imply that the software risk assessment should be revised. (In the case of bespoke systems, feedback from actual use would allow for a continuous process of improvement.)

10.2 A simple example

A platinum resistance thermometer has a small amount of embedded software, mainly to provide a convenient interface for the user. The basic algorithm is to apply a small non-linear correction derived from the last calibration of the measurement system.

Assuming the application is not **safety-direct**, and that we have assurance that the basic algorithm is as described above, then it would be reasonable to conclude that end-to-end testing of the measurement system as a whole will provide the assurance needed. In consequence, no specific software validation process is required. The assurance provided by the *supplier*, working under an appropriate quality management environment, should be sufficient. 3–7.7.2.6

For another, full example in which end-to-end testing provides high assurance, see appendix C.2. (This is an electrical bridge in which the data processing software is very similar to that of the simple example here.)

10.3 A complex example

Consider the example of the Autoclave in section 4.4. At first sight, the pharmaceutical company had every right to trust the Autoclave manufacturer to satisfy the requirements. However, FDA required visibility of the evidence which should support the correct operation of the Autoclave.

In undertaking the retrospective validation, errors were found in the computation of the (agreed) formula. Fortunately, the error was on the side of safety, but it seems clear that the error could have been unsafe so that the Autoclave did not sterilise the artefacts.

In terms of this Guide, the computation was **modest** since it involved integration over time. In this case, the errors in the clock compounded the problem.

The measurement of time and temperature to the accuracy required by this application is not demanding. In consequence, the trust placed by the pharmaceutical company seems reasonable. The observed errors in this case are therefore disturbing, especially since the correct operation of an Autoclave is almost certainly a safety factor.

The conclusion must be that even in cases in which the *supplier* could reasonably be expected to satisfy the requirements, *the user should ask for documentary evidence of this.*

Chapter 11

Software Reliability Requirements

The starting point here is that the software development process being used should have a rigour to match the MSI. This is the approach taken in several safety critical software standards [*IEC 61508, DO-178B*].

Theoretically, it is possible to undertake the testing of software to establish the actual reliability of the software. However, there are strict limits to what can be achieved in this area [*Littlewood*], and hence the approach taken here is the conventional one of examining the software development and validation process, *but only as far as the impact on the product is concerned*. In practical terms, software testing is expensive, and hence the most cost-effective solution is likely to be to minimise the testing and use other methods to gain confidence in the software.

11.1 Software and quality management

Following the outline in Part I (section 7.2), we assume that either *ISO 9001* or a similar quality management system is being applied by the *supplier*. The application of such a quality management system to software should imply that a number of technical issues have been addressed and documented and that the following requirements are met: 3–7.1.2.2

1. There should be documents demonstrating that a number of issues have been covered such as: design, test planning, acceptance, etc. The acceptance testing should ensure that the operator interaction issues have been handled and validated.

2. There should be a detailed functional specification. Such a specification should be sufficient to undertake the coding. This level of information is typically confidential to the software developer. (The developer and the supplier could be separate organisation, but since the responsibility, as far as the user is concerned, lies with the supplier, in this Guide we usually refer to the supplier.)

3. There should be a fault reporting mechanism supported by appropriate means of repairing bugs in the software. 3–7.4.7.4 3–7.4.8.5 3–7.5.2.6

4. The software should be under configuration control. This implies that either the software should itself include the version number, or the version can be derived from 3–7.5.2.8 3–7.7.2.5

other information, such as the serial number of the device. In the case of free-standing software, it should be possible for users to determine the version number. In may also be necessary to roll-back to a previous version.

We assume that these requirements are met, whatever MSI is to be addressed by the software. We equate these requirements with a MSI of zero.

In *EUR 19265*, three aspects are considered for which evidence is required for nuclear safety systems. These are listed below to contrast with the approach taken in this Guide in which none of these are explicitly covered:

The development process. In this Guide, evidence is required of *ISO 9001* or an equivalent quality management system and the ability to apply effectively the recommended validation techniques. For safety systems, the documented evidence of the design, development and validation is vital.

Adequacy of the product. In this Guide, this is covered by specific recommended validation techniques.

Competency of the staff. This is not currently covered in this Guide, but needs to be included in the certification process, almost certainly by applying the IEE/BCS scheme [*CompetencyStandards*].

11.2 Requirements for the development environment

Any potential defect in the development environment could result in undetected errors in
3–7.4.4.3 the delivered software. In almost all the measurement systems being considered here, the key component of this environment is the programming language and associated tools.

11.2.1 Development environment issues

In this section, we list nine common problems which can easily undermine the development process. We then recommend a documented means of overcoming these problems which needs to reflect the overall **MSI** required.

The nine problem areas are as follows:

Source text control. This is a requirement of *ISO 9001*. Several tools are available to track changes to key files. However, the major issue is that of ensuring that the results
3–7.4.4.6 of any validation applies to the system as delivered. If a validation method detects a fault, then correction of that fault implies that the validation must be repeated, including potentially, many other validation techniques.

Source text control is *very* important for systems which are being continually updated. Updates are often undertaken with fewer people than the original development, in which case, some processes involved in building a new system might be omitted.

Tool control. Many tools have several options, and ensuring that the "correct" option
3–7.4.4.2 is always used is important. This a vital with compilers which often have means of varying the checking performed (both statically and at run-time).

Coding errors. Most coding errors will be detected by the compiler or by testing. Hence
3–7.4.4.5 consideration needs to be given to any coding errors which could remain undetected by the routine development process. Code review or walk-throughs should be used.

Unset variable. This is probably the most common programming error. Requiring that all variables are explicitly initialised on declaration is not usually a good solution, since such a value may not be logically correct. Implicit initialisation is also not recommended, but this may be automatic in some programming language environments.

Range/index error. In most measurement software, the physics implies a specific range of possible values. In addition, the programming language will require that an index used to access an array element is within the bounds of the array.

Numeric error. Numeric underflow and overflow of floating point values is considered separately (see sections 12.9 and 12.18). However, integer overflow can also arise and often will not be detected.

Pointer error. If the programming language provides pointers which can be directly manipulated by the programmer, then pointer errors can easily arise and be hard to detect and trace. Hence, in this context, it is vital to have techniques in place to avoid or manage the issue.

Other language errors. Apart from the errors already noted, the programming language will often list situations in which the program will not behave in a predictable manner. These situations must be understood and procedures put in place to ensure that they are dealt with.

Timing error. While the main processor will almost always operate in an asynchronous matter, this is unlikely to be true of the external environment. It is therefore important that any timing characteristics of the basic sensors (say) are taken into account. If the programming language provides a means of running more than one task at a time, or it is necessary to handle interrupts, then timing problems can easily arise.

All but the first two issues are associated with the execution of the software — although detection before execution is highly desirable. The detection of these remaining conditions could be as follows:

Run-time. In some contexts, it could be sufficient to detect the problem during the execution of the measurement system. If which case, building the systems with the inclusion of run-time checks will be acceptable. 3–7.4.2.10

Component test. During development, run-time checks are enabled during component testing. The faults found are corrected. Hence, unless the complete system executes a component in ways not allowed for during component testing, the problem will have been resolved.

System-test. The system-testing is undertaken with checks enabled. The problem here is that ensuring that all the paths through the software are covered is much more difficult than with component testing.

Regression test. A special suite of tests is used, which reflect experience with previous releases of the system. Typically these are run as near to the final system as possible, but with built-in checks so that a failure is detected.

Manual check. This can be very labour-intensive and it is hard to quantify the success achieved.

Tool check. A static analysis tool is used to analyse the source code. There is a substantial variation in the way tools work with respect to the amount of manual analysis that needs to be done after the application of the tool.

11.2.2 Illustrative example

We consider a high safety integrity, probably MSI4 system, being developed using the Spark subset of Ada 95 [*ISO 15942*]. The Unix tool for source text control is used and all the main tools are called by means of project-defined macros. This effectively handles the issues not associated with the execution of the application.[1]

In this application, the detection of a fault at run-time is not considered appropriate as the safety integrity of the total system could be compromised. In consequence, the emphasis is upon the use of the Spark tool to show the absence of run-time errors (Ada exceptions). Component testing is used primarily to show adherence to the specification of the component, rather than detection of any remaining run-time faults.

Error	*Run-time*	*Component-test*	*System-test*	*Manual-check*	*Tool-check*
Coding	N/A	Yes	No	Yes	Spark (90%)
Unset variable	No	No	No	No	Spark (100%)
Range/Index	No	No	No	Yes	Spark (97%)
Numeric	No	No	No	Yes	Spark (97%)
Pointer	No	No	No	No	Spark (100%)
Other	No	Yes	Yes	90%	
Timing	No	No	No	No	Spark (100%)

Table 11.1: Error detection

In order to justify the safety integrity claims, we need to make crude estimates of the detection of the faults listed above by the means used for the project. This results in table 11.1.

It is clear that the "gap" in the error detection by the language environment is in the "other language error" category which in this case is from floating point overflow and underflow. This is handled by classical numerical analysis, see section 12.9.

11.3 Recommended techniques

In table 11.2, we list the recommended techniques, which are all defined in chapter 12.
3–7.1.2.6 Techniques for MSI4 are only suggestions in this edition of the Guide.

Note Independent audit (section 12.1) is recommended at MSI0, so does not appear in the table, but may be applied at all Indexes.

Regression testing is not included in this table since its use if not dependent on MSI but rather on the frequency of new releases and the amount of functionality that remains unchanged between releases.

The table does not mention those techniques which are described but not recommended (at any Index) in this edition of the Guide.

When selecting techniques, it is important to obtain a balance between *testing* and *analysis* methods. The table lists the analysis techniques first, and the testing techniques last. A technique like "code review" is essentially an analysis of the implementation, while "review of specification" is an analysis that does not logically require an implementation.

[1] This example is based upon, but not the same as *PROOF*, which is a complete system developed using SPARK.

Ref.	*Recommended technique*	*Measurement Software Index*			
		1	2	3	4
12.2	Review of informal specification	Yes	Yes		
12.3	Software inspection of specification		Yes	Yes	
12.4	Mathematical specification	Yes	Yes	Yes	Yes?
12.5	Formal specification				Yes?
12.6	Static analysis		Yes	Yes	Yes?
12.6	Boundary value analysis		Yes	Yes	
12.7	Defensive programming	Yes	Yes		
12.8	Code review	Yes	Yes		
12.9	Numerical stability		Yes	Yes	Yes?
12.10	Microprocessor qualification				Yes?
12.11	Verification testing			Yes	Yes?
12.12	Statistical testing		Yes	Yes	
12.13	Structural testing	Yes			
12.13	Statement testing		Yes	Yes	
12.13	Branch testing			Yes	Yes?
12.13	Boundary value testing		Yes	Yes	Yes?
12.13	Modified Condition/Decision testing				Yes?
12.15	Accredited testing		Yes		
12.16	System-level testing	Yes	Yes		
12.17	Stress testing		Yes	Yes	
12.18	Numerical reference results	Yes	Yes	Yes?	Yes?
12.19	Back-to-back testing		Yes	Yes	
12.20	Source code with executable				Yes?

Table 11.2: Recommended techniques, *see text for selection.*

All techniques should be *considered*. For instance, if a "mathematical specification" has not been produced, then the implications of that need to be taken into account. (For an example without a mathematical specification, see the analysis of images in appendix C.3.)

Ensuring that numerically sound results are produced is clearly vital. In consequence, in all but the simplest of calculations, one must gain assurance by using only stable algorithms (i.e., ensuring the design is correct, see section 12.9), or by checking the implementation using numerical reference results (see section 12.18), or by testing the implementation against a reference implementation (using back-to-back testing, see section 12.19). (Very high assurance would require more than one technique.) Since we know of no other appropriate methods of gaining assurance for numerical calculations, one of these techniques should be used.

The fact that a technique is recommended at a specific Index does not (in general) imply that not applying the method would imply poor practice or that all the methods should be applied. For instance, the example given under Accredited testing (see section 12.15) is a good choice precisely because other strong methods are not effective. Any design should involve a trade-off between the various relevant methods.

In the context of certification to *IEC 61508*, the techniques recommended here should be applied, or a reason produced as to why any errors that are likely to be detected would be found by techniques which have been applied (and are also recommended here).

There is one important factor that should be taken into account in determining the techniques that should be used. In performing the software risk analysis, it was noted that

the complexity of the software increases the software risk significantly (see chapters 4 and 10). Hence, in the case of *complex* software, almost all of the recommended techniques should probably be used. Conversely, in the case of very simple software, the number of techniques used could be reduced. Note that therefore the cost of validation can be expected to increase more than linearly with the size of the software.

Note that Structural testing, Statement testing, Equivalence partition testing, Boundary value testing and Branch testing are all (software) component test methods (see section 12.13).

11.4 Software development practice

The issues here are again listed as questions. The first set are generic to all Indexes, and the others are specific to a given Index. The lists increase in complexity with increasing **MSI**. Since the requirements at Index n ($n > 0$) imply those at Index $n - 1$, all the questions should be asked up to the Index required.

More detailed questions which are designed for auditing, appear in appendix B.3.

11.4.1 General software issues

1. What information is available from the measurement system supplier or developer of the software?

2. What confidence can be gained in the software by end-to-end tests on the measurement system?

3. Can the raw data be extracted from the measurement system?

4. Can the raw data be processed independently from the measurement system to give an independent check on the software?

5. Are software reliability figures available for a measurement system using similar software (i.e., produced by the same supplier)?

6. Is a log available of all software errors?

   Has this log been inspected for serious errors?

7. Does the control function have a direct effect on the basic test or measurement data?

8. Has an assessment been made of the control software against the operating manual? If so, by whom?

9. Do operators of the measurement system require formal training?

10. Have all the answers to the questions above in this list been taken into account in determining the MSI?

11. Has a list been made of all the unquantifiable aspects of the software?

11.4.2 Measurement Software Index 0

1. Is there design documentation?
2. Is there evidence of test planning?
3. Is there a test acceptance procedure?
4. Is there an error log?
5. What evidence is there of clearance of errors?
6. Is there a detailed functional specification?
7. Are security, usability and performance aspects covered in the specification?
8. How is configuration control managed?
9. Is there a defined life-cycle?
10. How can the user determine the version number of the software?
11. Have all changes to: hardware platform, operating system, compiler, and added functionality been checked?
12. Have all corrections been checked according to the defined procedures?
13. Have all the staff the necessary skills, and are these documented?

11.4.3 Measurement Software Index 1

1. Has a review of the informal specification been completed?
2. Has a mathematical specification been produced of the main algorithms used for processing the test/measurement data?
3. Is the processing code derived directly from the mathematical specification?
4. Are all possible paths in the software either handled or covered by defensive programming checks?
5. Has a code review been completed?
6. What form of structural testing has been applied, and what metrics of the level of testing have been produced?
7. What level of testing has been applied to the control and processing components of the software?
8. How has the completeness of the system testing been assessed?
9. Has the system testing covered all reasonable misuses of the measurement system?
10. What records are available on the system tests?
11. Are the security features consistent with any regulations or intended use?
12. Are the test strategies, cases, and test completion criteria sufficient to determine that the software meets its requirements?
13. Are the key metrology algorithms checked by means of numerical reference results?

11.4.4 Measurement Software Index 2

1. Has an independent audit been undertaken?

   Have all problems identified been resolved?

   Did the audit apply to the basic quality management system or to the software techniques as well?

2. Have activities been undertaken in the development which are not auditable (say, no written records)?

3. Unless software inspection has been applied, has a review of the informal specification been successfully completed?

4. Assuming software inspection been used on the project, what documents has it been applied and what was the estimated remaining fault rate?

5. Has any form of static analysis been applied to the software? Note that this covers boundary value analysis.

6. Has the numerical stability of the main measurement/data processing routines been checked?

   Has a floating point error analysis been taken into account in formulating the mathematical specification of the routines?

7. Has any form of statistical testing been applied to the software?

8. For what components has equivalence partition testing been applied?

   Has the technique been applied to the components processing the basic test or measurement data?

9. To what components has statement testing been applied? What coverage was obtained?

10. Has boundary value testing been applied to key components?

11. Has regression testing been applied?

    If so, at what point in the development did it start?

12. Does accredited testing have a role in gaining confidence in the software?

    If a test suite is used for accredited testing, have all the results of other forms of testing been fed into this?

13. Has stress testing been applied to the software?

    To what extent have the limits of the software been assessed by this testing?

    Has the stress testing revealed weaknesses in the system testing?

14. Could back-to-back testing be applied to the software to compare it to some equivalent software?

15. Are known remaining software bugs documented?

    Are they adequately communicated to the user?

16. What methods have been applied to avoid structural decay?
    (See appendix A.3.)

11.4.5 Measurement Software Index 3

1. What forms of static analysis have been undertaken?
2. Has verification testing been performed with the appropriate degree of independence and level of effort?
3. What coverage was obtained with branch testing?

11.4.6 Measurement Software Index 4

To be considered in detail in a later edition of the Guide.

1. Has a Formal Specification been produced of any of the software components?

   Has this revealed weaknesses in the functional specification, testing, etc.?

   Is it possible to derive an executable prototype from this specification to validate the equivalence partition testing?
2. Has the microprocessor in use been suitably qualify?
3. Has the source code been compared with the executable?

11.5 A simple example revisited

We reconsider the simple example mentioned in section 10.2. Having undertaken the preliminary analysis, the instrument is to be regarded as MSI1, and complete access is available to the design and coding of the system. We assume that the basic hardware satisfies the required safety integrity target.

The instrument has calibration constants stored in non-volatile memory. However, an analysis showed that if the user made a mistake during calibration, the calibration constant could fall outside the allowed range. Hence the supplier added sanity checks within the calibration software.

The analysis of the instrument showed that there were only 2^{16} possible data input values. Hence exhaustive end-to-end testing was agreed to be adequate in this case.

Note that in this example, we have used an analysis of the software to show, in effect, that we can treat the device much like hardware, *but this is sufficient only because of the extreme simplicity of the software.*

11.6 A pressure transducer

We now consider a pressure transducer very similar to Druck's device mentioned in section 3.6. We assume that the instrument is required to satisfy MSI2, and that it needs to have a safety integrity better than one failure per 10^6 hours of continuous operation (this corresponds to SIL2 of *IEC 61508*, see table 6.1).

Although the software is not complex, there is no possibility of validating the software comprehensively according to this Guide, since the safety integrity requirements are quite demanding. Indeed, we cannot test the system directly to measure safety integrity, since the mean time between faults should exceed 110 years.

Hence we consider each of the recommended techniques:

Review of informal specification. Undertaken using an independent consultant, who advised some changes which were applied.

Software inspection of specification. Not done as the review above was thought sufficient.

Mathematical specification. Done, and checked as part of the review.

Static analysis. C code checked as complying with the standard. Manual checks applied to ensure code would execute correctly.

Boundary value analysis. Done as part of component testing.

Defensive programming. Run-time checks added as a consequence of the review.

Code review. Done.

Numerical stability. Done by a qualified numerical analyst.

Statistical testing. No adequate information available on usage to allow such testing to be meaningful — not done.

Statement testing. Done, with the exception of defensive checking code.

Boundary value testing. Done as part of component testing.

Accredited testing. No available service for such a device.

System-level testing. Done.

Stress testing. Not done.

Numerical reference results. Not done, since the checks on numerical stability were thought to be adequate, and no such numerical results available for this application.

Back-to-back testing. Not done as there was no available system to compare with.

Hence, of the 16 techniques, 10 were undertaken. The manual checks for correct execution were thought to be adequate since the code was small. Since the system was certainly not complex, and there was a reasonable balance of static and dynamic techniques, the conclusion from validation was positive.

With any system, it is worthwhile to consider what errors might remain. In this case, the basic microprocessor used does not have floating point hardware. Since the C code uses floating point, it is possible that the subroutines added by the compiling system have some errors which were not detected during testing. This is a question of Software of Uncertain Pedigree [*SOUP–1, SOUP–2*]. This is a difficult issue, in general, although in this specific case, tests are available to validate basic numerical operations [*FloatingPointValidation*].

Chapter 12

Software Validation Techniques

Here we list specific recommended techniques. These are either defined here, or an appropriate reference is given. A good general reference to software engineering is *SERB*. The techniques are ordered by their use within the life-cycle of software development. This Guide only considers *validation* and therefore techniques which are appropriate for development are not necessarily considered.

In the context of safety systems, it is very important to use a mixture of techniques which, when combined, give a high assurance of the suitability of the system. The mixture needs to include *dynamic* methods which execute test cases, and *static* methods which depend upon an analysis of the design. For further information, see Appendices A and B of *IEC 61508–3*.

12.1 Independent audit

In the UK, independent audit to *ISO 9001* is widely established. This provides evidence to third parties of a **MSI** of 0. It would not require that the stronger (and more expensive) techniques are applied, nor that the recommendations here are applied. In consequence, auditing to comply with the other standards mentioned in chapter 2 would be better.

Example

NPL, and many other organisations, are independently audited to *ISO 9001*. In the context of measurement (and testing), assessment to *ISO/IEC 17025* is probably more relevant.

Scope

The scope of this technique is universal, but such auditing/assessment is not as widespread in the USA as in many other developed countries. This implies that assurance via independent audit may not be effective for some imported measurement systems.

Safety implications

Independent audit (for functional safety) is a requirement of *IEC 61508*. When software is a key component of the safety system, auditing of the software development process is to be expected. This Guide assumes that such an audit is undertaken.

The CASS scheme provides a means to gain certification of a organisations management processes to *IEC 61508* through Functional Safety Capability Assessment (FSCA), see section B.1.

Cost/Benefits

The cost of auditing and assessment depends largely on the number of man-days of auditing effort. The benefits are in a similar proportion.

Conclusions

If you have any doubts about the standing of a measurement system supplier, you should ask about registration to *ISO 9001*. If the supplier calibrates any measurement system, including their own, then they should be assessed independently to *ISO/IEC 17025* or an equivalent standard. As a user, you may be able to calibrate the measurement system yourself, but nevertheless, the supplier should be able to demonstrate this capability too.

12.2 Review of the informal specification

It is widely thought that errors in the specification of software are the most common and most expensive type of error in software. In consequence, techniques which are likely to detect faults in the specification prior to investing effort in the implementation are very worthwhile.

Some form of review should always be undertaken — hence the main issue is to decide the nature of the review and the level of effort that should be devoted to it.

Unfortunately, a long and detailed specification is hard to check completely before implementation. Some aspects of the specification may only become clear when part of the implementation has been completed. On the other hand, with some measurement systems, the mathematical formulae giving the relationship between the basic sensor outputs and the resulting measurement gives an effectively complete specification. Hence software developers should assess the potential risks of an error in a specification and direct the review effort appropriately.

Individuals undertaking a review should be independent[1] of the writers of the specification, but have the necessary expertise to criticise the specification. If the software specification is written by a physicist (say), but is to be implemented by a software engineer (say), then appropriate checks should be made that the specification reflects the need to bridge the two disciplines.

When a detailed review is being considered, the Software Inspection method may well be appropriate, see section 12.3.

Scope

This technique is applicable to all systems.

Safety implications

A review of some type for the complete system is the requirement of *IEC 61508* (Part 2, 7.4.2.6 for the software requirements). A review which is not a software inspection is

[1] Safety standards typically define the degree of independence required.

recommended at MSI1 and MSI2. Further guidance in the context of the certification to *IEC 61508* is given in section B.3.1.

Cost/Benefits

It is probably unwise to spend less than a third of the cost of writing the specification on a review. A well-focused review is likely to be highly advantageous in ensuring a quality product.

Conclusions

Some kind of review of the specification should always be undertaken; if the specification is informal then an informal review, as described here, is recommended. The review should focus on areas of risk. At higher MSIs the review should be more formal, such as software inspection, mathematical specification review, or formal specification review; as described in the following sections.

12.3 Software inspection

This technique is a formal process of reviewing the development of an output document from an input document. It is sometimes referred to as Fagan inspection. An input document could be the functional specification of a software component, and the output document the coding. An excellent book giving details of the method and its practical application is given in *SoftwareInspection*.

The method is not universally applied (in industry), but many organisations apply it with great success. It tends to be applied if the organisation has accepted it and endorses its benefits.

Scope

The requirements are an input document, an output document and terms of reference for the inspection. As such, the method can be applied to a very wide range of documents, and hence to the processes producing such documents.

However, the method is only effective if used on a regular basis within an organisation. People need some training in the technique in addition to reading the reference given (or similar material).

Safety implications

Applying software inspection to the specification of the software is a recommended technique for MSI2/MSI3 systems. In the context of safety systems, auditing recommendations are given in section B.3.2. Note that a software inspection is regarded as being a *stronger* technique than review and hence is recommended at MSI3 while review is not.

Cost/Benefits

The process requires at least three people in an inspection. Experience shows that one needs to allow at least half an hour per page, and some preparation time and time to write up the results of the inspection. Hence the cost is probably twice that of a single person undertaking a review — the alternative which is probably more widely used in industry.

The advantage is that after a successful inspection, one has more confidence in the output document than is attained by an individual review. Specifically, the result produces a document that is very likely to have consensus support within the organisation.

Conclusions

Organisations should consider applying the technique on a regular basis if it is felt that this depth of review of documents is advantageous.

For those organisations who already apply the technique, it is recommended that they review the proportion of critical documents that are inspected. Internal quality procedures and records should indicate the areas where the main gains are to be found.

12.4 Mathematical specification

A mathematical specification gives the output data values as a function of the input data values. This method is suitable for the simpler processing of basic measurement data, and should clearly be expected. The mathematical function may well not be the way the actual output is computed, for instance, the specification may use the inverse of a matrix, while the results are actually computed by Gaussian elimination. This method should avoid a common error of not specifying the exact effect of the end of a range of values. It is not easy to apply the method to digital images (say); since the algorithms applied are quite complex, any "complete" specification is likely to be very similar to the software itself.

The mathematical specification needs to be validated against the underlying physics. This includes establishing that the model describes the system sufficiently well and ensuring that the errors introduced by the system are fully understood.

In some cases, the mathematical specification could be based upon an empirical relationship rather than directly upon the underlying physics. The validation of this empirical relationship has been covered in the Modelling Theme [*SSfM-Model*] of the SS*f*M programme [*SSfM–2*].

Example

See *SmartInstrument* for an example of mathematical specification applied to a pressure transducer. In that case, the mathematical specification has been analysed to show the accuracy of the computation in the implementation.

Scope

> *I often say that when you can measure what you are speaking about and express it in numbers you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of* science*, whatever the matter may be.*
>
> Lord Kelvin, Lecture on Electrical Units of Measurement, 3rd May 1883.

This quotation expresses the universal nature of mathematical specifications in the area of measurement.

Safety implications

Although *IEC 61508* does not mention mathematical formulae, it is clear that, in the context of safety related measurement systems, a precise specification is vital. Hence, such a specification is recommended at all Indexes. Auditing recommendations are given in section B.3.3. 3–7.4.2.5

Cost/Benefits

Given its universal nature, the only question is the rigour with which the method is applied. For linear systems, supported by calibration, perhaps little needs to be done.

Conclusions

Apply in all cases in which the mathematical formulae are not obvious.

12.5 Formal specification

Several methods are available for providing a specification in a completely formal way which can handle most functional aspects of a specification. The best known methods are VDM [*VDM-SL*] and Z [*Spivey-Z*]. For the primary author's personal views of this technique, see *FormalMethods*.

Example

The use of such techniques is more frequently associated with safety and security systems than metrology. However, an example of such a specification involving measurement is from a system to alleviate the effects of gusts on an aircraft [*AttitudeMonitor*].

In this application, three basic measurements are combined and then, if the alleviation mechanism is invoked, the Flight Control System (FCS) must be informed to take the necessary action. The key issue here is that the FCS is **safety-direct**, and therefore, by association, so is the gust alleviation system. Moreover, the logic is complex enough to demand very careful specification which, in this case, is achieved by the application of the Z language.

Scope

Formal specification languages have wide applicability, but are most effective when used to specify complex discrete systems. If all the relevant parties can read such a specification, it substantially reduces the risk of an ambiguity resulting in an error in the software.

Safety implications

This method is certainly relevant to MSI4 systems, but no additional guidance is given in this Guide (although it may be available in a later edition). 3–7.4.2.5

Cost/Benefits

Effective use requires people trained in the use of the technique being employed (such as VDM or Z). This could mean high additional costs for a project, which implies using Formal Methods only on the most critical systems.

Conclusions

If a system has a large number of discrete states, all or most of which could influence the measurement results, then it is unlikely that sufficient assurance can be gained from testing alone. In such cases, the use of a Formal Specification should be considered.

12.6 Static analysis/predictable execution

This technique determines properties of the software primarily without execution. One specific property is of key interest: to show that all possible executions are predictable, i.e., determined from the semantics of the high level programming language in use. Often, software tools are used to assist in the analysis, typically using the programming language source text as input. In general, the analysis techniques employed range from the very simple (say, all variables are explicitly declared) to the very strong (formal proof of correctness), but the goal of showing predictable execution should be cost-effective for high Indexes of safety integrity software. In essence, this technique performs a complete analysis of the code to detect the errors noted in section 11.2.

The use of this technique does have implications on the design methods used. The requirement of the nuclear guidance [*EUR 19265*] reflects this: *The design shall be deterministically constrained in ways which make the execution of the software verifiable and predictable.*

An important aspect of static analysis is to ensure that the boundaries between different paths taken by the algorithm are correctly placed. This is *boundary value analysis*, and can reasonably be undertaken as part of static analysis. This is recommended at MSI2/MSI3.

For a general discussion on static analysis, see *StaticAnalysis*.

Example

An example of this technique is to be found in the *SmartInstrument* project. NPL analysed the C code within Druck's pressure transducer using the NPL C code validation service tools.

The code was first converted to work in a free-standing manner on a Linux system. Then the code was compiled with a special compiler which warns of any non-standard (or potentially non-standard) constructs. Lastly, the code was executed with test cases designed to ensure statement coverage and boundary value testing. A few problems were located which were either corrected, or shown not be of concern in this application.

This form of testing is a comprehensive method of demonstrating the correctness of the source code of an application. It will not locate errors in the specification of the software, nor errors which an incorrect compiler could introduce.

Scope

All software written in languages with precisely defined semantics.

Safety implications

This technique helps to ensure that the software performs the intended function. It is very
3–7.4.4.5 useful in conjunction with conventional testing, since it can reveal defects in software which would be hard to demonstrate with specific test cases. This technique is recommended at MSI2 and above, see section B.3.5.

Cost/Benefits

Effective use depends upon appropriate tools. Such tools exist for C. For Ada and Java, most of the checking is undertaken by the compiler. Comprehensive checking of C++ code is not within the state of the art. Most proprietary languages have no static analysis tools.

Conclusions

Consider the application of the method when the source text is available and high assurance is required.

12.7 Defensive programming

Defensive programming is essentially a design technique but is included here due to the impact upon validation. Coding of a component part inevitably depends upon assumptions made about the program state when the component is called. If such a pre-condition is not satisfied, then the program could behave in unpredictable ways.

One method that partly avoids such problems is to test dynamically that the pre-condition is satisfied, producing an appropriate indication if it fails. Another method is to *prove* by static analysis of the program (see section 12.6) that the pre-condition cannot be violated.

For measurement systems, it is usually possible for no measurement result to be produced — hence one has an alternative when such a pre-condition check fails. Simpler systems may not have a means of producing a diagnostic message indicating the reason for a failure.

Example

The example given in appendix A.7, shows how spurious measurement results can be generated by unexpected situations. Trapping values outside the expected range and attempting to correct for this situation can lead to further problems.

Scope

The method has virtually universal scope. However, it is particularly appropriate for systems having a large set of states in which some actions can only be undertaken in certain states. An example of this could be an interactive dialogue with the user to calibrate an instrument.

Safety implications

This method is recommended at MSI1 and MSI2. (Note that, at the higher Indexes, static analysis should be used to *prove* that the pre-conditions are satisfied.) Further guidance is provided in the context of certification to *IEC 61508* in section B.3.6. 3–7.4.2.10

Cost/Benefits

Dynamic checking of pre-conditions is usually cheap and simple to undertake. Testing that all such pre-conditions have been implemented correctly is more demanding. Proof that pre-conditions are not violated is typically only undertaken in high Indexes of safety systems.

This technique has the major benefit of ensuring that the pre-conditions required for predictable execution hold. For instance, if an array subscript can go outside the bounds of the array, then program execution could be unpredictable.

Conclusions

This technique is highly advantageous to ensure a robust implementation.

12.8 Code review

This technique is similar to software inspection, but carried out only on the source code of an application. The concept is to ensure the readability and maintainability of the source code, and potentially to find bugs. Ensuring the traceability of the code from the specification would be covered in a software inspection, but is not usually undertaken in a code review. We are considering the code review as a manual process, rather than the analysis of source text by tools as in static analysis (see section 12.6).

Organisations developing software typically have a procedure for undertaking code review which may list a number of aspects to be checked, such as ensuring that the programming language identifiers are meaningful.

If the supplier makes the source text available to users, then a user may undertake an independent code review.

Code review is vital if long-term support of the software is required, see appendix A.5.

Example

This example relates to an extensive review of software source code for a robot measuring system by a user. In this review the following points were found.

In two modules the comment "this needs checking" was present. Either the software developer had not returned to check the code or had checked it and not removed the comment.

In one module a decision element was `if 1 = 1 then`. Thus, over a page of software under the `else` option could never be entered.

In another module the following series of decision elements was found:

```
if a = 1 then A
```
was followed by
```
        else if a = 1 or b = 1 or c = 1 or d = 1 then B
```
which was followed by
```
        else if a = 1 or b = 1 or c = 1 or d = 1 then C
```
and again followed by
```
        else if a = 1 or b = 1 or c = 1 or d = 1 then D
```

Again elements `C` and `D` could never be entered.

Poor constructs led to the decision to do a 100% review of the code. This resulted in a number of actual errors being detected, e.g. the setting of an incorrect flag or warning lamp which would lead to the operator searching for a fault that had not occurred rather than the actual fault.

The developer had to perform a number of changes to the code.

Scope

Code review should be undertaken by any organisation developing software. In this Guide, we assume that either this technique is applied, or another method is applied of a similar strength to gain confidence in the software in its source text form.

Safety implications

Code review is typically a less rigorous process that static analysis, but should certainly be performed if static analysis is not undertaken. Code review is recommended at MSI1 and MSI2, see section B.3.7. 3–7.4.4.5

Cost/Benefits

As a manual process, its success depends upon the skill of the reviewers and the level of effort applied. The costs can vary from a few minutes per page to over an hour per page if the intention is to study the source in depth. The varying costs are also reflected in the benefits which can vary from just an increased confidence in the readability, to a clear indication that the reviewer confirms the belief that the program is correct.

Conclusions

As part of software development, either the process of code review should be undertaken or a process which is clearly superior, like software inspection.

12.9 Numerical stability

It is unreasonable to expect even software of the highest quality to deliver results to the full accuracy indicated by the computational precision. This would only in general be possible for (some) problems that are perfectly conditioned, i.e., problems for which a small change in the data makes a comparably small change in the results. Problems regularly arise in which the conditioning is significant and for which no algorithm, however good, can provide results to the accuracy obtainable for well-conditioned problems. A good algorithm, i.e., one that is numerically stable, can be expected to provide results at or within the limitations of the conditioning of the problem. A poor algorithm can exacerbate the effects of natural ill-conditioning, with the consequence that the results are poorer than those for a good algorithm.

Software used in scientific disciplines can be unreliable because it implements numerical algorithms that are unstable or not robust. Some of the reasons for such failings are

1. failure to scale, translate, normalise or otherwise transform the input data appropriately before solution (and to perform the inverse operation if necessary following solution),
2. the use of an unstable parameterisation of the problem,
3. the use of a solution process that exacerbates the inherent (natural) ill-conditioning of the problem,
4. a poor choice of formula from a set of mathematically (but not numerically) equivalent forms.

The development of algorithms that are numerically stable is a difficult task *Higham*, and one that should be undertaken with guidance from a numerical analyst or someone with suitable training and experience. It requires that the intended data processing is posed sensibly and, if "off-the-shelf" software modules are used, that such software is appropriate.

There are established high-quality libraries of numerical software that have been developed over many man-years and cover a wide range of computational problems. Examples include the NAG library [*NAG*] (which is available in a number of computer languages and for a variety of platforms), LINPACK [*LINPACK*], and NPL libraries for data approximation [*DASL*] and numerical optimisation.

Example

An example which shows numerical (in-)stability is the calculation of the angle (θ) between two unit vectors. Given two vectors, $\mathbf{a}$ and $\mathbf{b}$, each of unit length, two mathematically equivalent formulae for evaluating the angle are

$$\theta = \cos^{-1}(\mathbf{a}^{\mathrm{T}}\mathbf{b})$$

and

$$\theta = 2\sin^{-1}\left(\frac{1}{2}||\mathbf{b}-\mathbf{a}||\right)$$

For vectors that are very nearly parallel, i.e., $\mathbf{a}^{\mathrm{T}}\mathbf{b} = 1-\delta$, where $|\delta|$ is much less than 1, the first formula gives $\theta = \sqrt{2\delta}$ (using the approximation $\cos\theta = 1 - \frac{\theta^2}{2}$).

The smallest θ computable in this way is the square root of the computational precision, i.e., approximately 10^{-8} radians for IEEE arithmetic. Thus, the first formula is unable to detect an angle smaller than 10^{-8} radians unless it is computed as zero, whereas the alternative formula can be expected to return accurate values.

This example has serious consequences for those concerned with implementing the procedures described in the International Standard [*ISO 10360–6*]. This standard is concerned with testing software for computing Gaussian best-fit geometric elements to measured data that is used in industrial inspection and geometric tolerancing applications. The standard requires that the unit direction vector used in the parameterisation of such geometric elements as planes, cylinders and cones and returned by test software is compared with a reference solution by computing the angle between the test and reference vectors. The standard defines acceptance of the test vector if this angle is smaller than 10^{-8} radians. It is clear from the above analysis that if the first formula is used to evaluate the angle, this acceptance criterion can never be satisfied no matter how close are the test and reference vectors unless the angle is computed as zero. On the other hand, there is no problem with undertaking the comparison if the second formula is used.

Scope

Numerical stability is an issue for all mathematical software except, perhaps the simplest arithmetic.

Safety implications

For safety applications, it is essential to ensure that the numerical results correspond to the mathematical specification (to within the required accuracy). For MSI1 systems, numerical reference results could be applied instead, but for all other Indexes, numerical stability analysis is recommended. See section B.3.8 for further guidance.

Cost/Benefit

There are costs in finding a sufficiently stable numerical algorithm for a particular application and there may be increased costs of development and computation in using a numerically stable algorithm as compared to the "obvious" calculation. However the costs from inaccuracy in the output from using an unstable algorithm can be catastrophic; they can certainly lead to "out of bounds" errors (e.g. $1/0$, or $\sqrt{x}$ where $x < 0$) in subsequent calculations.

Conclusions

It is usually right to use a numerically stable algorithm if one is known. The use of respected software libraries for generic calculations will take advantage of the numerically stable algorithms used by such software. Otherwise, it is usually right to do some investigation of the stability of algorithm being used. If a numerically stable algorithm is much more expensive to implement or use, then an analysis of the benefits in terms of the desired accuracy and robustness is needed.

12.10 Microprocessor qualification

For safety systems, the safety integrity requirements imply that any design error in the microprocessor itself could be significant. In consequence, a process needs to be undertaken to assess the risk of such an error and, potentially, undertake measures to reduce the risk.

It might be thought that this process applies only to hardware and therefore is not relevant to this guide. However, design errors may be effectively software errors in the microcode of the processor and hence should be treated as similar to conventional software errors.

For details of the issues raised, see *Microprocessor*.

Example

A frequently quoted example of a microprocessor design fault is that of the divide error in the early Pentium chips from Intel. When the error was first reported, substantial usage of the chip had been made, and therefore it could potentially have been used in safety applications. Note that the error was found by a person using the Pentium in an unusual way, which indicates that such errors can remain dormant for a considerable time.

Analysis of the Pentium bug

The bug was reported by Dr Nicely on the 24 October 1994. He was able to detect the error because he was undertaking calculations with long integers, which are most efficiently done with the floating point hardware. In some cases, he *knew* the result should be an integer, but the Pentium was not giving that.

The bug was in the FDIV instruction, which for about 1 in 10^{10} pairs of randomly chosen divisors suffers a catastrophic loss of precision, ending up accurate to only 4 to 6 digits instead of the usual 18.

The bug is easily detected with the NPL/NAG Floating Point Validation package [*FloatingPointValidation*] since the package uses carefully constructed bit patterns for testing, which give a much higher rate of detection than random operands.[2]

[2]There is a very obscure error in all the early Intel chips in that denormalised numbers are not rounded correctly. This error should never cause a safety system to go unsafe and it is quite reasonable for a supplier

The actual fault has been reconstructed from knowledge of the patterns that fail, without Intel's assistance. The algorithm is a radix 4 division with a four-way case statement. One of these cases which is very rarely executed was incorrectly coded. If this error was in code rather than microcode, the non-execution of a test case would be a gross error according to *DO-178B*, the civil avionics standard.

There is no question that this bug could cause a "correctly" coded safety system to become unsafe. This is worrying since the error rate with random operand values is sufficiently low to remain undetected during the conventional testing process.

Given a safety system using a Pentium, then it is quite likely that floating point division is not used, or that the four digits of precision are sufficient. Let us assume that this does not apply and that one division is undertaken per millisecond. We can assume that the operands are random, since the physical quantities would not have the specific bit patterns. Hence the error rate would be once every 116 days. The key issue would be to determine the system error rate from this software error rate. In any case, the evidence from this bug is that "proven in use" is inadequate unless carefully reasoned, since the Pentium chip had been around in very large quantities before the bug was reported.

More worrying are the implications of similar Pentium-like bugs. Division is a very well-defined operation, but the detection depended upon Dr Nicely's use of the instruction which gave a degree of self-checking that would not be typical of other uses. Hence error rates of the same magnitude could remain undetected for much longer (and obscure ones might never be corrected since the market would not demand it).

Scope

This technique is only relevant to MSI4 systems, since it is only in that context that the safety integrity of conventional processors need be questioned.

Safety implications

As above, this is only relevant for MSI4 systems.

Cost/Benefits

If the microprocessor vendor does not provide the necessary information, it can be hard and expensive to resolve the issue.

Conclusions

It would be best to ensure that a supplier is identified, who can provide the assurance needed, before a specific chip is chosen.

12.11 Verification testing

By the term "verification testing", we mean functional testing that aims to be as "complete" as possible (sometimes called "validation testing"). Such testing is conventionally undertaken separately from the development team and uses tests based upon a detailed functional specification. The testing acts as a check on the specification as much as it acts as a check on the implementation.

not to repair such faults, unlike the Pentium divide.

Such testing can require a substantial effort but cannot easily be avoided if a high level of assurance is required, i.e., MSI3 or MSI4.

Since the concept of "completeness" cannot be effectively defined, it is conventional practice to combine this form of testing with structural testing to some level, say branch testing. In the case of the avionics standard [*DO-178B*], the structural testing required at the highest level of assurance (MSI4) is MC/DC (see section 12.13).

Scope

Due to the cost, this technique is only appropriate to MSI3/MSI4 systems.

Safety implications

This technique is recommended for MSI3/MSI4 systems, and further guidance in the context of certification to *IEC 61508* is to be found in section B.3.10. 3–7.4.8.1

Cost/Benefits

The main benefit of such testing is to increase confidence in the specification of the software. It is therefore essential that the specification is analysed separately from the implementation team, otherwise a simple misinterpretation could be missed.

Conclusions

This is an expensive technique which cannot be avoided for high assurance systems.

12.12 Statistical testing

At the software component level, statistical testing is one of the methods in the British Standard [*BS 7925*]. However, in this case, we are using the technique for the entire software system, rather than a component.

If the statistical distribution of the inputs to a system is known, then statistical testing can be undertaken by observing the result of samples from that distribution. One can then deduce the reliability rate for the system. Unfortunately, it is rarely the case that the distribution is known accurately enough, and even if it is, it may not be possible to check the correctness of the outputs (i.e., the measurements in our case).

Scope

In principle, the method can be applied to any system. In practice, the limiting factor is the lack of accurate enough data on the input distribution.

Safety implications

A key advantage of this method is that it gives a bound to the reliability of the system. Unfortunately, the bound is inadequate for highly reliable systems since it is practically impossible to run the millions of tests that would then be necessary, see *Littlewood*.

This method is recommended in the context of certification to *IEC 61508* for MSI2/MSI3 systems, see section B.3.11 for further guidance.

Cost/Benefits

Depending on the nature of the system, running the number of tests needed can be expensive.

Conclusions

This method is very different from most forms of conventional testing but provides direct evidence of reliability.

12.13 Component testing

Component testing is a basic software engineering technique which can (and should) be quantified. The software component to which the method is applied is the smallest item of software with a separate specification (sometimes called a module). It is very rare for the technique *not* to be applicable for a software development. The best standard, which is now available through BSI, is the British Computer Society standard [*BS 7925*]. The method is objective, repeatable and reproducible, and hence satisfies the requirements for accredited testing.

The *BS 7925* standard allows for many levels of testing and in this Guide we reference five forms below. The term **structural testing** is really a generic term to all those forms of testing related to the structure of the code. However, here we use it in a specific sense for the simplest testing technique.

Structural testing. This method requires that there is evidence of component testing which shows that all the primary program units have been executed. This is a way of ensuring a degree of adequacy of the testing. This is recommended at MSI1.

Equivalence partition testing. This is complete functional testing at the component level. This is to be applied to those components handling the basic measurement/test data. This method is not recommended due to the practical difficulties of its implementation. But it is an ideal technique to be used if available; and if used, it is recommended to undertake this to 100% coverage.

Statement testing. This method concerns the statement coverage for those components handling the basic measurement/test data. If a statement has not been executed, then a reason for this should be documented. Defensive programming techniques and detection of hardware malfunction give rise to statements that cannot be executed (but are quite acceptable). It is recommended to undertake this to 100% statement coverage for MSI2 and MSI3.

Branch testing. This method requires testing of different branches from decision points. The aim should be 100% coverage, but this may not be achieved in which case the reasons need to be documented. This is recommended at MSI3 and MSI4, taking the same approach as for statement testing.

Boundary value testing. In this case, values are chosen which lie on the boundary between partitions. This form of testing is designed to show that the boundary cases themselves are correctly handled. It is recommended to use it for MSI2 and above.

Note that this is a dynamic testing technique as opposed to boundary value analysis which is a static analysis method to analyse boundary values within an implementation.

Modified Condition/Decision testing. This form of testing is required at the highest levels in the civil avionics standard [*DO-178B*]. The terms used in that standard in Modified Condition/Decision Coverage (abbreviated "MC/DC", see reference *MCDC*), even though the coverage required is 100%. However, due to its high cost, it is not recommended (below MSI4) in this edition of the Guide. It is a technique which needs to be considered when MSI4 guidance is completed.

Example

In the *SmartInstrument* research project, it was concluded that equivalence partition testing and boundary value testing are the most cost-effective methods in this area. However, the code in this example was relatively simple.

Scope

The technique is very general and can be applied to any software written in a conventional programming language. The method is widely used and understood. Metrics are available which provide a measure of the thoroughness of the actual testing undertaken.

The fundamental drawback of the method is that "missing code" cannot be identified. For instance, if a routine is defective in not protecting itself against invalid input, then structural coverage testing will not detect this.

The method cannot be applied to software only available in compiled form, such as library routines provided by other parties (i.e., COTS).

The technique should be applied to each component within the software. However, it may be difficult to produce test cases for some components in isolation from the rest of the software. When a component is not isolated, it may be difficult to produce all the test cases that are needed to attain a high coverage figure.

In the context of measurement system software, it may be difficult to apply if the software cannot be isolated from the measurement system itself, since it may not be possible to get complete repeatability. (This is when the entire software is regarded as a component.)

Safety implications

Testing is the most widespread form of validation for software. Component testing is one of the most effective forms of testing, since it is often possible to obtain a high coverage of the potential test cases. Hence any failure to undertake a reasonable coverage of test cases would be regarded as inappropriate for safety related software. 3–7.4.7.1

Various forms of component testing are recommended at all Indexes, and the details are to be found in section B.3.12.

Cost/Benefits

The only cost-effective method of applying the technique is with the aid of tools which automatically indicate which statement, etc. has been covered in the testing. Such tools are used on a routine basis by some software development teams but not others.

Costs can be high if, say, 100% coverage is required. Under such circumstances, work in analysis of the algorithm to produce the necessary test cases can be very demanding.

Equivalence partition testing is effectively black-box testing for a component done in an ideal manner. This method should be applied for the most critical software components of a

system. The method is simple and easy to apply to simple software, but much more difficult with complex software.

Conclusions

Software development teams should ensure they have the necessary tools to perform at least statement coverage analysis without a significant burden on their development process. The author of a component should review that statement coverage data to see if crucial test cases have been omitted.

Initial test cases should be derived from equivalence partition testing. The statement coverage data then shows if the testing is adequate.

12.14 Regression testing

Regression testing requires that tests are developed and used to re-test the software whenever a change is made. Typically, sometime before the first release, a set of tests will be designed and run on the software. From that point on, all errors located should result in an additional test being created which would detect the bug.

To be effective, one needs a method of re-running the set of tests automatically. For this reason, we have considered the technique to be part of the programming environment, see section 11.2. The technique is very good for software that is widely used and for which initial bugs are not a major problem. The effect of the method is that subsequent releases of software should be able to rely fully on the unextended facilities.

Scope

This method is very powerful for software that is being constantly updated and enhanced. Correspondingly, it is not effective for software embedded within a measurement system (say) which has to be without a serious defect on the first release. If a field upgrade is needed with such software, there could be very serious cost implications.

To be applied, the software must be separated from the hardware of the measurement system.

Safety implications

In many cases, regression testing is not appropriate for safety related systems, since the method assumes an initial relatively high failure rate. On the other hand, for bespoke SIL1/SIL2 systems, for which an upgrade does not have serious cost implications, regression testing may be very effective in ensuring the absence of known errors.

Cost/Benefits

Individual test cases need to be recorded within a system which allows for minimal effort to re-execute them. Hence, given a modest additional effort in the initial tests, further releases of the software should be testable very effectively. (One reason why compilers, which are typically very complex, are actually very reliable, is that almost all software developers use regression testing.)

Conclusions

Regression testing should be undertaken if there are to be regular new releases of the software with large amounts of functionality remaining unchanged between releases.

12.15 Accredited software testing using a validation suite

Accredited software testing requires that a set of tests be developed (the validation suite) against which the software can be tested. This is appropriate for software having an agreed detailed specification, such as compilers and communication software. Accredited testing ensures that the tests are run and the results interpreted correctly, with the specific requirements of objectivity, repeatability and reproducibility.

The method is significantly stronger if the set of tests is updated regularly by means of regression testing. This implies that errors in any implementation will result in tests being applied to all (validated) systems.

This form of testing provides an ideal basis for certification.

Example

An example of this form of testing for a measurement system is that being proposed in the area of Nuclear Medicine [*COST B2*]. Here, gamma-camera pictures are taken of patients when being treated with substances containing radio-active trace elements, see appendix C.3. The camera output is translated into a standard file format, but the difficult numerical part is the analysis of the picture to give the basic information for a medical diagnosis. Other strong methods, such as a mathematical specification cannot be applied, and hence this method provides a means for the whole international Nuclear Medicine community to gain confidence in the analysis software.

Note that the application is **safety-indirect** and the complexity of the processing of data is **complex** which implies a high **MSI**, usually MSI3. At this Index, the technique of accredited testing is not actually recommended (see table 11.2 in section 11.2), but it is one of the few methods which can provide reasonable assurance in this context.

This method is made more effective by means of software phantoms which are pictures for whom an agreed diagnosis is available (at least in the cardiac and renal areas), as explained in *COST B2*.

Scope

All the known applications of this method are generic: such as compilers, for which many instances are available (to effectively the same specification).

A potential example in the area of length metrology would be tests for the software that analyses data from Coordinate Measuring Machines.

For such a method to be effective, there has to be an underlying test strategy which can be applied to a range of software products. For instance, it is unclear that the method could be applied to word processors, since there is no generic specification (with sufficient precision for testing).

On the other hand, the method *could* be applied to the common mathematical operations within a spreadsheet.

Safety implications

This is a specialised method which is only appropriate for systems of MSI1 or MSI2. It is only recommended here for MSI2 systems, see section B.3.14.

Cost/Benefits

The costs to produce the generic test data can be quite high so that the method is limited to those cases in which there is a national or international consensus that such a development should be undertaken.

Conclusions

This technique is appropriate if the relevant industrial sector can agree on the production of the required validation suite, and the MSI required is 1 or 2.

12.16 System-level software functional testing

System-level software functional testing (or simply "system testing") is based upon testing the entire software as a black box by a careful examination of the functionality specified, ensuring that every aspect of the functionality is tested. An International Standard is based upon application of this test method [*ISO 12119*].

The method can clearly be applied to any system. In this context, we are considering its application to the complete software, but separate from the measurement system hardware. This limits the application to those cases in which such separation is possible.

The problem with the technique is the cost of constructing the necessary test cases, and ensuring that the test cases are sufficient. Since errors have been reported in many software systems which have been very carefully tested, one cannot gain very high assurance from this method.

However, in cases in which the software has a simple structure, such testing may be sufficient.

Example

NPL has recently tested a software product to *ISO 12119*. The product was a small item of software to detect if a PC had a problem with the clock at the time of the millennium year change, and correct it, if required.

The program came on a floppy disc, with essentially no options. Hence the testing was mainly to ensure that different types of PC would have their clocks corrected, as claimed in the supplier's specification. This could be undertaken by first seeing if the PC had a problem, then running the software, and finally checking the machine again.

The testing was undertaken in line with UKAS requirements, and the test report produced is available for inspection [*Year2000*].

If the program had 20 different installation options, then the complete testing required by the standard could require 2^{20} test cases and hence be too costly to undertake.

Scope

The technique can be applied to software systems which are sufficiently simple that complete functional testing is economically viable.

Unfortunately, the method is not viable for most complex software.

Safety implications

System level testing is recommended at MSI1 and MSI2. If reliance is placed on this form of testing alone, then it is important that the complexity of the software is such that all the functionality is adequately demonstrated to have complied with the specification. Additional guidance in the context of certification to *IEC 61508* appears in section B.3.15. 3–7.4.8.1

Cost/Benefits

As noted above, the problem is that the costs can be high unless the software is very simple (at least in its specification). If high assurance is required and the claims made for the software can be converted to appropriate test cases, then the method can be applied.

Conclusions

Since this technique is completely independent of the development, it can be considered whenever resources are available and the independence of the assurance provided is thought to be worthwhile.

12.17 Stress testing

Stress testing involves producing test cases which are more complex and demanding than are likely to arise in practice. It has been applied to testing compilers and other complex software with good results. The best results are obtained when the results can be automatically analysed.

Example

Consider a measurement system which claims to be able to make measurements even when there is some background "noise" on the input data. Then a method of stress-testing would be to take a set of input data without noise, and then add noise. The correct answer is therefore known and if the noise can be added in an effective manner, then a stress-testing method can be applied. Clearly, when the level of the noise gets too high, the measurement system cannot be expected to give a reading. However, a false reading should not be given.

The generation of the "noise" by software should be easy to undertake, and since the correct result is known, the automatic testing of the measurement system should be possible with perhaps a few hundreds of individual test cases. If any failures arise, these can be examined manually to determine the source of the problem.

A special case of this technique is the application to video images. The starting image could be a high quality image or one produced artificially. An increasing level of noise is added to the image until the processing software fails. The final image which the software failed to process is then printed and a manual inspection made to see if a failure is acceptable in this case. It appears that this technique is widely used in validating software which processes video images.

See *StressTest* for an example in testing compilers.

Scope

The method can be applied without knowledge of the internal structure of the software. It is particularly appropriate for complex software when it is possible to generate test cases automatically.

Safety implications

Additional guidance on the method in the context of certification to *IEC 61508* is to be found in section B.3.16.

Cost/Benefits

The costs depend on the development needed for the test case generator. The benefits depend upon the assurance that passing such tests gives. For the example using compilers [*StressTest*], the cost of producing the test generator was quite high. However, the cost may be acceptable because very few other means are available for independently generating a large number of complex test cases.

The technique applied to video images does not require a test generator, since noise can be added to captured images. Also, graphics processing packages can be used to manipulate images prior to the addition of the noise. However, very high confidence in image processing is hard to achieve since the variety of images that can be produced is very large.

Conclusions

Stress testing is recommended at MSI2 and MSI3. The use of the technique depends upon the ability to derive test cases and expected results with ease. Given these requirements, then a cost/benefit analysis should be undertaken to see if the method is viable.

12.18 Numerical reference results

There is a growing need to ensure that software used by scientists is fit for purpose and especially that the results it produces are correct, to within a prescribed accuracy, for the problems purportedly solved. Methodologies, such as that presented in *ReferenceDataSets*, have been developed to this end. The basis of the approach is the design and use of reference data sets and corresponding reference results to undertake black-box testing.

The approach allows for reference data sets and results to be generated in a manner that is consistent with the functional specification of the problem addressed by the software. In addition, graded data sets corresponding to problems with various "degrees of difficulty" or conditioning (section 12.9), and with application-specific properties, may be produced. The comparison of the test and reference results is made objective by the use of quality metrics. The results of the comparison are then used to assess the degree of correctness of the algorithm, i.e., the quality of the underlying mathematical procedure and its implementation, as well as its fitness-for-purpose in the user's application.

The methodology has been applied successfully in particular areas of metrology. In dimensional metrology, for example, coordinate measuring machines (CMMs) are typically provided with software for least-squares (Gaussian) geometric element fitting. The methodology provides the basis of an International Standard [*ISO 10360–6*] for testing such software, and it is intended to base a testing service on this standard. Data sets have been developed in such a way that the corresponding reference results are known a priori. Consequently, there

is no reliance on reference implementations of software to solve the computational problems, but the generation of the data sets is dependent on a set of simpler "core" numerical tasks that are well understood.

Example

In a number of metrology areas it is necessary to identify *peaks* within spectral and other traces and, particularly, to quantify a peak in terms of parameters such as location, amplitude and width. Algorithms for peak analysis occur in a range of measurement-system software [*Chemometrics, Dyson, AOAC*]. Commonly, a particular functional form for the mathematical representation of the peak is assumed, e.g., Gaussian, Lorenzian or modified Gaussian. This form is fitted by, e.g., least squares, to the measured values from the trace. The required parameters are then determined from the fitted form. A widely-used form is the Gaussian model:

$$y = A \exp\left(-\frac{(x-\overline{x})^2}{2\sigma^2}\right)$$

where $\overline{x}$ represents location, A amplitude and σ peak width.

In order to determine whether peak-fitting software is functioning correctly, numerical reference results can be devised. Such test sets consist of synthesised reference data sets and corresponding reference results [*CISE 26/99*]. Such data sets can be generated corresponding to ranges of key variables (extent of peak spanned by the data, peak height, width ratio, signal/noise value, etc.) using the null-space data generation technique;
see [*ReferenceDataSetsDesign, CISE 25/99, ReferenceDataSets, GradedReferenceDataSets*]. These ranges should be selected to correspond to those expected to be experienced in the use of the measurement system.

Scope

The null-space method of data generation is very general, being applicable to a wide range of fitting and optimisation problems [*Gill, SoftwareTesting*]. Function forms other than Gaussian peaks can be considered, and fitting measures other than least squares can be entertained. The use of the numerical reference results permits an assessment of the extent to which a numerically-satisfactory algorithm solution has been implemented. For instance, it will be possible to deduce whether a potentially-unstable process such as one based on the formation of the normal equations [*CISE 27/99, Golub*], as opposed to the direct use of the observations (design) matrix, has been employed. Moreover, it can be discerned whether conceptual errors in the solution process have been made, such as fitting the logarithm of the Gaussian model (which is equivalent to a quadratic polynomial) to the logarithm of the data. Within such approximate solution processes there are "degrees of correctness" such as partial compensation, through the use of appropriate weighting functions, for such model and data transformations. Again the use of numerical reference results can help to identify such techniques, the adequacy of which depends on practical quantities such as the signal-to-noise ratio.

Safety implications

This technique is recommended at MSI1 and MSI2. If an analysis for numerical stability is not possible, then the method is particularly appropriate. Thus, it can also be used at MSI3 and MSI4. Additional guidance in the context of the certification to *IEC 61508* appears in section B.3.17.

Cost/Benefits

The methodology for generating data sets for numerical reference results is established. Its implementation in any particular instance may be governed by the ease with which the measurement system supplier has made provisions for software testing using simulated data. Since such testing should be fundamental to measurement system development, there would be an advantage in the manufacturer providing comparable facilities for user testing to establish fitness for purpose.

Conclusions

Synthesised numerical reference results provide a powerful facility for testing the degree of numerical correctness of measurement software. Techniques such as the null-space method can be used to generate test sets with known solutions. It is only necessary to be able to provide a characterisation of the solution to the problem purportedly solved by the software. This characterisation is closely related to the functional specification of the software, which has to be prepared in any case.

It is necessary that the measurement system supplier provides a facility by which data sets may be input to, and corresponding results output from, the software. Such provisions should be a design consideration of the manufacturer's system configuration and testing regime.

Arguably the most important feature of numerical reference results is that they can be prepared to accord with the expected function of the measurement system and the nature of the measurements taken, in terms of data densities, signal-to-noise ratios, etc.

12.19 Back-to-back testing

In back-to-back testing, two comparable software systems are tested with the same input. The outputs from each test are compared — identical results are not usually expected when numerical testing is undertaken. Therefore, the comparison should seek agreement with an appropriate tolerance. If the comparison can be automated, then it may be possible to run a large number of tests, thus giving a high assurance that the two items produce similar results. Of course, one of the items under test is likely to be a version of known characteristics, while the other is the item being assessed.

This form of testing can be applied at a higher level than just a single software component. Indeed, the standard method of calibrating measurement systems can be seen as a back-to-back test of a trusted measurement system against one to be calibrated.

This form of testing can also be used to test the soundness of numerical software within a measurement system. The results of the numerical calculation from the measurement system are compared against the results of a piece of reference software which is a proven implementation of the numerical calculation. This is one of the methods used in the SS*f*M numerical software testing project, see *CISE 25/99*. Where there is no existing implementation, it may be possible to produce a reference implementation based on sound numerical analysis and using reliable software libraries. This reference implementation will presumably be less useful than the implementation used by the measurement system (e.g. it may be very slow) but it should give trusted output.

In the case of communications software, back-to-back testing takes a different form. Two versions of the software are coupled back-to-back and then run through appropriate test sequences to show whether or not they will interoperate satisfactorily. Such software is not relevant to conventional measurement systems, but will be used in distributed systems

in which the complete measurement system consists of two or more component systems, interconnected by a communications link.

Example

In the SMART reliability study *SmartInstrument*, this form of testing was used by testing a MatLab implementation against the C code within the measurement system. The test cases used were those derived from boundary value and equivalence partition testing.

Scope

The key requirement is a trusted system (or oracle) against which the testing can be applied. It is also necessary to have a comprehensive set of test cases from which a comparison can be made.

Note that back-to-back testing is quite different from parallel running or multi-channel systems. In this case, it is assumed that diagnostic information is available to allow the discrepancy to be traced and the software corrected. Hence the oracle forms no part in the delivered system.

Safety implications

This method is recommended at MSI2 and MSI3. Clearly, the method depends upon the availability of a *trusted system* with which to compare the system under test. Additional guidance in the context of the certification to *IEC 61508* is to be found in section B.3.18.

Cost/Benefits

Given a trusted implementation and a set of test cases, the method is cheap to apply. Conversely, if a trusted implemented is not available this method is probably not applicable, except in the case of communications software, when two copies of the same software can still usefully be run back-to-back.

In the case of testing numerical software, if there is not an existing reference implementation it may be possible (but expensive) to produce an implementation of the numerical calculations in isolation.

Conclusions

Since a trusted implementation is required and would be expensive to produce specially, the method depends upon the existence of such an implementation. On the other hand, given such an implementation, the method is recommended for MSI2 and MSI3.

12.20 Comparison of the source and executable

Since it is effectively impossible to test dynamically all the possible cases with any non-trivial piece of software, one is bound to depend upon the source text. This implies an implicit dependence upon the correctness of the compiler (linker, etc) in producing the executable program.

For the highest level of assurance (MSI4), it is necessary to ensure that a compiler bug does not invalidate the reasoning that the system will be safe. One method of achieving this is to compare the source code with the executable image.

Example

An excellent example of this method is to be found in *SIZE1* and *SIZE2*. However, although two errors were found in 100,000 lines of code, it is thought that most compilers would be more reliable than that. An alternative method of obtaining confidence in a compiler is by stress-testing, see section 12.17.

Scope

This technique is universal to compiled languages, but only appropriate for the highest assurance (MSI4).

Safety implications

This method is only recommended in the context of certification to *IEC 61508* for MSI4 systems. Additional guidance is to be found in section B.3.19.

Cost/Benefits

This technique is too expensive for use with anything but MSI4 systems.

Conclusions

Comparison of source and executable code should be applied for MSI4 systems, especially when information about the status of the compiler is lacking.

Appendix A

Some Example Problems

A number of illustrative examples are collected here of software problems that have been reported to NPL over a number of years. The exact sources are deliberately not given, even when they are known.

A.1 Software is non-linear

A simple measuring device was being enhanced to have a digital display. This was controlled by an embedded microprocessor, with the code produced in assembler. The product was then to be subjected to an independent test. The testers discovered, almost by accident, that when the device should have displayed 10.00 exactly, the actual display was nonsense. The fault was traced to the use of the wrong relational operator in the machine code.

The example contrasts with pre-digital methods of recording measurements in which the record is necessarily linear (or very nearly linear).

The example illustrates that the testing of software should include boundary conditions. However, only the most demanding standards actually *require* that such conditions are tested. For a four-digit display in this example, it should be possible to cycle through all the possible outputs to detect the error.

A.2 Numerical instability

The repeatability standard deviation of a weighing balance was required as part of a reference material uncertainty estimate. Successive weighing of a nominal 50 gramme weight produced a set of fifteen replicate values as follows: 49.9999 (1 occurrence), 50.0000 (5 occurrences), 50.0001 (8 occurrences) and 50.0002 (1 occurrence).

The processing of the data used an in-built "standard deviation" function operating to single precision (eight significant figures). Because the data could be represented using six significant figures, the user anticipated no difficulties. The value returned by the function, however, was identically zero.

The reason is that the function implements a "one-pass" algorithm that, although fast to execute, is numerically unstable. The standard deviation computation in this algorithm is based on a formula involving the difference between quantities which are very close for the above data values, thus causing the loss of many figures. An alternative "two-pass" algorithm that first centres the data about the arithmetic mean and then calculates the standard deviation returns an answer for the above data that is correct to all figures expected.

Unfortunately, the "one-pass" algorithm is widespread in its use in pocket calculators and spreadsheet software packages.

The example described above is concerned with the stability of the algorithm chosen for the required data processing. Numerical difficulties may also arise from the improper application of good algorithms. In one example, the processing software was to be ported from one (mainframe) platform to another (PC) platform. Although the platforms operated to similar precisions, and the same numerically stable algorithm was used (albeit coded in different languages), the results of the processing agreed to only a small number of significant figures. The reason was that the linear systems being solved were badly scaled and, therefore, inherently ill-conditioned, i.e., the solution unnecessarily depended in a very sensitive way on the problem data.

The lessons of this are: ensure the required data processing is stated as a well-posed problem; then use a stable algorithm to solve the problem.

A.3 Structural decay

A contractor is used to develop some software. They have very high coding standards which include writing detailed flow diagrams for the software before the coding is undertaken. The contractor corrects these diagrams to reflect the actual code before delivery to the user. It is satisfactory to use flow charts to generate a program. But once the program is written, these charts become history (or fiction), and only charts generated from the program source are trustworthy.

The user has tight deadlines on performing modifications to the software over the subsequent five years. For the first two amendments, the flow diagrams were carefully updated to reflect the changes to the code, but after that, no changes were made so that the flow diagrams were effectively useless. As a result, the overall "design" provided by the contractor was effectively lost. The problem was that the "design" was not captured in a form that could be easily maintained.

The conclusion from this is that for programs which have a long life, one must be careful to capture the design in a format that can be maintained. Hence it is much better to use design methods which support easy maintenance — hand-written flow charts are exactly what is *not* needed!

A more serious example of the same aspect is the use of programming languages which do not support high-level abstraction, for instance C as opposed to C++.

A.4 Buyer beware!

Professor W Kahn is a well-known numerical analyst who has also tested many calculators over many years. Several years ago, he illustrated an error in one calculator in the following manner: assume the calculator is used to compute the route to be taken by an aircraft flying between two American cities, then the error in the computation would result in the aircraft flying into a *specific* mountain.

Modern calculators are cheap and usually reliable. However, errors do occur. Hence the use of such machines in life-critical applications needs serious consideration. If the same calculations were being performed by software within the aircraft, then the very demanding avionics standard would apply *DO-178B*. Hence, when used for a life-critical application the same level of assurance should be provided by the calculator (however, it probably would not be cheap).

In the context of a measurement system, the above dilemma is critical. Many systems are sold without any specific application in mind. If the supplier makes no claim about such a system, is it appropriate to use the system in a safety application? If the measurements have a direct impact upon the safety of a system, then it is clear that the user has a responsibility to ensure that the system will not make the system unsafe (within the constraints of As Low As Reasonably Practical — ALARP).

With most physical devices, if high quality/assurance is required, there is likely to be a significant cost implication. Hence the supplier could be expected to charge a premium for an system qualified for use in safety application. However, with software, the situation is not so clear-cut, since the replication cost of software is effectively zero. Why should a mass-produced calculator be any less reliable/accurate than a similar device produced for a specialist market?

A.5 Long term support

Many applications at NPL are required to be available for over 10 years. For this length of time, problems can easily be encountered in ensuring the (hardware and) software will be properly supported.

Even with short-term support, problems can arise. Often, software is purchased some time before very extensive use is made of it. When such extensive use is applied, the initial support may have lapsed. Very many years ago, some defects were found in a small computer which resulted in the following response:

> *The software engineer who wrote the ZX81 BASIC is no longer with us, and his successors do not feel qualified to comment on your findings.*

Of course, this machine is no longer of interest, but the principle still applies. Companies can respond more easily and more quickly if faults are reported in the first few months of release, while the original design team is still available.

Users should estimate the expected life-time of an application and plan for the required support.

A.6 Measurement System Usability

In the quest for higher accuracy, it is sometimes forgotten that a measurement system will not always be used by the expert who designed it. With modern interactive software, it is possible to make the most complex of measurement systems "easy" to use. However, it can be poor unless the design ensures that operator error is minimised.

Software within a measurement system can aid the user, but care is needed if the user is not to be misled. For instance, one modern spectral analyser provided an automatic means of separating adjacent peaks on a spectrograph, rather than being operator-driven. This worked satisfactorily most of the time, but the cases of failure were hard to detect and easy to miss. If automatic analysis is to replace manual analysis, one needs to ensure a comparable level of accuracy.

A.7 Tristimulus colour measurement

In this section, we consider an actual application with its own lessons to be learnt.

Three standard curves were developed by user trials in NPL many years ago as part of work with the CIE (International Commission on Illumination — abbreviated as CIE from its French title Commission Internationale de l'Eclairage). These curves correspond to nominal spectra for Red, Green and Blue respectively.

When a tristimulus colorimeter performs a measurement, the system performs a match of the spectra generated to apportion a best fit to the standard curves. The software then uses a set of standard equations to display the colour using one of the very many possible scales. The particular scale used was the $L\,C\,H$ where L is the Lightness, C the Chroma and H the Hue.

A source code review of the internal code of this particular instrument was undertaken and a number of protocols were performed to test the functionality of the instrument.

After about 3 months routine use, some trials were undertaken to introduce a new product. In theory, the maximum value for L is 100. The instrument obtained results of over 10000. Close examination of the sample showed it was generating a small stream of CO_2 bubbles through incomplete degassing of the solution. The minute stream of CO_2 bubbles was blocking the light path. The result was that the detector was only picking up noise and was trying to match this to the standard curves.

The manufacturer added a software trap to detect transmission values of less than 0.5%.

Two months later on a routine test, several samples produced a lightness value of 100 and a chroma value of 10. This was investigated as the chroma should have been approximately 25. This time investigation showed the fault lay with the operator who had calibrated the instrument using a cell filled with the solvent as the "white" standard. In this case the cell had not been properly cleaned before the calibration. It then was revealed that the manufacturer's software contained a trap that if the sample's measured values were greater than obtained for the white standard they would be reduced to that of the white standard. The manufacturer's theory was that as a white standard is defined as 100% of light being transmitted then you can never exceed 100% and any values exceeding 100% could only be noise.

The manufacturer removed this trap.

A.8 Balances

A new balance was launched and was assessed for its suitability for use. One of the options when the balance was linked to a PC was to send a command to lock out parts of the keyboard. When we carried out some trials it was found that following a certain combination of commands a fault occurred and the balance keyboard was completely disabled.

The manufacturer issued a new software version.

Internally the user produced a check weighing package. Extensive test protocols were performed using a wide variety of balances from a number of suppliers. The software was issued to one of the user's customers using a balance option that had not been specifically tested. It was found that when a sample was added to the balance the weight displayed on the balance continued to display zero. Investigation showed that the user software was requesting a weight from the balance and used an algorithm to test that weight had increased. If the weight had not increased the software would request a further weight. The problem was that the balance contained two chips which gave rise to the difficulties. The first contained the software communicating to the weighing cell. The second contained general software that collected the weight from the first chip and sent it to the balance display and the PC. What was happening was that the software on the second chip was prioritising the data link to the PC. The manufacturer's testing had been performed using a 486 PC. The user's customer

was using a Pentium II processor and the additional speed at which data was requested was overwhelming chip 2. Chip 2 no longer had the capacity to talk to chip 1 to see if the weight had increased.

The manufacturer modified the software to solve the prioritisation.

Appendix B

Certification to *IEC 61508*

B.1 The CASS Scheme for Safety-related Systems

Systems comprising electrical and/or electronic components have been used for many years to perform safety functions in most industry application sectors. Computer based systems, generically referred to as programmable electronic systems, are being used in all application sectors to perform non-safety functions and, increasingly, to perform safety-related functions. If computer system technology is to be effectively and safely exploited, it is essential that those responsible for making decisions have sufficient guidance on the safety aspects on which to make those decisions. Such decision makers need assurance that instrumentation they purchase for integration into safety-related systems itself meets acceptable safety requirements.

Where measurements are made as part of safety-related systems, for example detectors initiating automatic emergency shut-down in petrochemical plants, the design of both hardware and software aspects of the measuring instruments must be carefully considered to ensure they do not compromise the overall functional safety of the system. Safety-related systems must be reliable and must ensure that, in an emergency, plant shut down is achieved in a controlled manner. *IEC 61508* is the international standard for the functional safety of safety-related systems comprising electrical, electronic and/or programmable electronic components. It sets out a generic approach for all safety life-cycle activities from initial specification, through operation and maintenance, to eventual de-commissioning of systems that are used to perform safety functions.

This unified approach has been adopted in order that a rational and consistent technical policy can be developed for all such systems, and provides the basis for facilitating the development of application sector safety standards. CASS (Conformity Assessment of Safety-related Systems), is an initiative that has been developed by industry, in conjunction with the Health and Safety Executive (HSE) and supported by the Department of Trade and Industry. It provides an overall safety framework for the development of a conformity assessment scheme meeting the requirements of *IEC 61508*, and those of the HSE, whereby accredited third party certification bodies can offer consistent conformity assessment certification for safety-related systems. The scope of the CASS Scheme, which is being introduced in phases, will cover the activities of all those involved in the specification, design, development, manufacture, implementation, support and application of hardware and software components and complete systems, across many sectors. It will cover both

off-the-shelf products and application-specific systems, and the operation and maintenance of those systems.

The first phase of the Scheme is for the certification of an organisation's Functional Safety Capability Assessment (FSCA), which relates to management processes rather than individual products, and is the first step for an organisation seeking certification to *IEC 61508* through CASS. Organisations can then go on to have products or systems individually certified where appropriate. This two stage approach avoids duplication of assessment of the common safety management system elements when issuing individual product certificates.

3–7.3.2.4

Manufacturers of measurement instruments having embedded software, especially where such instruments may be used in safety-critical field bus applications, are increasingly likely to be asked to demonstrate conformance to *IEC 61508*. CASS offers a route by which this can be achieved.

Further information on the CASS Scheme can be obtained from:

CASS Scheme Ltd, South Hill, Chislehurst, Kent, BR7 5EH;
Tel: 020 8295 2951; Fax: 020 8467 0145;

and on accredited certification from:

Sira Certification Service, South Hill, Chislehurst, Kent, BR7 5EH;
Tel: 020 8467 2636; Fax: 020 8295 1990; e-mail: certification@siratc.co.uk

B.2 Software documentation checklist

3–7.1.2.7
3–7.7.2.3

Table A.3 of *IEC 61508–1* gives an example of the information and/or documentation needed for a compliant system. This is re-worked here in the context of this Guide. The list below is in life-cycle order.

Software safety requirements. The assumption in this Guide is that the safety issue is the provision of correct measurements. See also section 12.2.

Software validation planning. This issue is covered implicitly by the guidance given.

Software architecture. Not specifically covered, but this could impact validation in a major way if the architecture allowed for components to be treated separately, see section 6.2. A complex architecture can give rise to timing issues, see appendix A.8.

Software system design. Not specifically covered as this is a validation Guide, not a design guide; but see *BPG-Software*.

Software module design. Not specifically covered as this is a validation Guide, not a design guide, but (again) see *BPG-Software*.

Coding. See section 11.2, and the techniques:

- Defensive programming (section 12.7),
- Code review (section 12.8), and
- Static analysis (section 12.6).

Software module testing. See the techniques:

- Component testing (section 12.13),

- System testing (section 12.16),
- Statistical testing (section 12.12), and
- Stress testing (section 12.17).

Software integration. Not specifically covered in this Guide.

Programmable electronic integration.
Not specifically covered in this Guide.

Software operation and maintenance procedures.
Not specifically covered in this Guide.

Software safety validation. The assumption in this Guide is that the safety issue is the provision of correct measurements. On this basis, the Guide provides comprehensive advice throughout.

Software modification. Not specifically covered in this Guide. Validation is assumed to be based on the specific version of the system.

B.3 Auditing checklists

Auditing checklists identify specific questions which should ideally have a positive response. The rationale for the questions and potential reasons for a negative response are given. Each checklist is specific to the MSI for which the technique is applied. However, when a lower Index checklist item is applicable to a higher Index, it needs to be interpreted in a manner appropriate to the higher Index.[1]

The checklist below (for the review of a specification at MSI1) is a general checklist which should be applied to all systems.

B.3.1 Review of informal specification (12.2)

List for MSI1

1. Is the person (or persons) undertaking the review independent[2] of the developers of the specification? 3–7.3.2.4
2. Do the person (or persons) undertaking the review have all the necessary knowledge and skills?
3. Is the level of effort devoted to the review agreed to be adequate, taking into account factors like the knowledge of the reviewer(s), novelty of the system, etc.?
4. Is the elapsed time to deliver the review long enough not to endanger its completeness?
5. If the system involves interaction with the user, has this interaction been checked to ensure that the user is unlikely to be confused?
6. If the measurement device has more than one mode of operation, is this clearly indicated? 3–7.2.2.7

[1] The third item under Review of specification (MSI1) is to check that the level of effort is appropriate; clearly the actual level will be higher when applied to MSI3, for example.

[2] Degrees of independence are defined in *IEC 61508–1*, 8.2.12.

7. Has the operating manual (if required) been checked against the software and checked for completeness?
8. If the system requires a skilled operator, is this clear from the specification?
9. Has the specification been checked against the required standards and regulations?
10. Is manual input this properly acknowledged and checked? (See the text in *IEC 61508–7* clause B.4.9, which is very helpful.)
11. If facilities are available to change the characteristics of the measuring device (like changing calibration data), is this clearly separated and controlled?
12. Are any "errors" in basic sensor data detected or corrected? Is this covered adequately in the specification?
13. Have all concerns raised in the review been resolved (and documented)?

List for MSI2

In addition to the checklist above, add:

1. Have the special skills required to review the specification been documented and is it clear that the reviewing team has those skills? (Such skills are to include software engineering and an area of metrology.)
2. Has at least one member of the review team reviewed a safety system software specification before?
3. If major problems were located on an initial review, was a further review undertaken?
4. Can one be confident that any misunderstanding by the user/operator cannot result in an unsafe action?
5. Does the review confirm the design decisions with respect to protection against operator/user error?

B.3.2 Software inspection of specification (12.3)

List for MSI2

1. Has the specification had a detailed review by those responsible for the specification (prior to the inspection) and had no major outstanding problems?
2. Does the organisation have a procedure to cover software inspection, or does it have a documented history of undertaking such inspections?
3. Have all concerns raised in the inspection been resolved (and documented)?

List for MSI3

In addition to the checklist above, add:

1. Prior to the inspection, have difficult areas of the specification been identified and have appropriate people been brought into the inspection team to cover these areas?
2. Is it true that no unreasonable barriers have been placed upon involving external experts on the software inspection?

B.3.3 Mathematical specification (12.4)

List for MSI1

1. Those aspects of the system which can be expressed by means of appropriate mathematical formulae, are they included in the specification of the software?
2. Is the range of the validity of the those formulae specified?
3. Have all the issues raised in section 3.2 been resolved?

List for MSI2

In addition to the checklist above, add:

1. Is the relationship between the basic sensor data and the resulting measurement specified by mathematical formulae?
2. Are the relevant mathematical formulae those derived from the underlying physics of the device?
3. Is it true that there are no gaps in the underlying model, see section 3.3?
4. Is it true that there are no essential difficulties in converting the formulae into a computational form?

List for MSI3

In addition to the checklists above, add:

1. Has the mathematical model used been validated, see section 3.3.

List for MSI4

In addition to the checklists above, add:

1. Is the validated mathematical model widely used, see section 3.3?

The above is to be reviewed in a later edition of the Guide.

B.3.4 Formal specification (12.5)

There is no additional guidance available at this time. An interesting possibility would be to include model checking since this has gained in popularity since *IEC 61508* was produced.

B.3.5 Static analysis/predictable execution (12.6)

In this Guide, we do not make any recommendations about a "suitable programming language", although highly recommended advice is to be found in *ATS-Language–1* and *ATS-Language–2*. Instead, we recommend static analysis to show that the program behaves predictably. Static analysis can also be used to show that vital safety conditions are satisfied.

Note that the development environment as a whole is considered in section 11.2.

Static analysis can be used to perform boundary value analysis.

List for MSI2

The application of static analysis is quite difficult due to the second item on the list below. Hence, although recommended, at this Index, it may be more practical to rely on Code review, see section 12.8.

1. Are the semantics of the programming language well-understood?

2. Will the main tool used to undertake static analysis locate almost all cases in which the program execution could be unpredictable? (This requires a detailed case-analysis.)

3. Are there no reservations on the analysis of access to unassigned variables? (Conventional analysis only handles simple variables, and hence an element of an array or record could still be unassigned.)

4. Are obscure cases in which the tool may not detect unpredictable execution properly documented? (This could involve, for instance, a function with a side-effect.)

List for MSI3

In addition to the checklist above, add:

1. Are the semantics of the programming language defined by a national or international standard, or by a formal definition?

2. Are all the areas in which a compiled program can behave unpredictably properly documented?

3. Does the development have a defined approach to detecting unpredictable behaviour?

List for MSI4

In addition to the checklists above, add:

1. Are all critical safety properties discharged by formal proof or appropriately documented reasoning?

The above is to be reviewed in a later edition of the Guide.

B.3.6 Defensive programming (12.7)

List for MSI1

1. Is there a defined strategy on the use of defensive programming?

2. When a defensive programming check fails, what is assumed (if anything) of the state of the variables and the external environment?

List for MSI2

In addition to the checklist above, add:

1. Are all reasonable sources of unpredictable execution excluded by means of defensive programming?

2. Are physically impossible/unlikely values excluded by means of defensive programming?

In spite of being recommended in *IEC 61508* for all Indexes, defensive programming is not included at the higher Indexes for the following reason. If a specific situation should be detected at run-time, then this should be part of the specification — perhaps to give a degraded reading "dazzled", say.[3] This property could be important in a determination of the correctness of the measurement which could have serious safety consequences. On the other hand, if the situation cannot arise due to the design characteristics, then this should be proven by static analysis.

B.3.7 Code review (12.8)

List for MSI1

1. Is there a coding standard to review against?

2. Is there a review list of aspects to be covered by code-review?

3. Are "tricky" language features subject to a special review or explicitly prohibited?

4. Is the person undertaking the review experienced in the language and (preferably) in maintaining code in the language?

List for MSI2

In addition to the checklist above, add:

1. Have sources of known problems in the language been identified to ensure that all known pitfalls are avoided?

In spite of being recommended at all Indexes in *IEC 61508*, code review is not recommended here at the higher Indexes. The reason for this is that any technique used at the higher Indexes needs to be shown to have a measurable effect on the quality of the software. This is hard with code review, which is essentially a manual method.

As a means of gaining confidence in the code, the use of static analysis is thought to be *more* appropriate at SIL3 and SIL4.

Those developing safety systems at SIL3 and SIL4 may well feel that they should follow the recommendation of *IEC 61508* and undertake code review anyway. However, there is a view that attempting code review with the thoroughness appropriate to SIL3 and SIL4 may well not be cost-effective.

[3] The term "dazzled" is used to refer to situations in which a correct measure may not be possible, as in Fieldbus and similar standards.

B.3.8 Numerical stability (12.9)

List for MSI2

1. Is the person undertaking an analysis for numerical stability competent? (This subject is not covered by most undergraduate courses; for example, the person must understand the term "backward error analysis".)

List for MSI3

In addition to the checklist above, add:

1. Has the numerical analysis been formally reviewed by an independent analyst?

2. Have appropriate test cases for the worst-case of rounding errors been produced and applied successfully?

List for MSI4

In addition to the checklists above, add:

1. Has back-to-back testing been applied, or acceptance testing based upon the numerical analysis?

B.3.9 Microprocessor qualification (12.10)

List for MSI4

This is to be considered in a later edition of the Guide.

B.3.10 Verification testing (12.11)

A degree of independence for the testing team is essential for this verification testing to be effective, bearing in mind the other testing undertaken.

The assumption here is that the testing is performed with the aid of some test harness to ensure the exact repeatability, to ensure there is no dependence upon inexact external stimuli.

List for MSI3

1. Does the testing team have both the independence and expertise necessary?

2. Have targets been agreed for the test coverage to be obtained from the tests devised entirely under this activity?

3. Is the level of effort in producing the tests at least comparable with that needed to obtain branch coverage?

List for MSI4

In addition to the checklist above, add:

1. Is the testing carried out "blind", i.e., the expected results produced without the benefit of the target software?
2. Have all the unexpected results which the testing team regard as their fault been checked against the documentation?

This checklist is to be reviewed in a later edition of this Guide.

B.3.11 Statistical testing (12.12)

The assumption here is that the tests are exactly repeatable, presumably due to the use of the test harness.

List for MSI2

1. Is there a sound basis of the statistically chosen test data?
2. Is there a clear reason for the selection (number) of test cases?
3. If test data is generated automatically, can one be assured of its statistical properties?
4. Is there a documented method for determining the expected results?

List for MSI3

In addition to the checklist above, add:

1. Does each test produce extra (diagnostic) data, apart from pass/fail?

B.3.12 Component testing (12.13)

The various forms of component testing are taken from *BS 7925*, and for MSI1, the recommended technique is "structural testing".

List for MSI1

1. Has the chosen form of structural testing been documented?
2. Are there means to ensure that appropriate test cases have been used and the results validated?

   One could instrument the code, use a tool to undertake this, or rely upon an analysis of the specification (for simple software).

List for MSI2

At this Index, statement testing and boundary value testing are recommended.
In addition to the checklist above, add:

1. Has adequate coverage been obtained with both types of component testing?
2. For those statements which were not executed, were all covered by the use of defensive programming?

List for MSI3

At this Index, the reason for each statement that has not been executed must be fully documented. At this Index, branch testing is recommended, along with the testing appropriate to MSI2.

In addition to the checklists above, add:

1. Has adequate coverage been obtained with branch testing?

List for MSI4

This is to be considered in a later edition of the Guide and is likely to include recommendations to use Modified Condition/Decision testing, branch testing, and boundary value testing.

B.3.13 Regression testing (12.14)

There is no specific recommendation for use of regression testing at any Index, but see section 11.2.

B.3.14 Accredited software testing using a validation suite (12.15)

List for MSI2

1. Does the validation suite have some kind of formal recognition?
2. Can the validation suite be considered (to some extent) a complete functional test of the software?

B.3.15 System-level software functional testing (12.16)

List for MSI1

1. Can the tests be considered (to some extent) a complete functional test of the software?
2. Has the specification been analysed to ensure all the basic functionality is covered?

List for MSI2

In addition to the checklist above, add:

1. Has some form of structural coverage been measured to ensure that the functional tests are complete in the sense of covering the code?

B.3.16 Stress testing (12.17)

List for MSI2

1. If the tests are machine-generated, do these tests sufficiently cover the functionality of the software?
2. Is it documented how the expected results for checking the correct result from the software are produced?

List for MSI3

In addition to the checklist above, add:

1. Does each test produce extra (diagnostic) data, apart from pass/fail?

B.3.17 Numerical reference results (12.18)

List for MSI1

1. Have the reference data-result pairs been validated?
2. Do the tests sufficiently cover the functionality of the software from the numerical analysis standpoint?

List for MSI2

In addition to the checklist above, add:

1. Have the numerical reference results been used to validate other software?

B.3.18 Back-to-back testing (12.19)

List for MSI2

1. Except for communications software, are the two implementations independent?

   For communications software, is the relationship between the two implementations clear and appropriate?
2. If one implementation is a reference implementation, is its status clear and acceptable?
3. Are differences in rounding error between the two implementations handled appropriately? (The difference should be less than the claimed accuracy of the measurements.)

List for MSI3

In addition to the checklist above, add:

1. Is adequate code coverage obtained with this back-to-back testing?
2. If verification testing is performed, is both the selection of the tests to be applied with back-to-back testing, and the coverage of those tests, acceptable?

B.3.19 Comparison of the source and executable (12.20)

List for MSI4

1. Have the papers on this topic been studied and an appropriate application designed?
2. Has an appropriate level of assurance been specified?
3. Is reliability data available for the compiler in order to assess the risks?

This checklist is to be reviewed in a later edition of this Guide.

B.4 Traceability

This section considers how *IEC 61508* relates to this Guide. The Guide has been written to aid the validation of measurement software which can be seen to satisfy the objective of quality measurement. Hence the relationship with the generic safety standard is not always clear. Those who already know *IEC 61508* may find this section useful.

B.4.1 Recommended techniques

The techniques from *IEC 61508–7* are considered in the order in which they appear in that document.

Many techniques are not used in this Guide and the reasons for this are stated here. This is to be expected, since the scope of the techniques in the standard covers entire systems — hardware and well as software. Even software development methods are not necessarily used in the Guide unless they have a direct bearing on the software validation process.

Techniques such as *structured programming* are very widely used and effective, but only have an indirect impact on validation and therefore are not considered in this Guide. Such techniques are marked **Not critical to validation** in the list below, even when they are marked as "HR" (for "highly recommended") in one of the tables in *IEC 61508*. Suppliers may well wish to claim compliance with such techniques, but this Guide does not provide any assistance in them. For development guidelines for measurement software, see *BPG-Software*.

In some cases, techniques which are thought to be very effective for validation are not handled in this Guide due to their lack of widespread use.

7–B.1.1 Project management: Required, since this Guide assumes *ISO 9001* or an equivalent quality management system is in use.

7–B.1.2 Documentation: There are extensive documentation requirements in *IEC 61508*. The certification process of the software covered in this Guide requires documentary support for the auditing questions/checklists given above. See section B.2.

7–B.1.3 Separation of safety-related systems from non safety-related systems: This is not a requirement of this software validation. However, if separation is used, the validation costs could well be reduced. For an example of this, see section 6.2. (The separation is not a requirement of *IEC 61508*, merely a recommendation.) **Not critical to validation.**

7–B.1.4 Diverse hardware: Outside the scope of this Guide. The safety integrity of the hardware needs to be assessed for certification, but again, this is outside the scope of this Guide.

7–B.2.1 Structured specification: It is not useful to assess a specification for its structure alone. Clearly, a poorly structured specification will make the entire development more difficult. It is also likely to make validation more expensive. **Not critical to validation.**

7–B.2.2 Formal methods: See 7–C.2.4 below.

7–B.2.3 Semi-formal methods: This does not seem to be a technique which can be audited for the purposes of validation.

7–B.2.3.2 Finite state machines/state transition diagrams: This can be a very useful technique to ensure the completeness of a specification, which can be carried through to test cases. There are some tools for this technique which can aid the development. At this stage, it is not one of the chosen techniques in this Guide. If there is a request for it to be added, this could be considered in a later edition of the Guide.

7–B.2.3.3 Time Petri nets: This technique is rather specialised and it is not thought worthwhile to consider it here.

7–B.2.4 Computer aided specification tools: The approach taken in this Guide is to consider the specification, rather than how it was developed. Of course, such tools can be very useful, especially in aiding traceability.

7–B.2.4.2 Tools oriented towards no specific method: This technique does not provide a method of checking its application to aid validation.

7–B.2.4.3 Model orientated procedure with hierarchical analysis: This technique does not provide a method of checking its application to aid validation. The reference to a paper of 1977 may indicate a method not in current use.

7–B.2.4.4 Entity models: This could certainly be an effective specification method, and the proposals in this Guide would allow the use of this technique.

7–B.2.4.5 Incentive and answer: This method does not seem to provide the clarity or precision needed for safety systems.

7–B.2.5 Checklists: General checklists are not useful since they cannot be effectively checked for completeness. On the other hand, this Guide uses prepared ones to identify issues which should be addressed. In other words, checklists can be useful but need to be assessed with other material.

7–B.2.6 Inspection of the specification: Appears to be Fagan Inspection, in which case, the reference to Myers is incorrect. This method is used directly in this Guide, see section 12.3. It is unclear how this relates to 7–C.5.15.

7–B.3.1 Observance of guidelines and standards: Only specific standards and guidelines are considered in this Guide.

7–B.3.2 Structured design: It is not useful to assess a design for its structure alone. Clearly, a poorly structured design will make the entire development more difficult. It is also likely to make validation more expensive. **Not critical to validation**.

7–B.3.3 Use of well tried components: As specified in *IEC 61508*, this seems to be a hardware technique and therefore not relevant. The use of such software components is not considered here, but see *SOUP–1, SOUP–2*.

7–B.3.4 Modularisation: The same remarks apply as for "structured design" (7–B.3.2, above), and so modularisation is therefore not used in this Guide. **Not critical to validation**.

7–B.3.5 Computer aided design tools: This is surely useful but not considered in this Guide.

7–B.3.6 Simulation: As described in the standard, this seems to be a hardware technique and therefore not relevant to this Guide. However, there is one application that is likely to be vital. To provide repeatable tests for a software-based instrument, it is essential to have a source of repeatable test cases which in turn means simulating the environment for the software.

7–B.3.7 Inspection (reviews and analysis): Not considered in the form specified in *IEC 61508*, but is related to the techniques that are used: software inspection of the specification (see section 12.3) and static analysis (see section 12.6).

7–B.3.8 Walk-through: This is potentially a very good technique. Unfortunately, it is very difficult to provide convincing evidence of its effective use. In consequence, it cannot be audited and therefore is not effective for the validation process used here.

7–B.4.1 Operation and maintenance instructions: This is part of required documentation — but this seems to apply to the hardware rather than the software (and hence is not relevant here).

7–B.4.2 User friendliness: This can be important if a misunderstanding can result in an unsafe action. See section B.3.1.

7–B.4.3 Maintenance friendliness: As worded in *IEC 61508*, this seems to be hardware-oriented. Obviously, not relevant for embedded software as user maintenance is not possible.

7–B.4.4 Limited operation possibilities: This appears to be a design technique to reduce possibilities during "ordinary" operation. It is unclear if this applies to safe measurement.

7–B.4.5 Operation only by skilled operators: This is common to the complex measurement devices used in metrology and covered by this Guide.

7–B.4.6 Protection against operator mistakes: This is related to the previous item. See section B.3.1.

(7-B.4.7 No such technique is listed: it was deleted from an earlier draft.)

7–B.4.8 Modification protection: Specific requirements for protection against unauthorised modification appear in the area of legal metrology (see section 2, on page 8). This is covered by ensuring that the specification review covers required standards and regulations.

7–B.4.9 Input acknowledgement: A checklist item has been included for this, see section B.3.1.

7–B.5.1 Functional testing: This Guide uses two forms of functional testing — *system testing*, see section 12.16 for the complete system, and *verification testing*, see section 12.11 for comprehensive testing of the software alone.

7–B.5.2 Black-box testing: The same as remarks apply as for 7–B.5.1, above.

7–B.5.3 Statistical testing: This is used in this Guide, see section 12.12.

7–B.5.4 Field experience: The view taken here is that this can only be undertaken for the whole system, hardware and software and is therefore outside the scope of this Guide. For its use for software alone, *SOUP–1* and *SOUP–2* could be consulted.

7–B.6.1 Functional testing under environmental conditions: Not relevant as it is a hardware issue.

7–B.6.2 Interference surge immunity testing: Not relevant as it is a hardware issue. However, see "memory protection" on section 9.3.

(7-B.6.3 No such technique is listed: it was deleted from an earlier draft.)

7–B.6.4 Static analysis: This Guide puts a lot of emphasis on static analysis for the higher MSIs, see section 12.6.

7–B.6.5 Dynamic analysis: This technique is not very clear. Several forms of testing appear in this Guide which may be thought of as "dynamic analysis", but it is unclear which are intended in the standard.

7–B.6.6.1 Failure modes and effects analysis: This key hardware technique is important for the overall system, but has little relevance to the software. Hence it is not used in this Guide.

7–B.6.6.2 Cause consequence diagrams: This is not relevant as it is a hardware technique.

7–B.6.6.3 Event tree analysis: This is not relevant as it is a hardware technique.

7–B.6.6.4 Failure modes, effects and criticality analysis: This is not relevant as it is a hardware technique.

7–B.6.6.5 Fault tree analysis: This is not relevant as it is a hardware technique.

7–B.6.7 Worst-case analysis: This is not relevant as it is a hardware issue. However, boundary value testing (see section 12.13) and boundary value analysis (see section 12.6) are included.

7–B.6.8 Expanded functional testing: This appears to be part of functional testing, see 7–B.5.1. The "rare" events covered under this heading should be part of verification testing, see section 12.11.

7–B.6.9 Worst-case testing: Ignoring the environmental aspects of this technique given in *IEC 61508*, the technique for software is the same as boundary value testing (see section 12.13).

7–B.6.10 Fault insertion testing: Generally, this is not relevant as it is a hardware issue. However, this is not strictly correct in that mutation analysis has been applied to software but found to be too expensive in realistic cases.

7–C.2.1.2 Controlled Requirements Expression (CORE): This technique is not explicitly covered, but the resulting specification can be reviewed according to the techniques in this Guide.

7–C.2.1.3 JSD — Jackson System Development: This technique is not explicitly covered, but the resulting specification can be reviewed according to the techniques in this Guide.

7–C.2.1.4 MASCOT: This technique is not explicitly covered, but the resulting specification can be reviewed according to the techniques in this Guide. In fact, this technique is not now widely used.

7–C.2.1.5 Real-time Yourdon: This technique is not explicitly covered, but the resulting specification can be reviewed according to the techniques in this Guide.

7–C.2.1.6 SADT — Structured Analysis and Design Technique: This technique is not explicitly covered, but the resulting specification can be reviewed according to the techniques in this Guide.

7–C.2.2 Data flow diagrams: This technique is not explicitly covered, but the resulting specification can be reviewed according to the techniques in this Guide.

7–C.2.3 Structure diagrams: This technique is not explicitly covered, but the resulting specification can be reviewed according to the techniques in this Guide.

7–C.2.4 Formal methods: In practice, one needs to be assured that the users understand the specification produced and that the specification has been effectively and independently audited. This is likely to be effective for MSI4 and hence consideration is delayed until this Index is handled more fully. Model checking may well be an appropriate method which is not mentioned in *IEC 61508*.

7–C.2.4.2 CCS — Calculus of Communicating Systems: Potentially, a very strong technique for the validation of concurrent systems, but it is not considered in this Guide since it is thought not to be in widespread use. It could be considered later, if requested.

7–C.2.4.3 CSP — Communicating Sequential Processes: Potentially, a very strong technique for the validation of concurrent systems, but is not considered in this Guide since it is thought not to be in widespread use. It could be considered later, if requested.

7–C.2.4.4 HOL — Higher Order Logic: It could be considered later, if requested.

7–C.2.4.5 LOTOS: Potentially, a very strong technique for the formal specification of concurrent systems, but is not considered in this Guide since it is thought not to be in widespread use. It could be considered later, if requested.

7–C.2.4.6 OBJ: It could be considered later, if requested.

7–C.2.4.7 Temporal logic: It could be considered later, if requested.

7–C.2.4.8 VDM, VDM++, — Vienna Development Method: Potentially, a very strong technique for the validation of sequential systems, but is not considered in this Guide since it is thought not to be in widespread use. It could be considered later, if requested.

7–C.2.4.9 Z: Potentially, a very strong technique for the validation of sequential systems, but is not considered in this Guide since it is thought not to be in widespread use. It could be considered later, if requested.

7–C.2.5 Defensive programming: As worded in *IEC 61508*, this is a design method and the implications for testing are not mentioned. In this Guide it is both a design method and has implications for testing and static analysis, see section 12.7.

7–C.2.6.2 Coding standards: Coding standards should be used in a code review, see section 12.8. This is clearly related to the area of language subsets for which *ATS-Language–1* and *ATS-Language–2* should be consulted. This is partly considered in section 11.2.

7–C.2.6.3 No dynamic variables or dynamic objects: This is poorly specified in *IEC 61508*, since surely stack variables are permitted. This issue is important at MSI4 and will be addressed in a later edition of the Guide. This is partly considered in section 11.2.

7–C.2.6.4 Online checking during creation of dynamic variables: It is not clear what is intended.

7–C.2.6.5 Limited use of interrupts: The intent is clear, but the wording is confusing. Event programming with modern interactive systems is certainly to be avoided. With MSI4 systems, the typical absence of an operating system implies a different approach. To be addressed in a later edition of the Guide.

7–C.2.6.6 Limited use of pointers: The intent is clear, but the wording is confusing. To be addressed with MSI3/MSI4 systems as part of static analysis. This is a major problem with C unless a very small subset is used. Only one static analysis tool appears to be able to analyse a C program adequately in this area. Considered in section 11.2.

7–C.2.6.7 Limited use of recursion: The intent is clear, but the wording is confusing. To be addressed with MSI3/MSI4 systems as part of static analysis. Conventional practice is to prohibit recursion rather than permit it to a limited depth.

7–C.2.7 Structured programming: Structured programming is, of course, highly recommended; but what is "statistically untestable behaviour"? This is covered in this Guide by means of the environment (see section 11.2), code review (see section 12.8) and static analysis (see section 12.6). Tools which produce complexity measures are regarded here as management tools rather than having a direct bearing on validation. **Not critical to validation**.

7–C.2.8 Information hiding/encapsulation: This is useful for program development. **Not critical to validation**.

7–C.2.9 Modular approach: A modular approach is, of course, highly recommended; but not included explicitly, as for structured programming (7–C.2.7, above). **Not critical to validation**.

7–C.2.10 Use of trusted/verified software modules and components: As worded, this is highly dubious since it says "not been formally or rigorously verified, but for which considerable operational history is available". Fortunately, the numbers given on the data from operations seem unlikely to be met in practice. This could be considered if requested, using *SOUP–1* and *SOUP–2*.

Note that Annex D of *IEC 61508–7* is useful here.

7–C.3.1 Fault detection and diagnosis: This appears to require diversity for software which does not have consensus support in the industry. Otherwise it seems to be merely part of the design and not relevant for validation. However, it is related to defensive programming, see section 12.7.

7–C.3.2 Error detecting and correcting codes: This is not relevant as it is a hardware issue. However, software components should use defensive programming (see section 12.7) to ensure preconditions are satisfied. In the context of legal metrology, see *WELMEC–7.1*.

7–C.3.3 Failure assertion programming: This is just defensive programming, see clause C.2.5 of the standard, and section 12.7 of this guide.

7–C.3.4 Safety bag: This method is not believed to be in widespread use and hence is not included.

7–C.3.5 Software diversity (diverse programming): The problem with software diversity is that it doubles the development and validation costs. In consequence, it is not in widespread use and is not considered in this Guide.

7–C.3.6 Recovery block: It is unclear if this technique is in widespread use and hence is not included.

7–C.3.7 Backward recovery: It is unclear if this technique is in widespread use and hence is not included.

7–C.3.8 Forward recovery: It is unclear if this technique is in widespread use and hence is not included.

7–C.3.9 Re-try fault recovery mechanisms: This is typically built into many data communication protocols, but would only be within the scope of this Guide if an instrument used such a protocol. Not explicitly handled at this stage.

7–C.3.10 Memorising executed cases: No use of this technique is known and hence is not included in this Guide.

7–C.3.11 Graceful degradation: May be part of the specification, but does not need to be considered separately here. One could have a measurement system consisting of several sensors, some of which could "fail" without closing down the entire system.

7–C.3.12 Artificial intelligence fault correction: It is unclear that this would ever be used in measurement systems — hence not included in this Guide. Any method of fault correction clearly needs to be very carefully reviewed.

7–C.3.13 Dynamic reconfiguration: It is unclear that this would ever be used in measurement systems — hence not included in this Guide.

7–C.4.1 Strongly typed programming languages: The use of such languages is good software engineering practice, but the importance of using then depend on the degree to which the language impacts static analysis for MSI3 and MSI4, see section 12.6. See also section 11.2.

7–C.4.2 Language subsets: This is important at MSI4 and hence not handled at this stage. At that Index, it is probably necessary to restrict this to Ada [*ISO 15942*] and C [*MISRA-C*]. See section 11.2.

7–C.4.3 Certified tools and certified translators: Very few tools can be effectively certified. This is only a real problem at MSI3–4, and then fitness-for-purpose should be the objective. This issue needs to be considered more specifically for MSI4 in a later edition of the Guide.

7–C.4.4 Translator: increased confidence from use: This is only a real problem at MSI3–4, and then fitness-for-purpose should be the objective. However, the issue is partly handled by source-code to executable comparison, see section 12.20.

7–C.4.4.1 Comparison of source program and executable code: See section 12.20.

7–C.4.5 Library of trusted/verified software modules and components: Not currently considered, but see *SOUP–1* and *SOUP–2*.

7–C.4.6 Suitable programming languages: This issue is not considered in this Guide, although it does have a significant impact on static analysis, see section 12.6. However, *ATS-Language–1* and *ATS-Language–2* are recommended (but note that since the publication of those reports, *ISO 15942* has been published). See also section 11.2.

7–C.5.1 Probabilistic testing: See 7–B.5.3, but this is included in the Guide, see section 12.12.

7–C.5.2 Data recording and analysis: Documentation is a requirement of *IEC 61508*, but this specific technique in not included.

7–C.5.3 Interface testing: The important part of this is included in static analysis, see section 12.6. The dynamic aspect of this is included in defensive programming, see section 12.7.

7–C.5.4 Boundary value analysis: The important part of this is included in static analysis, see section 12.6.

7–C.5.5 Error guessing: Unfortunately, this technique is not repeatable and hence is not included here.

7–C.5.6 Error seeding: This is not practical technique for systems of a realistic size.

7–C.5.7 Equivalence classes and input partition testing: Included in the Guide, see section 12.13.

7–C.5.8 Structure based testing: Included in the Guide, see section 12.13.

7–C.5.9 Control flow analysis: This is part of static analysis, see section 12.6.

7–C.5.10 Data flow analysis: This is part of static analysis, see section 12.6

7–C.5.11 Sneak circuit analysis: This technique is not thought to be in widespread use and hence is not included.

7–C.5.12 Symbolic execution: Insofar as this is useful, it is part of static analysis, see section 12.6.

7–C.5.13 Formal proof: This is too difficult for consideration at anything but MSI4: it will be considered in a later edition of the Guide. Model checking is not mentioned in the standard, but could be very effective in checking complex logic.

7–C.5.14 Complexity metrics: These are regarded as a management tool, not relevant to validation.

7–C.5.15 Fagan inspections: These are included in this Guide under "software inspection", see section 12.3.

7–C.5.16 Walk-throughs/design reviews: Walk-throughs are not included, but reviews via Fagan-style inspections are; see the previous item.

7–C.5.17 Prototyping/animation: This technique is to aid development and hence is not directly relevant for validation.

7–C.5.18 Process simulation: The testing of the measurement software in a repeatable fashion requires separation from the hardware, and in that sense, process simulation is a requirement of verification testing, see section 12.11.

7–C.5.19 Performance requirements: Specific performance requirements are part of functional testing, see 7–B.5.1.

7–C.5.20 Performance modelling: This technique is to aid development and hence is not directly relevant for validation.

7–C.5.21 Avalanche/stress testing: This is included in this Guide, see section 12.17.

7–C.5.22 Response timing and memory constraints: For MSI3–4 systems, this is effectively part of static analysis, see section 12.6.

7–C.5.23 Impact analysis: For a single validation, this is not relevant. However, small changes are often made to systems which then require that all the necessary testing is repeated unless impact analysis can show that fewer tests will suffice. This issue is not currently addressed, but will be considered in a later edition of the Guide.

7–C.5.24 Software configuration management: This is required as part of *ISO 9001*, see also section 11.2.

7–C.6.1 Decision tables (truth tables): There is no need to consider this separately from a system specification.

7–C.6.2 Hazard and Operability Study (HAZOP): This technique is typically hardware-based and used for the entire system. It is not clear that this Guide needs to consider it. Of course, the consequences of a software error need to be taken into account in determining the SIL. However, ensuring a robust design is very important, see section 9.3.

7–C.6.3 Common cause failure analysis: This technique is typically hardware-based and used for the entire system. It is not clear that this Guide needs to consider it.

7–C.6.4 Markov models: This technique is typically hardware-based and used for the entire system. It is not clear that this Guide needs to consider it.

7–C.6.5 Reliability block diagrams: This technique is typically hardware-based and used for the entire system. It is not clear that this Guide needs to consider it. It could be used for software, but this is not considered here.

7–C.6.6 Monte-Carlo simulation: As described in the standard, this technique is typically hardware-based and used for the entire system. It is not clear that this Guide needs to consider it. The method could be used to generate test cases, perhaps as part of stress testing, see section 12.17.

Monte-Carlo simulation is also used to validate uncertainty evaluations. As such, it is relevant to measurement, but not to validation of measurement systems — apart from any software specifically designed to evaluate uncertainties.

B.4.2 Guide techniques

IEC 61508 does not consider the following Guide techniques:

Regression testing. The word "regression" does not appear in Parts 3 or 7. In any case, one cannot release unsafe systems and wait for problems! Hence, in this edition of the Guide (which considers safety), this method is not applied in the context of *IEC 61508* validation. Note that it is referenced in section 11.2.

Accredited testing. The word "accredited" does not appear in Parts 3 or 7. However, the method does have a role to play as the example in the Guide shows (see appendix C.3).

Numerical stability. This is not mentioned in Parts 3 or 7, which seems to be a serious failing. Of course, most measurements made in a safety context do not quite high accuracy, but unless numerical accuracy is considered, there is a danger that all accuracy is lost.

Mathematical specification. This seems to be implicit:

> "The software safety requirements specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application."

Hence, in the measurement area it would appear that *IEC 61508* almost requires this!

Numerical reference results. This is essentially an NPL method (see section 12.18) and hence a reference would not be expected. The usefulness and effectiveness of this method clearly shows the need for it becoming more widely known and used.

Back-to-back testing. This is not referenced explicitly but is implicit in "diverse software", see Part 7: C.3.5. The technique can be used to provide an oracle for stress testing (12.17) or statistical testing (12.12).

Since *IEC 61508* is generic, and not specific to measurement systems, it does not consider issues such as the validity of the underlying physics or the accuracy with which the physics can be modelled; see chapter 3.

B.4.3 Requirements

In *IEC 61508*, as in all International Standards, the requirements are specified with a *shall*. This section considers those requirements and how they relate to recommendations in this Guide.

In this Guide, we are only considering the validation of the software which implies that many aspects of *IEC 61508* are not considered at all.

To simplify issues, consider the design of a measuring instrument being undertaken by a single company to satisfy *IEC 61508*. The user, responsible for a safety application, purchases the instrument, making all reasonable checks that it satisfies the requirements of the application. This implies that the following must be specified:

- The SIL for the instrument.
- The accuracy requirement for the safety application. The actual instrument is likely to be much more accurate to give a large safety margin.

- The MSI using this Guide.
- A validation plan, again based upon this Guide.

The directly applicable component of *IEC 61508* is Part 3. Hence, we list the requirements of Part 3 and show how they relate to the Guide. The Guide itself is annotated with references to the items below. Part 3 needs to be consulted for the details of each requirement.

3–1.1 d) Safety life-cycle: The primary responsibility for this life-cycle must rest with the user. The user must ensure that the suppliers' specification satisfies the requirements. Independent certification to *IEC 61508* for the instrument would provide assurance to the user of the suppliers' claims. Certification would typically be based upon confidential information provided by the supplier which would not necessarily be available to a user.

The whole Guide is relevant to this life-cycle.

3–6.2.2 Functional safety planning: Assuming that the instrument satisfies its requirements, the user needs to ensure that, as a consequence, the total system will satisfy the safety requirements. This aspect is not covered in this Guide.

Note that the accuracy of the instrument may be crucial to the safety assessment. Due to a safety margin, the claimed accuracy is likely to be much less than that of the basic instrument as specified in non-safety applications.

If calibration of the instrument is important to maintain the required accuracy, then the calibration process needs to be suitably controlled, see *EUR 19265*.

3–7.1.2.1 Software development: This Guide takes the approach that the development process should be appropriate to the MSI, see section 6.2.

3–7.1.2.2 Quality procedures: See sections 7.2 and 11.1. Note that some of the documentation requirements could be met by reference to this Guide.

3–7.1.2.3 Activities of life-cycle: See figure 6.1.

3–7.1.2.6 Techniques: The standard does not mandate specific techniques, but gives recommendations instead. In the context of measurement software, it would be appropriate to follow the recommendations given here instead, see table 11.2.

3–7.1.2.7 Documentation of activities: See section B.2.

3–7.1.2.8 Repeat changed activities: The need to repeat an activity due to a change to the software is not specifically covered in this Guide, but is implicit in the validation process, see figure 6.1.

3–7.2.2.2 Derivation of safety requirements: The implicit assumption here is that the reliability and accuracy of the instrument are the key safety requirements. Aspects such as the ease of use for calibration are not specifically addressed, but could be crucial in certain circumstances. See chapter 6.

3–7.2.2.3 Detail of safety requirements: See the list of six key aspects in chapter 6.

3–7.2.2.4 Software developer's review and check list: The list includes the response time for the instrument. This is typically not a problem but needs to be checked. For the highest Index(MSI4), it would be necessary to compute the worst case execution time.

3–7.2.2.5 Software procedures: It is important that the supplier and the user should agree on the SIL of the instrument. Clearly, if the supplier rates the instrument at a lower Index than that required by the user, safety assurance will not be provided. See table 6.1.

3–7.2.2.6 Precision of requirements: The precision clearly requires the use of established physics, see section 3.3.

3–7.2.2.7 Detail of modes of operation: It can be very important to check that a mode of operation for re-calibrating the instrument is properly checked. Simple manual errors should not permit a system to become unsafe.

3–7.2.2.8 Interaction of hardware and software: Not specifically covered in this Guide.

3–7.2.2.9 Self-test: This topic is not explicitly covered in this Guide since the self-test would typically be applied to the hardware. The software that performs testing of the hardware might well be difficult to validate. It is clearly important that such software should not "hide" serious faults in the hardware.

3–7.2.2.10 Non-safety functions: This is not specifically covered in this Guide.

3–7.2.2.11 Measuring instrument functions: This is not relevant in this context since the previous sections (3-7.2.2.1/10) have already been considered for the measuring instrument — there is no direct equivalent of "the project".

3–7.3.2.1 Validation planning: See figure 6.1.

3–7.3.2.2 Validation planning checklist: The checklist in the standard should be used when reviewing the validation report.

3–7.3.2.3 Choice of techniques: See table 11.2.

3–7.3.2.4 Assessor review and witness: It is essential that this issue is resolved for any formal certification, see B.1.

3–7.3.2.5 Pass/fail criteria: Each validation technique will have its own pass/fail criteria. For software testing, only limited assurance can be obtained by testing the instrument as a whole, see 5.1. Hence it is necessary to test the software in isolation so that the input and the results are completely predictable and do not vary with the external environment.

3–7.4.2.1 Division of responsibilities: This is not covered in this Guide.

3–7.4.2.2 Design methods and checklist: Design methods are not explicitly covered in this Guide. However, if this checklist is not followed, then validation is likely to be much more difficult and expensive. For the highest Index(MSI4), it may be impossible to obtain the assurance needed.

3–7.4.2.3 Testability and safe modification: The ability to test the software in isolation should imply that retesting can be repeated at a low cost (by a suitable script). The assumption here is that the validation is repeated after any modification is performed.

3–7.4.2.4 Design for modification: This really seems to require an intelligible design — vital for successful validation.

3–7.4.2.5 Unambiguous definition: This requirement is very demanding when taken seriously. An unambiguous definition typically implies the use of mathematics (see sections 12.4 and 12.5) and very little dependence upon natural language which can easily be ambiguous.

3–7.4.2.6 Minimise safety part: This requirement appears to imply that the design is for a system which is only for safety. In the case of a measuring instrument, the primary market may be non-safety applications, which therefore gives rise to a potential conflict in the design requirements.

3–7.4.2.7 Independence of non-safety part?: This issue does not seem relevant to most instruments, since the entire system would have to be regarded as safety-critical when used in a safety context.

3–7.4.2.8 Using different safety integrity levels: For an example of more than one level , see section 6.2.

3–7.4.2.9 Proof tests: It appears that "proof tests" in the context of measuring instruments are replaced by calibration — periodic tests to ensure that the instrument functions as required. Hence this is a hardware issue, although some software to support the process will be needed.

3–7.4.2.10 Monitoring control and data flow: This seems to refer to run-time checks (see section 11.2) and defensive programming (see section 12.7).

3–7.4.2.11 Previously developed software: See *SOUP–1* and *SOUP–2*.

3–7.4.2.12 Data validation: In the context of measuring instruments, the handling of calibration data can be very important, see *EUR 19265*.

3–7.4.3.1 Documented responsibilities: These are not covered in this Guide.

3–7.4.3.2 Information and checklist about components: This is probably not relevant in the sense that a measuring instrument is likely to be regarded as a single component. The issue of multi-component (possibly distributed) measurement systems should be addressed in a later edition of the Guide.

3–7.4.3.3 Changes agreed and documented: If the basic sensor is supplied to the instrument maker, then it would be important that any change to the sensor results in a formal acceptance of the change, which might require re-validation of the software.

3–7.4.4.1 Division of responsibility for conformance: In the context of this Guide, responsibility for the specific application must rest with the user, while the claims made for an instrument lie with the supplier. The supplier can in turn use evidence of the suitability of an instrument for a safety application by obtaining formal certification.

3–7.4.4.2 Selection of tools: This is covered in section 11.2.

3–7.4.4.3 Language with check list: The approach taken in this Guide is best illustrated by the example in section 11.2.2.

3–7.4.4.4 Alternative language: It would seem that this clause is needed to cover the use of C, since many constructs in that language can compile but execute unpredictably. It is not possible to "qualify" a C compiler to overcome this problem. Rather it is necessary to use tools to show that the code cannot execute unpredictably — an expensive and time-consuming task, even with tool support. See static analysis, section 12.6

3–7.4.4.5 Coding standards: This issue is handled in several ways in the Guide. There is the question of the *intelligibility* of the code which is handled by code review (see section 12.8). There is the question of ensuring the detection of common errors by using a suitable programming environment (see section 11.2). Lastly, for the higher Indexes (MSI3 and MSI4), strong static analysis techniques should be employed (see section 12.6).

3–7.4.4.6 Source code documentation with check list: This is covered in section 11.2.

3–7.4.5.1 Division of responsibility for detailed design: The user needs to be assured of the claims made by the supplier *and* that those claims are sufficient to ensure the safety of the system. The suppliers' claims can be backed by certification.

3–7.4.5.4 Module design: Module design is not explicitly covered in the Guide. However, a poor design will necessarily make testing and analysis more difficult. In the extreme, static analysis required for MSI3/MSI4 systems (see section 12.6) can be impossible with poor design.

3–7.4.6.1 Source code and checklist: The checklist does not cover assurance of the absence of common errors which this Guide does(see section 11.2).

3–7.4.7.1 Module test: This is called component testing in this Guide, see section 12.13. Unlike the standard, this Guide recommends different levels of coverage according to the MSI, see B.3.12.

3–7.4.7.2 Module test functionality: See section 12.13 and B.3.12. The form of module test specified in *BS 7925* does not cover the requirement in *IEC 61508* to check that a module does not perform unintended functions.

3–7.4.7.3 Module test results: The important aspect of module test results is that of structural coverage which is not covered in *IEC 61508*. See B.3.12.

3–7.4.7.4 Module test corrective action: In this Guide, corrective action is covered by compliance to *ISO 9001*. See section 11.1.

3–7.4.8.1 Integration test specification: Integration is not covered as a separate topic in this Guide, but is covered under sections 12.16 and 12.11.

3–7.4.8.2 Integration test specification checklist: See above.

3–7.4.8.3 Integration test results: See above.

3–7.4.8.4 Integration test documentation: See above.

3–7.4.8.5 Integration test corrective action: In this Guide, corrective action is covered by compliance to *ISO 9001*. See section 11.1.

3–7.5.2.1 Hardware and software integration test specification: In the case of measurement software, the testing of the integrated device with the hardware is undertaken by the measurement of samples of known characteristics. This is a topic of metrology and not covered in the Guide. However, if the software can be shown to be very simple indeed, such end-to-end testing may be sufficient for MSI1 systems, see 10.2.

3–7.5.2.2 Hardware and software integration test check list: See above.

3–7.5.2.3 Hardware and software integration test premises: See above.

3–7.5.2.4 Hardware and software integration test activities: See above.

3–7.5.2.5 Hardware and software integration: See above.

3–7.5.2.6 Hardware and software integration corrective action: In this Guide, corrective action is covered by compliance to *ISO 9001*. See section 11.1.

3–7.5.2.7 Hardware and software integration results: See above.

3–7.5.2.8 Hardware and software integration corrective action: In this Guide, corrective action is covered by compliance to *ISO 9001*. See section 11.1.

3–7.7.2.2 Validation activities: This Guide gives very detailed information concerning validation. For an overview, see figure 6.1.

3–7.7.2.3 Validation results: The documentation itself is not explicitly covered in this Guide, but see section B.2.

3–7.7.2.4 Validation results checklist: The list in the standard should be used to ensure the documentation is complete.

3–7.7.2.5 Validation results corrective action: In this Guide, corrective action is covered by compliance to *ISO 9001*. See section 11.1.

3–7.7.2.6 Validation requirements: The Guide places *more* emphasis on analysis methods at the higher Indexes, MSI3 and MSI4. Testing appears to *require* an environment in which the input signals are simulated, and therefore potentially excludes end-to-end testing (as in section 10.2).

3–7.7.2.7 Software tool qualification: This topic is not currently covered in this Guide.

3–7.7.2.8 Validation result requirements: In the case of certification, the results must be documented in a detailed manner to satisfy an assessment. See also section B.2.

3–7.8.2.1 Software modification procedures: Software modification is not explicitly covered in this Guide. In practice, it is necessary to use impact analysis to reduce the amount of re-validation to do after such a modification.

3–7.8.2.2 Software modification check list: See above.

3–7.8.2.3 Software modification impact: See above.

3–7.8.2.4 Software modification documentation: See above.

3–7.8.2.5 Software modification safety implications: See above.

3–7.8.2.6 Software modification planning: See above.

3–7.8.2.7 Software modification: See above.

3–7.8.2.8 Software modification documentation: See above.

3–7.8.2.9 Revalidation after software modification: See above.

3–7.8.2.10 Assessment of software modification: This Guide is written under the assumption that the validation is performed after any software modification. If this is not the case, then suitable justification needs to be produced.

3–7.9.2.1 Verification planning: In the context of a supplier, verification is the same as validation, since details of the intended use are not relevant.

3–7.9.2.2 Verification planning check list: See above.

3–7.9.2.3 Verification: See above.

3–7.9.2.4 Verification documentation: See above.

3–7.9.2.6 Verification phases: See above.

3–7.9.2.7 Verification activities: This section simply lists other relevant sections in the standard.

3–7.9.2.8 Verification safety requirements: See above.

3–7.9.2.9 Software architecture verification: See above.

3–7.9.2.10 Software design verification: See above.

3–7.9.2.11 Software module verification: See above.

3–7.9.2.12 Software code verification: Code verification is covered in this Guide by code review (12.8) and static analysis (12.6). Note also the importance of defensive programming, see section 12.7.

3–7.9.2.13 Data verification and checklist: Data embedded within an instrument must be verified as part of the code, while dynamic data needs special attention. For instance, some devices may have calibration data whose values are set during a calibration mode. Such a mode needs careful checking if an incorrect calibration by a user could make an application unsafe. See also *EUR 19265*.

B.5 Potential problems related to certification

In the context of certification to *IEC 61508*, a number of problems can arise which are not covered in this Guide. Some of these are as follows:

SOUP. Software Of Uncertain Pedigree may be involved. In general, it does not seem credible to offer certification in that context. However, the HSE Guidance in this area is highly recommended [*SOUP–1, SOUP–2*].

Use of a safety case. This method is not included in *IEC 61508*, although it can be highly advantageous.

Dependence on development. The supplier may wish to claim that the excellence of their development process ensures that the product will satisfy the safety claims made for it. Unfortunately, it is well-known that development is error-prone. One of the authors has heard of a SIL4 development which produced a product that was so buggy as to be unusable! Moreover, the development process is essentially indirect evidence about the product. Hence, validation is crucial, and the supplier should be able to provide evidence of validation that largely satisfies the recommendations stated in this Guide. As noted in figure 6.1, it is crucial that the development process includes the choice of the validation methods so that there is no retrospective element to the validation work.

Validation methods not covered in this Guide. It must be assumed that these methods appear in *IEC 61508*, and hence the section above can be consulted. As a general rule, an alternative method of validation may be used instead of one of the recommended methods, provided that it delivers a comparable level of assurance.

Complexity versus safety integrity. The Guide uses both to determine the MSI. In earlier versions of this Guide that did not handle *IEC 61508*, only complexity and the risk involved in the application were used to determine the MSI. To conform to the safety standard, only safety integrity is used to determine the SIL in the context of *IEC 61508*. Thus the SIL may be different from the MSI.

This is not to say that complexity is not a problem, since the more complex the software is, the more expensive it will be to apply the appropriate validation techniques.

Appendix C

Some Validation Examples

In this appendix, a number of examples are given of the approaches used in real measurement systems for demonstrating high quality for the software component. Each example references the main text and has been developed by using an earlier version of this Guide.

C.1 Measurement of flatness/length by a gauge block interferometer

NPL is the supplier of both the hardware and the software of this equipment.

The physics behind the measurement system is categorised as **validated physics**, see section 3.3.

The model of a measurement system (on section 3.1), applies to this measurement system.

The criticality of usage is **critical**, see section 4.1.1. If any fault were found, NPL would be very embarrassed. On the other hand, no life-critical application is known, nor seems likely.

The measurement system does not have to satisfy any specific legal requirement (see section 4.1.2).

The complexity of the control is rated as **very simple** or **simple**, see section 4.1.3.

The complexity of the data processing is rated as **modest** or **complex**, mainly because of the size (15-20,000 lines of C++), see section 4.1.4.

The answers to the six key questions (see section 5.1) were:

1. It is known that end-to-end testing cannot provide sufficient confidence in the software, since it would be hard to locate any numerical instabilities in the processing by that means.

2. Raw data can be extracted and used for testing and is probably adequate for the length measurement, but not for flatness. However, many artefacts do not provide demanding cases for the software.

3. The software had been tested against an older system by back-to-back testing.

4. All errors and comments are logged, but since the equipment is only in trial use at the moment, error statistics are not very meaningful.

5. Any error in the control software cannot result in a false measurement.

6. The operator interface is not thought to be complex nor likely to put any measurement at risk.

Issues arising from the software risk assessment questions (see section 10.1) were as follows:

- There is no record of a measurement system malfunction due to software (as opposed to errors in the software detected during development).
- The NPL software development process is internally audited and subject to an external audit by LRQA for *ISO 9001*.
- The measurement system does not require regulatory approval.
- The measurement system makes obvious internal checks and provides modest protection against operator error.
- The length computations are linear. The flatness calculations are non-linear, but have been checked by the NPL mathematics group for stability. The algorithm used for flatness is given in *ReferenceSoftware* and *ChebychevGeometric*.
- An internal numerical error (overflow, etc) should be detected by the software.

As a result of the above, the software was rated as MSI2.[1]
Issues arising from the general software questions (see section 11.4.1) were as follows:

- Since the measurement system has not yet been released to end customers, the provision of information (by NPL) has not yet arisen.
- No software reliability data is available, as the log of errors is an empty file at this point.
- The question of operator training and assessment of the user manual has not yet been addressed.
- The main aspect of the software which gives rise to concern is that of its size.

Issues arising from the MSI0 questions (see section 11.4.2) were as follows:

- No issues as NPL operates an *ISO 9001* process.

Issues arising from the MSI1 questions (see section 11.4.3) were as follows:

- Software inspection is not undertaken, but code review is being done. The two people on the project review each other's code.
- There is a mathematical specification of the processing algorithms, and the code for these algorithms has been derived from the specification.
- During the development, the main processing code was tested (informally) to branch level.

Issues arising from the MSI2 questions (see section 11.4.4) were as follows:

- Regression testing has been applied from the start, and the results have been recorded.

[1] This determination of the MSI pre-dates the table for calculating the MSI.

- The variant of stress-testing for image processing has been used. This testing revealed a weakness in the specification.
- All known (and documented) errors are trivial.
- The system is too young to suffer yet from structural decay.

Issues arising from the MSI3 questions (see section 11.4.5), although not necessary for MSI2 software, were as follows:

- As noted above, informal branch testing was undertaken on the main algorithms. This should imply 100% statement coverage.
- No Formal Specification has been produced.
- Static analysis has not been undertaken.
- It is not thought that there is a role for accredited testing of the software.
- An attempt was made to detect memory leaks in the software.

The Software Engineering techniques listed in this Guide were applied as follows:

Independent audit: (see section 12.1.) (NPL audit, and external *ISO 9001* audit.)

Review of informal specification: (see section 12.2.) Yes.

Software inspection of specification: (see section 12.3.) No, but code review undertaken.

Mathematical specification: (see section 12.4.) Yes.

Formal Specification: (see section 12.5.) No.

Static analysis: (see section 12.6.) No.

Code review: (see section 12.8.) Yes.

Numerical stability: (see section 12.9.) Yes.

Component testing: (see section 12.13.) Yes, on processing algorithms.

Accredited testing: (see section 12.15.) Not applicable.

System-level testing: (see section 12.16.) Yes.

Stress testing: (see section 12.17.) Yes, for image processing part.

Numerical reference results: (see section 12.18.) Yes, for flatness algorithms.

Back-to-back testing: (see section 12.19.) Yes, with older software.

Regression testing: (see section 12.14.) Yes.

Conclusions

The validation of this software is generally consistent with the MSI of 2. One of the difficulties facing the development is that only two people were involved, making a comprehensive independent software inspection effectively impossible. Code review of each other's code was undertaken as a result of an internal NPL audit. The size of the code made this a large task.

C.2 Electrical measurement resistance bridge

The device with its software provides the primary service for NPL to calibrate resistance measuring devices. The measurement system is entirely for in-house use by NPL, and hence NPL is the supplier and the sole user.

Oxford Instruments are licensed by NPL to manufacture a similar device, but this device has different software and is not considered here.

The physics behind the measurement system is categorised as **validated physics**, see section 3.3.

The model of a measurement system (on section 3.1), applies to this measurement system.

The criticality of usage is **critical**, see section 4.1.1. If any fault was found, NPL would be very embarrassed, especially with the international intercomparisons.

The measurement system does not have to satisfy any specific legal requirement (see section 4.1.2).

The complexity of the control is rated as **complex**, see section 4.1.3. The control consists of 10,000 lines of Turbo-Pascal which uses its own macro language to aid the control.

The complexity of the data processing is rated as **simple**, see section 4.1.4. Only two stored calibration constants are used in a linear calculation. The data processing part is a separate program from the control logic, communicating via a file.

The answers to the six key questions (see section 5.1) were:

1. Very good confidence in the measurement system can be established by end-to-end testing. Standard artefacts can be used for this purpose and the number of individual tests that need to be undertaken is widely recognised.

2. Since the control and processing programs are separate, they can be validated independently.

3. The software has been in use for 10 years in various versions.

4. No errors have been recorded which gave an incorrect measurement. The main problems have been in the operator interface to the control program.

5. It seems hard for the control software to produce an undetected error in a measurement. Only a few lines copy the basic data into a file for processing by the data processing program.

6. Only qualified staff at NPL can use the measurement system for calibration work.

Issues arising from the software risk assessment questions (see section 10.1) were as follows:

- As noted above, the control software has essentially no impact on the measurements themselves.

- The data processing software in classified as **simple** and hence straightforward to validate.

- Since the measurement system is only used by NPL staff, the control software does not need to handle many cases of misuse. (The Oxford Instruments version is quite different in providing simpler control with fewer user options.)

As a result of the above, the software was rated as MSI1.[2]
Issues arising from the general software questions (see section 11.4.1) were as follows:

- With the supplier being the sole user, the provision of information is not a problem.
- The raw data is in a file used by the data processing program.
- An independent check has been made on the data processing program by comparing it with the similar program developed for the Oxford Instruments device.
- Completeness of the system testing has not been measured by, for instance, test coverage. This is not thought to be a problem due to the unlikelihood of the control software giving a problem.

Issues arising from the MSI0 questions (see section 11.4.2) were as follows:

- Since the basic design of the software is ten years old, much of the design is actually embedded in the code itself, rather than in a separate document. However, the complex design aspects are in the control software which do not impact the basic measurements.
- The only recent change to the computing environment has been the compiler, which caused the system to be re-tested.
- The system is maintained by a physicist rather than a programmer. This is adequate in this instance, since the system is assessed at a MSI of 1. If a higher Index was required, additional skills in software engineering would be needed.

Issues arising from the MSI1 questions (see section 11.4.3) were as follows:

- It was noted that a code review should have been undertaken on the software.
- The system testing applied to the control software did not provide any measurement of test coverage, such as statement coverage.

Issues arising from the MSI2 questions (see section 11.4.4), although not necessary for MSI1 software, were as follows:

- Only internal audits have been undertaken.
- It is thought that structural decay is a minor problem due to the age of the software.

The Software Engineering techniques listed in this Guide were applied as follows:

Independent audit: (see section 12.1.) (NPL audit, and external *ISO 9001* audit.)

Review of informal specification: (see section 12.2.) No — specification not separately documented.

Software inspection of software: (see section 12.3.) No.

Mathematical specification: (see section 12.4.) Yes.

Formal Specification: (see section 12.5.) No.

Static analysis: (see section 12.6.) No.

[2] Again, this determination of the MSI pre-dates the table for calculating the MSI.

Numerical stability: (see section 12.9.) Yes.

Component testing: (see section 12.13.) No.

Accredited testing: (see section 12.15.) Not applicable.

System-level testing: (see section 12.16.) Yes, and gives required confidence in the measurement system.

Stress testing: (see section 12.17.) No.

Numerical reference results: (see section 12.18.) No.

Back-to-back testing: (see section 12.19.) Yes, NPL data processing software against the similar software for the Oxford Instruments device.

Regression testing: (see section 12.14.) Yes, for data processing program.

Conclusions

The software risk analysis shows that this software is of low risk but (the impact of) the complexity of control is complex and hence a MSI of 1 is appropriate, even though a measurement error would be an embarrassment to NPL. The key aspect is that testing the measurement system as a black-box gives a high assurance of the system, including the software relevant to the measurement.

The validation of the software undertaken by NPL is generally consistent with the MSI of 0, rather than MSI1. Further validation is therefore to be recommended.

C.3 Mean renal transit time

This medical application consists of a gamma camera and associated software for measurement of mean renal transit time. The diagnostic imaging procedure in question is called a *renogram*, and the medical speciality of which it is a part is called *nuclear medicine*.

The patient is injected (in an arm vein) with an appropriate radiopharmaceutical, whose passage through the body can be observed by the use of a large field of view (approx 50×40cm) radiation detector called a gamma camera. In this application, the detector is positioned under the patient (lying supine on a low attenuation imaging couch), covering the posterior aspect of the lower chest and abdomen, centred on the kidneys. Immediately after injection, a sequence of digital images (10 second frames) is recorded for a period of 20 minutes, showing the passage of the injected material into, and out of, the organs within the field of view (heart, spleen, liver, kidneys and bladder). In this case, the organs under scrutiny are the kidneys.

The first stage of the image processing involves the creation of so called regions-of-interest (ROI) around particular organs and structures (both kidneys, heart, and two "background" areas). This is usually done manually, using a mouse/cursor. In image processing parlance this would be called "segmentation". Each ROI is then applied, in turn, to each of the 120 frames, and the "information density" (in this case count rate) within each ROI plotted as a function of time. Five curves are thus produced. These curves are then analysed further to calculate the mean renal transit time for each kidney. The use of this model/algorithm/software simulates the effect of direct injection into the renal artery, which is extremely invasive and requires an anaesthetic. Intravenous injection, by contrast, is minimally invasive and is happily tolerated by the vast majority of patients. The passage

of a non-reabsorbable solute (of which the radiopharmaceutical injected is an example) through the kidney is slowed in various renal disorders, and the mean renal transit time (of that solute) is thus prolonged. The measurement of mean renal transit time can be used to diagnose kidney disorders such as upper urinary tract obstruction, which is usually caused by narrowing of the junction between kidney and ureter.

The information below has been provided by Mr P.S.Cosgriff who is a user of the system, but also has a detailed knowledge of its workings. He has written some software that can perform part of the analysis, although most of the system being considered is supplied by other (commercial) parties. The user-written software was produced according to general principles described in a recent guide published by the Institute of Physics and Engineering in Medicine [*Cosgriff*]. The gamma camera was supplied by one company, and the image processing software by another.

The physics behind the instrument is categorised as **Physics can be modelled**, see section 3.3.

The model of an instrument (on section 3.1), applies to this instrument, but note the degree of interaction as noted above.

The criticality of usage is **safety-indirect**, see section 4.1.1. The patient's doctor (hospital consultant) makes the final decision regarding the use of the results in deciding treatment.

The instrument has to satisfy the specific legal requirements (see section 4.1.2) in Medical Device Regulations 1996 (as a Class 2A product).

The complexity of the control is rated as **Complex**, see section 4.1.3.

The complexity of the data processing is rated as **Complex**, see section 4.1.4.

The answers to the six key questions (see section 5.1) were:

1. Due to the complexity of both the control and data processing software, only limited assurance can be obtained from end-to-end testing. It is also difficult to produce appropriate models (hardware "phantoms") for testing due to the difficulty in simulating a complex biological system.

2. The data from the camera is essentially separate and is typically available in a format that can be processed by any nuclear medical image processing. In consequence, the gamma camera and the software can be considered separately.

   Naturally, the camera can be considered as an instrument it its own right, but this is not considered further here.

3. Similar instruments are available for which some reliability data is available.

4. An error log is available from the bug-fix information sent out by the software supplier.

5. False measurements are certainly possible. For instance, if the patient moves and the operator fails to take this into account, the image processing software could fail to locate the kidney correctly.

6. The operator interface is complex and only suitably trained staff can operate the equipment.

Issues arising from the software risk assessment questions (see section 10.1) were as follows:

- At worst, an instrument malfunction could cause inappropriate treatment to the patient.

- The gamma camera performs some built-in tests, but the software does not.
- The control functions cannot be effectively protected against operator error. (Hence the need for skilled operators.)
- The computation is complex and non-linear. The final calculation after the image processing uses differences and requires that the initial data points are heavily smoothed to avoid numerical instabilities.
- It is unclear how robust the software is with regard to issues like numerical overflow.

For this example, the Measurement Software is taken as MSI3; MSI2 may be more realistic from an economic point of view. A drug infusion pump connected to a patient on an Intensive Care Unit would certainly be SIL4 with *IEC 61508*, but the software contained would be probably 100 times less complex than that described above and would also probably have hardware interlocks to prevent catastrophic failure. On balance, therefore, MSI3 is thus considered reasonable for the measurement of mean renal transit time.

Issues arising from the general software questions (see section 11.4.1) were as follows:

- An independently specified file format is available which allows a user or manufacturer to test the software without the use of the gamma camera.

The questions concerning software development usually be answered, since the information available is from a user.

Issues arising from the MSI 0 questions (see section 11.4.2) were as follows:

- There is an error log and evidence of clearance of errors.
- It is unclear how the exact version of the software is identified (which is non-trivial due to updates being applied by users). The Notifying Body oversight should check for this.

There was no information available to answer any of the questions for MSI1–3.

The only information available about the Software Engineering techniques applied is as follows (little information was available at the time):

Accredited testing: (see section 12.15.) Accredited testing to be undertaken under *COST B2*.

Independent audit: (see section 12.1.) As part of type approval.

Conclusions

The software risk analysis shows that there is a moderate risk of inappropriate treatment should the complex software produce an incorrect result. The ability to test the software by means of "software phantoms" has led to a proposal to develop a formal certification process for this type of medical software [*COST B2*]. The information to hand about the validation so far completed is inadequate to make any positive conclusion about the validation of the software.

Bibliography

Note This bibliography has the pages on which an item is cited at the end of each entry. Hence it can be used as an additional index to this Guide.

AOAC S.L.R. Ellison, M.G. Cox, A.B. Forbes, B.P. Butler, S.A. Hannaby, P.M. Harris and Susan M. Hodson. Development of data sets for the validation of analytical instrumentation. J. AOAC International, 77:777–781, 1994. 77

ATS-Language–1 Requirements for the Selection and Use of Programming Languages for Safety Related ATS Systems. Part 1: Preliminary material. February 1998. 91, 102, 105

ATS-Language–2 Requirements for the Selection and Use of Programming Languages for Safety Related ATS Systems. Part 2. February 1998. 91, 102, 105

AttitudeMonitor A Coombes, L Barroca, J S Fitzgerald, J A McDermid, L Spencer and A Saed. Formal Specification of an Aerospace system: the Attitude Monitor. Chapter 13, Application of Formal Methods. M G Hinchey and J P Bowen. Prentice-Hall 1995. 61

BPG-LIMS S M Lower. Software Support for Metrology Best Practice Guide No. 9: Selection and use of LIMS. March 2001. `http://resource.npl.co.uk/docs/science_technology/scientific_computing/ssfm/documents/ssfmbpg9.pdf` 15

BPG-Model R M Barker, M G Cox, A B Forbes and P M Harris. Software Support for Metrology Best Practice Guide No. 4: Discrete modelling and experimental data analysis. March 2004. `http://resource.npl.co.uk/docs/science_technology/scientific_computing/ssfm/documents/ssfmbpg4.pdf` 13

BPG-Software T Clements, L Emmet, P Froome, S Guerra. Software Support for Metrology Best Practice Guide No. 12: Software Development, Virtual Instruments and Mixed Languages. September 2002. `http://resource.npl.co.uk/docs/science_technology/scientific_computing/ssfm/documents/ssfmbpg12.pdf` 88, 98

BPG-Uncertainties M G Cox and P M Harris. Software Support for Metrology Best Practice Guide No. 6: Uncertainty evaluation. March 2004. `http://publications.npl.co.uk/npl_web/pdf/dem_es11.pdf` 12

BS 7925 British Computer Society Specialist Group in Software Testing. Standard for Software Component Testing (Working Draft 3.0). Glossary of terms used in software testing (Working Draft 6.0). October 1995. Now revised as a BSI standard, BS7925 part 1 and 2 available from BSI. 69, 70, 95, 111

CASS-subsystems Barrie Reynolds. CASS boundary templates for subsystems. CASS Report No CASS2-BR-COMP-D4. 5th April 2001. 30

ChebychevGeometric R Drieschner. Chebyshev approximation to data by geometric elements. Numerical algorithms. Edited by C Brezinski. Volume 5. pp509–522. J C Baltzer. 1993. 116

Chemometrics R. G. Brereton. Chemometrics: Applications of Mathematics and Statistics to Laboratory Systems. Ellis Horwood, Chichester, England, 1990. 77

CISE 25/99 H.R. Cook, M.G. Cox, M.P. Dainton and P.M. Harris. A methodology for testing spreadsheets and other packages used in metrology. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme. NPL Report CISE 25/99, September 1999. `http://publications.npl.co.uk/npl_web/pdf/cise25.pdf` 7, 12, 77, 78

CISE 26/99 H.R. Cook, M.G. Cox, M.P. Dainton and P.M. Harris. Testing spreadsheets and other packages used in metrology: A case study. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme. NPL Report CISE 26/99, September 1999. `http://publications.npl.co.uk/npl_web/pdf/cise26.pdf` 77

CISE 27/99 H.R. Cook, M.G. Cox, M.P. Dainton and P.M. Harris. Testing spreadsheets and other packages used in metrology: Testing the intrinsic functions of Excel. Report to the National Measurement System Policy Unit, Department of Trade and Industry, from the UK Software Support for Metrology Programme. NPL Report CISE 27/99, September 1999.
`http://publications.npl.co.uk/npl_web/pdf/cise27.pdf` 77

CMSC 29/03 Trevor Esward, Gabriel Lord, and Louise Wright. Model Validation in Continuous Modelling. NPL Report CMSC 29/03, September 2003.
`http://publications.npl.co.uk/npl_web/pdf/cmsc29.pdf` 13

CompetencyStandards Competency Standards for Safety-Related System Practitioners, IEE/BCS, 1999. `http://www.iee.org.uk` 48

COST B2 COST B2: Final Report (EUR 17916EN): Quality assurance of nuclear medicine software, Cosgriff PS, ed. Luxembourg: Office of Official Publications of the European Communities, 1997. Downloadable from website:
`http://www.nuclearmedicine.org.uk` 73, 122

Cosgriff Houston A, Cosgriff PS. Quality assurance of in-house developed software in nuclear medicine. In: Quality Control of Gamma Camera Systems. IPEM Report #86, IPEM Publications, York, Y024 1ES, 2003. 121

DASL The National Physical Laboratory's Data Approximation Subroutine Library. G T Anthony and M G Cox. In J C Mason and M G Cox, editors, Algorithms for Approximation, pages 669–687. Clarendon Press, Oxford, UK. 1987. 66

DO-178B Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B). December 1992. 10, 47, 68, 69, 71, 82

Dyson N. Dyson. Chromatographic Integration Methods. Royal Society of Chemistry, 1990. 77

EUR 19265 Nuclear Safety and the Environment: Common position of the European nuclear regulators for the licensing of safety critical software for nuclear reactors. European Commission, November 2000. 30, 42, 48, 62, 108, 110, 113

FDA ODE Guidance for the Content of Premarket Submission for Medical Devices Containing Software. Draft, 3rd September 1996. 8

FloatingPointValidation J Du Croz and M Pont. The Development of a Floating Point Validation Package. NAG Newsletter No 3, 1984. 56, 67

FormalMethods B A Wichmann. A personal view of Formal Methods. National Physical Laboratory. March 2000. Available free on the internet at `http://resource.npl.co.uk/docs/science_technology/scientific_computing/ssfm/documents/baw_fm.pdf` 61

FunctionalSafety D J Smith and K G L Simpson, Functional Safety — A Straightforward Guide to IEC 61508 and Related Standards. Butterworth Heinemann. ISBN 0–7506–5270–5. 2001. 29, 41

GAMP Supplier Guide for Validation of Automated Systems in Pharmaceutical Manufacture. ISPE 3816 W. Linebaugh Avenue, Suite 412, Tampa, Florida 33624, USA. 35

Gill P.E. Gill, W. Murray and M.H. Wright. Practical Optimization. Academic Press, 1981. 77

Golub G.H. Golub and C.F. Van Loan. Matrix Computations. John Hopkins University Press, Baltimore, MD, USA, 1996. Third edition. 77

GradedReferenceDataSets M.G. Cox. Graded reference data sets and performance profiles for testing software used in metrology. In P. Ciarlini, M.G. Cox, F. Pavese, and D. Richter, editors, Advanced Mathematical Tools in Metrology III, pages 43–55, Singapore, 1997. World Scientific. 77

HF-Guide Safety-related systems — Guidance for engineers. Hazards Forum. March 1995. ISBN 0 9525103 0 8. 4

Higham N. J. Higham. Accuracy and Stability of Numerical Algorithms. SIAM. 1996. 66

HSC The use of computers in safety-critical applications. Final report of the study group on the safety of operational computer systems. HSC. London. ISBN 0–7176–1620–7. 1998. 9

IEC 601 IEC 601–1–4: 1996. Medical electrical equipment — Part 1: General requirements for safety 4: Collateral Standard: Programmable electrical medical systems. 8

IEC 61508 IEC 61508: Parts 1–7, Functional safety of electrical/electronic/ programmable electronic (E/E/PE) safety-related systems. *See following.* 3, 4, 9, 10, 28, 29, 37, 41, 47, 51, 57, 58, 61, 75, 98, 122

IEC 61508–1 IEC 61508–1:1998, Functional safety of E/E/PE safety-related systems — Part 1: General requirements. Corrigendum No 1 (April 1999). *This version has "BSI 08–1999" on each page.* 88, 89

IEC 61508–2 IEC 61508–2:2000, Functional safety of E/E/PE safety-related systems — Part 2: Requirements for E/E/PE safety-related systems.

IEC 61508–3 IEC 61508–3:1998, Functional safety of E/E/PE safety-related systems — Part 3: Software requirements. Corrigendum No 1 (April 1999). *This version has "BSI 08–1999" on each page.* 9, 29, 32, 57

IEC 61508–4 IEC 61508–4:1998, Functional safety of E/E/PE safety-related systems — Part 4: Definitions and abbreviations. Corrigendum No 1 (April 1999). *This version has "BSI 08–1999" on each page.*

IEC 61508–5 IEC 61508–5:1998, Functional safety of E/E/PE safety-related systems — Part 5: Examples of methods for the determination of safety integrity levels.

IEC 61508–6 IEC 61508–6:2000, Functional safety of E/E/PE safety-related systems — Part 6: Guidelines on the application of IEC 61508–2 and IEC 61508–3.

IEC 61508–7 IEC 61508–7:2000, Functional safety of E/E/PE safety-related systems — Part 7: Overview of techniques and measures. 90, 98, 103

IEE-PLC SEMSPLC Guidelines: Safety-Related Application Software for Programmable Logic Controllers. IEE Technical Guidelines 8: 1996. ISBN 0 85296 887 6. 41

IGE-SR-15 Programmable Equipment in Safety Related Applications. Institute of Gas Engineers. IGE/SR/15 Edition 3, with approved amendments, July 2000. 29, 41

ISO 9000–3 ISO/IEC 9000–3: 1997. Quality management and quality assurance standards — Part 3: Guidelines for the application of ISO 9001:1994 to the development, supply and maintenance of software. 8, 35

ISO 9001 EN ISO 9001:2000, Quality management systems — Requirements. 8, 26, 35, 57, 58

ISO 10360–6 BS EN ISO 10360–6:2001. Geometrical product specifications (GPS) — acceptance test and reverification test for coordinate measuring machines (CMM). Part 6: Estimation of errors in computing Gaussian associated features. 35, 66, 76

ISO 12119 ISO/IEC 12119: 1994, Information technology — Software packages — Quality requirements and testing. 74

ISO 12207 ISO/IEC 12207: 1995. Information technology — Software life cycle processes. 36

ISO 13233 ISO/IEC TR 13233: 1995. Information Technology — Interpretation of Accreditation Requirements in ISO/IEC Guide 25 — Accreditation of Information Technology and Telecommunications Testing Laboratories for Software and Protocol Testing Services. 8, 35

ISO 15942 ISO/IEC TR 15942: 2000, Guide for the use of the Ada programming language in high integrity systems. 50, 104, 105

ISO/IEC 17025 ISO/IEC 17025: 1999. General requirements for the competence of testing and calibration laboratories. 7, 57, 58

LIMS R Fink, S Oppert, P Collison, G Cooke, S Dhanjal, H Lesan and R Shaw. Data Measurement in Clinical Laboratory Information Systems. Directions in Safety-Critical Systems. Edited by F Redhill and T Anderson, Springer-Verlag. pp 84–95. ISBN 3–540–19817–2 1993. 15

LINPACK LINPACK User's Guide. J J Dongarra, C B Moler, J R Bunch, and G W Stewart. SIAM, Philadelphia. 1979. 66

Littlewood B Littlewood and L Strigini. The Risks of Software. Scientific American. November 1992. 22, 47, 69

MCDC A Dupuy and N Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE–2 satellite software. Digital Aviation Systems Conference, October 2000. 71

Microprocessor B A Wichmann. Microprocessor design faults. Microprocessors and Microsystems, Vol 17, No 7, pp399–401. 1993 67

MISRA Development Guidelines For Vehicle Based Software. The Motor Industry Software Reliability Association. MIRA. November 1994. ISBN 0 9524156 0 7. 9

MISRA-C MISRA-C:2004 Guidelines for the use of the C language in critical systems. The Motor Software Reliability Association. October 2004. (Enquiries to misra@mira.co.uk) 104

NAG The NAG Fortran Library. The Numerical Algorithms Group Ltd., Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, UK. 66

NIS40 Storage, Transportation and Maintenance of Magnetic Media. NIS40. NAMAS, October 1990.

PROOF Steve King, Jonathan Prior, Rod Chapman and Andy Pryor. Is Proof More Cost-Effective Than Testing? IEEE Trans. Soft. Eng. Vol 26, No 8. pp676–685. 2000. 50

ReferenceDataSets B.P. Butler, M.G. Cox, S.L.R. Ellison, and W.A Hardcastle, editors, Statistics Software Qualification: Reference Data Sets. Royal Society of Chemistry, Cambridge, ISBN 0–85404–422–1. 1996. 76, 77

ReferenceDataSetsDesign M.G. Cox and P.M. Harris. Design and use of reference data sets for testing scientific software. Analytica Chimica Acta 380, pp 339 - 351, 1999. 77

ReferenceSoftware G T Anthony, H M Anthony, B P Butler, M G Cox, R Drieschner, R Elligsen, A B Forbes, H Gross, S A Hannaby, P M Harris and J Kok. Reference software for finding Chebyshev best-fit geometric elements. Precision Engineering. Vol 19, pp28–36. 1996. 116

ReliabilityBounds Peter G Bishop. Rescaling Reliability Bounds for a New Operational Profile. ISSTA 2002. Rome. Italy. ACM. 22

RiskIssues Guidelines on Risk Issues. The Engineering Council. February 1993. ISBN 0–9516611–7–5. 19

SERB J A McDermid (Editor). Software Engineer's Reference Book. Butterworth-Heinemann. Oxford. ISBN 0 750 961040 9. 1991. 57

SIZE1 Pavey, D.J. and L.A. Winsborrow, Demonstrating Equivalence of Source Code and PROM Contents. The Computer Journal, 1993. 36(7). 80

SIZE2 Pavey, D.J. and L.A. Winsborrow, Formal demonstration of equivalence of source code and PROM contents: an industrial example, in Mathematics of Dependable Systems, C. Mitchell and V. Stavridou, Editors. 1995, Clarendon Press, Oxford. 80

SmartInstrument A Dobbing, N Clark, D Godfrey, P M Harris, G Parkin, M J Stevens and B A Wichmann, Reliability of Smart Instrumentation. National Physical Laboratory and Druck Ltd. August 1998. Available free on the internet at `http://resource.npl.co.uk/docs/science_technology/scientific_computing/ssfm/documents/overall2.pdf` 8, 9, 17, 60, 62, 71, 79

SoftwareInspection T Gilb and D Graham. Software Inspection. Addison-Wesley 1993. ISBN 0–201–63181–4. 59

SoftwareTesting B.P. Butler, M.G. Cox, A.B. Forbes, S.A. Hannaby, and P.M. Harris. A methodology for testing classes of approximation and optimisation software. In R. F. Boisvert, editor, The Quality of Numerical Software: Assessment and Enhancement, pages 138–151, London, 1997. Chapman and Hall. 77

SOUP–1 P G Bishop, R E Bloomfield and P K D Froome, Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP). HSE Books, ISBN 0 7176 2011 5. 2001. 29, 56, 99, 100, 103, 105, 110, 113

SOUP–2 P G Bishop, R E Bloomfield and P K D Froome, Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications. HSE Books, ISBN 0 7176 2010 7. 2001. 29, 56, 99, 100, 103, 105, 110, 113

Spivey-Z Spivey J M. Understanding Z, CUP 1988. 61

SSfM–2 D Rayner. National Measurement System Software Support for Metrology Programme, 2001–2004. `http://www.npl.co.uk/ssfm` 11, 13, 60

SSfM–3 D Rayner. National Measurement System Software Support for Metrology Programme, 2004–2007. `http://www.npl.co.uk/ssfm` 10

SSfM-Model National Measurement System Software Support of Metrology Programme, Theme 1: Modelling Techniques. `http://www.npl.co.uk/server.php?show=nav.1204` 11, 13, 60

StaticAnalysis B A Wichmann, A A Canning, D L Clutterbuck, L A Winsborrow, N J Ward and D W R Marsh. An Industrial Perspective on Static Analysis. Software Engineering Journal. March 1995, pp69–75. 62

StressTest B A Wichmann. Some Remarks about Random Testing. National Physical Laboratory. May 1998. Available free on the internet at `http://resource.npl.co.uk/docs/science_technology/scientific_computing/ssfm/documents/stress.pdf` 75, 76

UKAS-61508 Study to investigate the feasibility of developing an **accreditable** product-based scheme of conformity assessment and certification based on satisfying the requirements of International Standard IEC 1508. UKAS. 1997. Available free on the internet at `http://www.npl.co.uk/npl/cise/UKAS/` 9

UKAS-TickIT Use of TickIT as a benchmark for software. UKAS. CIS1 November 1999. Available free on the internet at `http://www.ukas.com/pdfs/TickIT.pdf` 36

VDM-SL J Dawes. The VDM-SL Reference Guide. Pitman Publishing. 1991. ISBN 0–273–03151–1 61

WELMEC–2.3 Guide for Examining Software (Non-automatic Weighing Instruments). WELMEC 2.3, January 1995. Available free on the internet at `http://www.welmec.org/publications/2-3.asp` 8, 20

WELMEC–7.1 Software Requirements on the Basis of the Measuring Instruments Directive. WELMEC 7.1, January 2000. Available free on the internet at `http://www.welmec.org/publications/7-1.asp` 8, 20, 103

Year2000 Test Report: Annodeus Ltd, Diagnostic and Fix programs. National Physical Laboratory. May 1st 1998. 74

Index

Note

The bibliography shows where each reference is used and hence can aid locating specific issues in addition to this index.