*Measurement Good Practice Guide*

No. 5

# Software in Scientific Instruments

*B.A. Wichmann*

NPL National Physical Laboratory

*The National Physical Laboratory is operated on behalf of the DTI by NPL Management Limited, a wholly owned subsidiary of Serco Group plc*

# Measurement Good Practice Guide No 5
# Software in Scientific Instruments

Brian A Wichmann

Information Systems Engineering
National Physical Laboratory

**Abstract:** Software is playing an increasingly important part in scientific instruments and hence there is a need to ensure that the quality of the software is appropriate for the instrument and its application. This Guide provides a means for suppliers and users to assure themselves of the quality of the software.

ISSN 1368–6550

Revised, January 1999

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



This guide was produced under the *Competing Precisely* programme, which is managed on behalf of the DTI by the National Physical Laboratory. NPL is the UK's centre for measurement standards, science and technology.
The objectives of *Competing Precisely* are:

- to demonstrate the contribution accurate measurement can make towards improving industrial competitiveness and compliance with standards and regulations
- to support the development and application of good measurement practice by providing a practical 'toolkit' that companies can use to improve measurement quality
- to back this up with access to local and national sources of measurement expertise and services.

For more information on *Competing Precisely*, contact Paula Knee on 020 8943 6329 (e-mail: paula.knee@npl.co.uk) or by post to National Physical Laboratory, Teddington, Middlesex, TW11 0LW.

# Contents

# 1 Executive summary

Almost all of the current generation of scientific instruments 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 an instrument, how can I be assured that the software is of a sufficient standard to justify its use?
- As a developer of such software, what development techniques should I use, and how can I assure my customers of the quality of the resulting software?

This Good Practice Guide addresses these two questions. The intended readership are those responsible for software in scientific instruments and those using such software. 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. We also exclude the more specialised area of Programmable Logic Controllers, since another guide is available covering this [14].

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

The general approach is a three stage process as follows:

1. A risk assessment based upon a model of an instrument with its software.
2. An assessment of integrity required on the software, based upon the risk assessment (called the **Software Integrity Level**).
3. Guidance on the software engineering methods to be employed determined by the **Software Integrity Level**.

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 as an integral part of the software development process.

To aid in the application of this Guide, some check lists are provided.

To avoid complexity in the wording of this Guide, it is assumed that 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.

No consideration has been given here of the possibility of standardising this material, or obtaining some formal status for it.

# 2 Introduction

The use of software either within or in conjunction with a scientific instrument can provide additional functions in a very cost-effective manner. Moreover, some instruments 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 standard.

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

An area in which there has been a substantial effort to remove software errors is in safety applications, and hence the methods used and specified in safety-critical systems are used here as an indication of what might be achievable, typically at a significant cost. For general advice in this area, see [11].

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 about one man-day per machine instruction — obviously too demanding for almost all software within scientific instruments. Hence the main objective behind this Guide is to strike a balance between development cost and the proven quality of the software.

The main approach taken here is one of *risk assessment* as a means of determining the most appropriate level of rigour (and cost) that should be applied in a specific context.

A major problem to be faced with software is that the failure modes are quite different than with 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** on page 21.

## 2.1 Requirements

There are a number of standards which specify requirements for software in scientific instruments which are collected together here with an indication of the issues to be covered by this Guide. The more relevant standards appear first.

**ISO/IEC Guide 25.** This is the ISO equivalent [16] of the M10 (see below). Paragraph 10.6 states: *Calculations and data transfers shall be subject to appropriate checks.* Paragraph 10.7 states: *Where computers or automated equipment are used for the capture, processing, manipulation, recording, reporting, storage or retrieval of calibration or test data, the laboratory shall ensure that:*

1. *the requirements of this Guide are complied with;*
2. *computer software is documented and adequate for use;*

3. *procedures are established and implemented for protecting the integrity of data; such procedures shall include, but not be limited to, integrity of data entry or capture, data storage, data transmission and data processing;*
4. *computer and automated equipment is maintained to ensure proper functioning and provided with the environmental and operating conditions necessary to maintain the integrity of calibration and test data;*
5. *it[the laboratory] establishes and implements appropriate procedures for the maintenance of security of data including the prevention of unauthorized access to, and the unauthorized amendment of, computer records.*

This Guide gives the most detailed indication of the requirements, and hence is the most useful for both development and assessment.

**EN45001.** This standard [5] is the European equivalent to the previous standard. Section 5.4.1 states that: *All calculation and data transfers shall be subject to appropriate checks. Where results are derived by electronic data processing techniques, the reliability and stability of the system shall be such that the accuracy of the results is not affected. The system shall be able to detect malfunctions during programme execution and take appropriate action.*

The standard gives a different gloss on the same area. Here, (numerical) stability and reliability are mentioned, but security and integrity are not. Again, following this Guide should ensure compliance with this standard.

**M10.** This document is the UKAS laboratory accreditation standard [26]. Section 8.6 states: *The Laboratory shall establish procedures when using computer data processing to ensure that the collection, entry, processing, storage, or transmission of calibration or test data is in accordance with the requirements of this Standard.* Section 8.7 states: *Calculations and data transfers shall be subject to appropriate checks.*

In practice, these requirements are interpreted by the UKAS Assessor. It is hoped that this Guide will aid this interpretation and reduce the risk of the Assessor taking a different view from the Laboratory.

**Weighing machines.** The WELMEC document summarises the position for such machines [32]. 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 part of these requirements are relevant to the software of instruments 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 instruments 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 [17], in any case, but there is no requirement that ISO 9001 is applied to such machines).

**IEC 601-1-4.** The standard covers the software in medical devices [13] and is used both in Europe to support a Directive and by the FDA in the USA [10]. The standard is based upon risk assessment with the software engineering based upon ISO 9000-3 [18]. The flavour of the standard can be judged from a few key extracts below, 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 which are on-line to the patient as opposed used to analyse specimens from a patient (say). Not all medical applications could be regarded as 'scientific instruments', and therefore the relevance of this guide needs to be considered.

**DO-178B.** This is the civil avionics safety-critical software standard [25]. 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. This standard can be taken as an ideal goal, not achievable in practice, due to resource constraints.

**IEC 61508 (draft).** This standard is a generic one for safety-critical applications [12]. In contrast to the previous standard, this one allows for many methods of compliance. A catalogue is provided in Part 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 generic standard is less demanding than the previous one, and hence could be applied without the same demands on resources. Guidelines for use within the motor industry have been developed from this standard [30]

The conclusion for this Guide is that this standard is not directly relevant, 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 [41].

A very informative report [24], that has many parallells to this one, undertakes an assessment of devices which could be an instrument based upon an early draft of IEC 1508 (now 61508).

For a detailed research study of assessing instruments for safety application by means of a worked example, see the SMART Reliability study [4]. This study was based upon (an earlier edition of) this Guide, 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 [15].

The conclusion from this survey of the standards is that they are broadly similar and that aiming to meet all the requirements is a reasonable way of proceeding. One exception to this is that the very demanding requirements in DO-178B cannot be realistically merged with the others. Hence, if an instrument 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, with the exception of DO-178B (levels A and B).

## 2.2 An instrument model

In order to provide a framework for the discussion of the software for an instrument, we present here a simple model. The components of the model in Figure 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 instrument. (In the case of complex instruments, 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 instrument. A typical example would be calibration data. Another example might be a clock which could be used to 'time-out' a calibration.

**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 instrument, but the data processing software could be

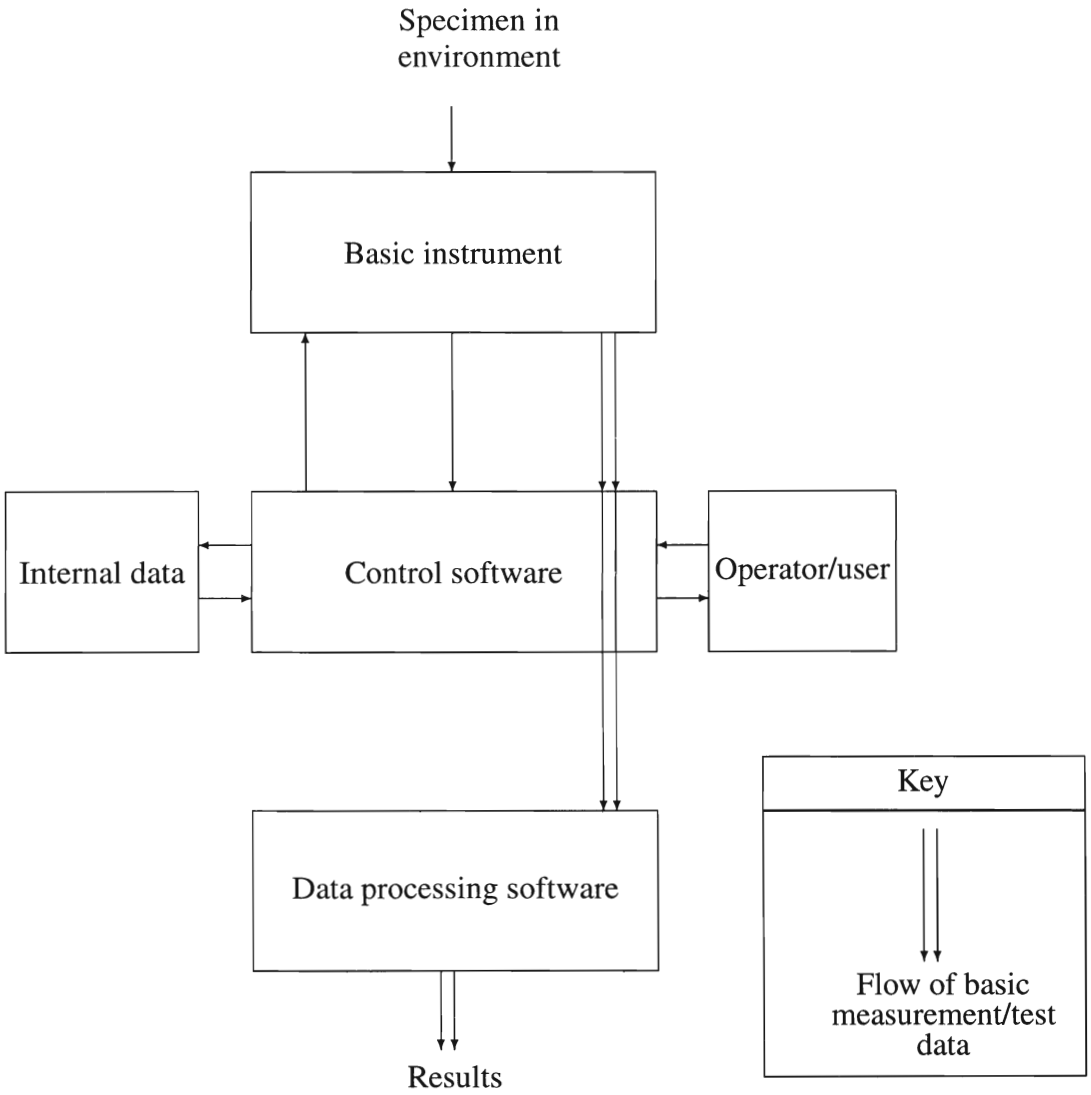



Figure 1: A model of an instrument

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 [6] on a medical application provides some useful insights. The integrity of the production of the test/measurement certification should not be forgotten [29].

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 suitable 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.

# 3 Risk factors

In this section, we undertake an analysis of a scientific instrument and its related software, the purpose of which is to make an objective assessment of the likely risks associated with a software error. For a general discussion on risk, see [9].

The first step in undertaking the risk assessment is to characterise the instrument according to aspects which influence the risk. Hence we list those aspects below.

**Criticality of Usage.** It is clear that the usage of some instruments 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**, **potentially life-critical** and **life-critical**. (The assumption in using this Guide is that some degree of criticality is involved.)

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 instrument 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!** on page 22.

**Legal requirements.** Several instruments are used in contexts for which there are specific legal requirements, such as with the WELMEC guide [32]. In this context, an instrument malfunction could have serious consequences.

To make an assessment, one clearly needs to know if there are any specific legal requirements for the instrument and have a reference to these. (It may be necessary to check what current legislation applies.)

**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.

**Very simple.** An example of a very simple control is when the instrument 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 instrument takes the operator through a number of stages, ensuring that each stage is satisfactorily complete 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.** An example of complex control is when the software contributes directly to the functionality of the instrument. For instance, if the instrument moves the specimen, and these movements are software controlled and have a direct bearing upon the measurement/test results.

**Complexity of processing of data.** In this context, we are concerned with the processing of the raw data from the basic instrument (ie, the instrument without the software). In the case of software embedded within the instrument 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.)

**Very simple.** In this case, the processing is a linear transformation of the raw data only, and 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. For an example of a problem that can arise in this area, see **Numerical instability** on page 21.

**Complex.** Anything else.

A risk assessment must be based upon the visible information. Some complex devices may internally record information which is difficult or impossible access. Examples of such information is the selection of operator options, or low-level data within a complex system. For programmable instruments, it should be possible to reconstruct the program from a listing, and repeat the execution from the data recorded from a prior execution.

# 4 Integrity assessment

At this stage, we are looking at the instrument as a black box but assuming that some questions can be asked (and answered) which might not be directly apparent from the instrument. The underlying reasoning behind the questions is to assess the affects of the risk factors involved. If some key questions can not be answered, then clearly any assessment must be incomplete.

The first part is to go through the previous section of this Guide and characterise the instrument in terms of all the parameters mentioned there.

We now have a set of additional issues to resolve as follows:

1. What degree of confidence can be obtained in the instrument merely by performing 'end-to-end' tests, i.e, using the instrument with specimens of known characteristics? (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.) Such tests just regard the entire instrument as a black-box and effectively ignores that software is involved. To answer this leading question you need to take into account the risk factors noted above. For instance, if complex software is being used which uses unpublished algorithms, then high confidence cannot be established.

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?

3. Has essentially the same software for the data processing been applied to a similar instrument 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.)

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

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

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

At this point, sufficient information should be available to make an assessment of the integrity required of the software taking into account all the factors above including the target risk to be taken. This assessment should be as objective as possible, but is bound to have a modest degree of subjectivity. If answering the six questions above is straightforward, raising no problems, then the **Software Integrity Level** is the same as the complexity of the data processing above. Hence, unless answering the above questions reveals additional problems, **very simple** complexity of data processing would have a **Software Integrity Level** of 1.

Thus classify the result of this process as follows:

**Software Integrity Level 1.** The data processing software is **very simple**. No significant problems or risks were revealed in the analysis. The control software had no impact on the measurement data.

**Software Integrity Level 2.** The data processing software is **simple** or some problems or risks were encountered and the processing was **very simple**. The impact of the control software was clear.

**Software Integrity Level 3.** There is at least one major unquantifiable aspect to the software. This could be an inability to check the software since there is no facility to provide the raw data (combined with complex processing). Another possibility might be that the control software influences the basic measurement/test data in ways that cannot be quantified.

**Software Integrity Level 4.** Here we either have complex processing which is difficult to validate, or processing of modest complexity with significant additional problems (or both!).

As the software integrity level increases, so should the risk of errors be reduced due to the application of more rigorous software engineering techniques, which is the topic of the next section.

## 4.1 Computing the software integrity level

It has been suggested that there should be an algorithm for computing the software integrity level from the information which is requested in the last two sections. Each key factor is on a 4-point scale, as is the resulting software integrity level. Hence one possibility is:

*Software Integrity Level = max(Usage, Control, Processing) levels.*

This suggestion has not been developed further since it seems to be difficult to take into account all the factors necessary. For instance, even the maximum function above is not quite correct since the complexity in the control function could be off-set by other factors.

On balance, it seems more appropriate for the factors to be determined, the check lists used, and then a subjective judgement made (which could well be based upon the formula above). The important aspect is to show how, and on what basis, the software integrity level was determined. Some alternative methods are given in [12].

# 5 Software development practice

The starting point here is that the software development process being used should have a rigour to match the software integrity level. This is the approach taken in several safety-critical software standards [12, 25].

For any software integrity level, basic practices must be established which could be a direct consequence of the application of ISO 9001 to software [17, 18] 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 levels of integrity. However, even 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 [7].

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 [22], and hence the approach taken here is the conventional one of examining the software development process. In practical terms, software testing is expensive, and hence the most cost-effective solution uses other methods in addition to gain confidence in the software.

In the UK, it is reasonable to expect suppliers to be registered to ISO 9001, which in the case of software implies the application of TickIT. If a supplier is *not* registered, then one lacks an independent audit on their quality system.

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 [21]. For those companies not formally registered to ISO 9001, we assume that a similar quality management approach is used.

The application of 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:

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. A detailed functional specification should exist. Such a specification should be sufficient to undertake the coding. This level of information is typically confidential to the developer.

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

4. The software should be under configuration control [27]. This implies that either the software should itself include the version number, or the version can be derived from 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.

| *Software Integrity Level* | *Recommended techniques* |
|---|---|
| 2 | Software inspection(C.1), Mathematical specification(C.7)<br>Structural testing(C.2), System testing(C.5) |
| 3 | Regression testing(C.3), Equivalence partition testing(C.2)<br>Independent audit(C.9), Numerical stability(C.6)<br>Stress testing(C.10), Reference test sets(C.12) |
| 4 | Statement testing(C.2), Formal specification(C.8)<br>Static analysis(C.11), Accredited testing(C.4)<br>Back-to-back testing(C.13) |

Table 1: Recommended techniques

We assume that these requirements are met, whatever software integrity level is to be addressed by the software.

For **Software Integrity Level 1**, the above requirements are recommended. For the higher software integrity levels, a recommendation for level $n$ also applies to any higher level. In Table 1, we list the recommended techniques, which are all defined in Appendix C. Note that Statement testing, Equivalence partition testing, and Structural testing are all (software) component test methods.

The fact that a technique is recommended at a specific level 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 C.4, is a good choice precisely because other strong methods are not effective. Any design should involve a trade-off between the various relevant methods.

# 6 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 an instrument may not be able to obtain information from the supplier to determine if appropriate software engineering practices have been used.

At this point, no consultation has taken place with instrument suppliers, so it is unclear if this Guide reflects current practice, although it is based upon both established software engineering methods and information from appropriate standards.

Many useful comments were obtained on the first draft of this Guide. Every attempt has been taken to reconcile the comments in this edition.

# 7 Acknowledgements

Thanks to Bernard Chorley and Graeme Parkin for support and help with this project. Brian Bibb (Phillips Medical Instruments) provided some useful references which greatly assisted this work. Peter Harris provided the material in sections on numerical issues (B.2, C.6 and C.12). Comments are acknowledged from: Dr Peter Vaswani (UKAS), Dr John Wilson (BMTA), Brian Bibb (Elektra Oncology Systems Ltd), John Stockton, Bill Lyons (Claude Lyons Ltd), Dr Richard Mellish (Medical Devices Agency) and Graeme Parkin.

# References

[1] 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.

[2] COST B2: the quality assurance of nuclear medicine software. K E Britton and E Vauramo. European Journal of Nuclear Medicine. 1993. vol 20. pp815-816. Details available on:

`http://www.phb-mpd.demon.co.uk/index.html`

[3] J Dawes. "The VDM-SL Reference Guide". Pitman Publishing. 1991. ISBN 0-273-03151-1

[4] 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:

`http://www.npl.co.uk/ssfm/download/index.html`

[5] EN 45001. General criteria for the operation of testing laboratories. CEN. June 1989.

[6] 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.

[7] Supplier Guide for Validation of Automated Systems in Pharmaceutical Manufacture. ISPE 3816 W. Linebaugh Avenue, Suite 412, Tampa, Florida 33624, USA.

[8] T Gilb and D Graham. Software Inspection. Addison-Wesley 1993. ISBN 0-201-63181-4.

[9] Guidelines on Risk Issues. The Engineering Council. February 1993. ISBN 0-9516611-7-5.

[10] ODE Guidance for the Content of Premarket Submission for Medical Devices Containing Software. Draft, 3rd September 1996.

[11] Safety-related systems — Guidance for engineers. Hazards Forum. March 1995. ISBN 0 9525103 0 8.

[12] IEC 61508: Parts 1-7, Draft. Functional safety: safety-related systems. Draft for public comment, 1998. (Part 3 is concerned with software which is the relevant part for this Guide.)

[13] IEC 601-1-4: 1996. Medical electrical equipment — Part 1: General requirements for safety 4: Collateral Standard: Programmable electrical medical systems.

[14] SEMSPLC Guidelines: Safety-Related Application Software for Programmable Logic Controllers. IEE Technical Guidelines 8: 1996. ISBN 0 85296 887 6.

[15] 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.

[16] ISO/IEC Guide 25: 1990. General requirements for the competence of calibration and testing laboratories. (This is to be replaced in 1999 by ISO/IEC 17025 with the same title.)

[17] EN ISO 9001:1994, Quality systems — Model for quality assurance in design, production, installation and servicing.

[18] ISO/IEC 9000-3: 1991. Quality management and quality assurance standards — Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software.

[19] ISO/IEC 9126:1991, Information technology — Software product evaluation — Quality characteristics and guidelines for their use.

[20] ISO/IEC 12119: 1993, Information technology — Software packages — Quality requirements and testing.

[21] ISO/IEC 12207: 1995. Information technology — Software life cycle processes.

[22] B Littlewood and L Strigini. The Risks of Software. Scientific American. November 1992.

[23] J A McDermid (Editor). Software Engineer's Reference Book. Butterworth-Heinemann. Oxford. ISBN 0 750 961040 9. 1991.

[24] Guidelines for assessment of software in microcomputer controlled equipment for safety-related systems. NT TECHN Report 287. May 1995.

[25] 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.

[26] NAMAS Accreditation Standard. M10. NAMAS. 1992.

[27] A Guide to Managing the Configuration of Computer Systems (Hardware, Software and Firmware) used in NAMAS Accredited Laboratories. NIS37. NAMAS, October 1993.

[28] Storage, Transportation and Maintenance of Magnetic Media. NIS40. NAMAS, October 1990.

[29] Use of Word Processing Systems in NAMAS Accredited Laboratories. NIS41. NAMAS, November 1991.

[30] Development Guidelines For Vehicle Based Software. The Motor Industry Software Reliability Association. MIRA. November 1994. ISBN 0 9524156 0 7.

[31] Spivey J M. Understanding Z, CUP 1988.

[32] Guide for Examining Software (Non-automatic Weighing Instruments). WELMEC 2.3, January 1995.

[33] Statistics Software Qualification: reference data sets. Edited by B P Butler, M G Cox, S L R Ellison and W A Hardcastle. Royal Society of Chemistry. 1996. ISBN 0-85404-422-1.

[34] B A Wichmann. A personal view of Formal Methods. To be published (available from the author).

[35] 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.

[36] B A Wichmann. Some Remarks about Random Testing. To be published (available from the author).

[37] The NAG Fortran Library. The Numerical Algorithms Group Ltd., Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, UK.

[38] LINPACK User's Guide. J J Dongarra, C B Moler, J R Bunch, and G W Stewart. SIAM, Philadelphia. 1979.

[39] 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.

[40] ISO/DIS 10360-6: 1996. Method for testing the computation of Gaussian associated features.

[41] 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:

`http://www.npl.co.uk/npl/cise/UKAS/index.html`

# A Check lists

The check lists below ask key questions concerning the substance of this Guide. It is not usually possible to answer them with a simple yes/no. If any question is unclear, the main text of the Guide should be referred to.

In some cases, the check list questions do not relate to the larger issues discussed in the main text, but to lesser issues which are known to have caused problems in the past.

## A.1 Risk factors

1. What is the criticality of usage? (**critical/business-critical/ potentially life-critical/life-critical**).
2. Is the supplier aware of the criticality of your application?
3. Are there specific legal requirements for the instrument?
4. What are the consequences of an instrument malfunction?
5. Is independent evidence needed of the software development process?
6. Does the instrument require regulatory approval?
7. What is the complexity of control? (**very simple/simple/ modest/complex**).
8. Does the instrument perform built-in testing?
9. Do the control functions protect the operator from making specific errors?
10. Can an error in the control software cause an error in the basic test/measurement data?
11. Is the raw data available from the instrument?
12. Does the instrument 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.

## A.2 Integrity assessment

1. What information is available from the instrument supplier or developer of the software?
2. What confidence can be gained in the software by end-to-end tests on the instrument?
3. Can the raw data be extracted from the instrument?
4. Can the raw data be processed independently from the instrument to give an independent check on the software?
5. Are software reliability figures available for an instrument using similar software (ie, 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/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 instrument require formal training?
10. Have all the answers to the questions above in this list been taken into account in determining the software integrity level?
11. Has a list been made of all the unquantifiable aspects of the software?

## A.3 Software development practice

These lists are increasing in complexity. Since the requirements at level *n* imply those at level *n-1*, all the questions should be asked up to the level required.

### A.3.1 Software Integrity Level 1

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. What steps have been taken to ensure there is no problem at the millennium change?

12. Have all changes to: hardware platform, operating system, compiler, added functionality been checked?

13. Have all corrections been checked according to the defined procedures?

14. Have all the staff the necessary skills, and are these documented?

### A.3.2 Software Integrity Level 2

1. Is software inspection used on the project? If so, to what documents has it been applied and what was the estimated remaining fault rate?

2. What alternatives have been used if software inspection was not applied?

3. Has a mathematical specification been produced of the main algorithms used for processing the test/measurement data?

4. Is the processing code derived directly from the mathematical specification?

5. What form of structural testing has been applied, and what metrics of the level of testing have been produced?

6. What level of testing has been applied to the control and processing components of the software?

7. How has the completeness of the system testing been assessed?

8. Has the system testing covered all reasonable misuses of the instrument?

9. What records are available on the system tests?

10. Are the security features consistent with any regulations or intended use?

11. Are the test strategies, cases, and test completion criteria sufficient to determine that the software meets its requirements?

### A.3.3 Software Integrity Level 3

1. Is regression testing applied? If so, at what point in the development did it start?

2. For what components has equivalent partition testing been applied? Has the technique been applied to the components processing the basic test/measurement data?

3. 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?

4. Has the numerical stability of the main measurement/data processing routines been checked? Has the rounding error analysis been taken into account in formulating the mathematical specification of the routines?

5. 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?

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

7. Are known remaining software bugs documented? Are they adequately communicated to the user?

8. What methods have been applied to ensure that structural decay is avoided? (See Appendix B.3.)

### A.3.4 Software Integrity Level 4

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

2. 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?

3. What forms of static analysis have been undertaken?

4. 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?

5. Is beta site testing undertaken?

6. Is memory utilization testing undertaken, or can it be shown that such testing is not needed?

# B Some example problems

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

## B.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 10.000 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.

## B.2 Numerical instability

The repeatability standard deviation of a weighing balance was required as part of a reference material uncertainty estimate. Successive weighings of a nominal 50 gram 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.

## B.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 customer. 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 customer 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++.

## B.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 in 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 an instrument 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 [25]. 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).

# C Recommended software engineering techniques

Here we list specific recommended techniques. These are either defined here, or an appropriate reference given. A good general reference to software engineering is [23].

## C.1 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 [8].

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

## C.2 Component testing

This 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 [1].

The BCS standard allows for many levels of testing and in this Guide we select four levels as follows:

**Structural testing.** Several forms of structural testing defined in the standard, but not to any specified level. In this context, we specify branch testing with 50% coverage.

**Equivalence partition testing.** Undertaking this to 100% coverage. This is complete functional testing at the component level. This is to be applied to those components handling the basic measurement/test data.

**Statement testing.** Undertaking 100% 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 gives rise to statements that cannot be executed (but are quite acceptable).

**Boundary value testing.** In this case, which can be seen as an addition to equivalence partition testing, values are chosen which lie of the boundary between partitions. This form of testing is designed to show that the boundary cases themselves are correctly handled.

## C.3 Regression testing

This technique 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 addition to the set of tests of an example which would detect the bug.

To be effective, one needs a method of re-running the set of tests automatically. 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 very reliable on the unextended facilities.

## C.4 Accredited testing using a validation suite

This technique 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 result 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 analysis. 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.

An example of this form of testing for a scientific 'instrument' is that being proposed in the area of Nuclear Medicine [2]. Here, gamma-camera pictures are taken of patients when being treated with substances containing radio-active trace elements. 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 analysis software. Note that the application is **potentially life-critical** and the complexity of the processing of data is **complex** which implies a high **software integrity level**, say 3. At this level, the technique of accredited testing is not actually recommended (see Table 1 on page 13), 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 the reference above.

## C.5 System-level functional testing

This technique is based upon testing the entire software as a black box by a careful examination of the functionality specified and ensuring that every aspect of the functionality is tested. An ISO standard is based upon application of this test method [20].

## C.6 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 parametrisation 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, 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 [37] (which is available in a number of computer languages and for a variety of platforms), LINPACK [38], and NPL libraries for data approximation [39] and numerical optimisation.

## C.7 Mathematical specification

Such a 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 so that 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.

## C.8 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 [3] and Z [31]. For the author's personal views of this method, see [34].

## C.9 Independent audit

In the UK, independent audit to ISO 9001 is widely established. This provides evidence to third parties of a **software integrity level** of 1. 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 section 2.1 would be better.

## C.10 Stress testing

This testing technique 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.

For a paper on this method, see [36].

## C.11 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 can be very minor (say, all variables are explicitly declared), or very strong (formal proof of correctness), but the goal of showing predictable execution should be cost-effective for high integrity software.

For a general discussion on static analysis, see [35].

## C.12 Reference test sets

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 [33], 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, data sets corresponding to problems with various 'degrees of difficulty' or condition (section C.6), 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 ISO Standard [40] 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.

## C.13 Back-to-back testing

In this form of testing two comparable software systems are tested with the same input. The output from each test is then compared — identical results are not usually expected when numerical testing is undertaken.

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 to other is the item being assessed.

In the SMART reliability study[4], this form of testing was used by testing a MatLab implementation against the C code within the instrument. The test cases used were those derived from boundary value/equivalence partition testing.

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

# D Background to this Guide

This Guide has been produced under the NMSPU programme that NPL has agreed with the Department of Trade and Industry whose support in acknowledged.

The author is a software engineer rather than an expert on scientific instruments. Hence several NPL staff have been asked to comment on drafts of the Guide.

Every attempt has been made to produce the Guide from existing material. However, nothing was found in the literature which matched the requirements. Reference has not been made to the ISO generic standard on software quality [19], since it has no requirements.

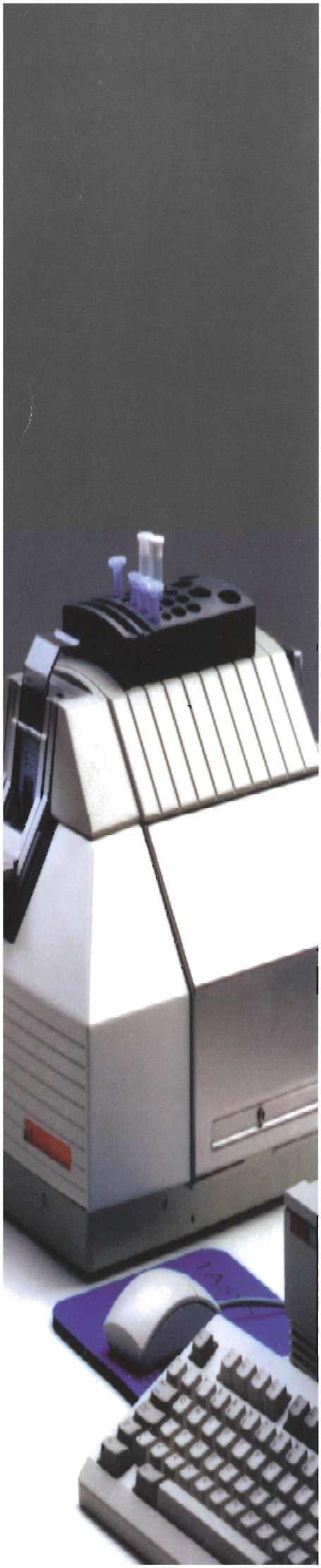
This revision of the Guide has been undertaken with the requirements for the Measurement System Validation activities of the Software Support for Metrology (SSfM) programme in mind. The SSfM programme is much broader than the objectives of this Guide and will produce corresponding reports in due course. However, additions to this Guide have been made in those areas consistent with the narrower objectives of the *Competing Precisely* guides.

# E Comments

To comment on this Guide, please give precise details of the issue.

They can be sent by a variety of means, which in order of preference are:

1. e-mail to `brian.wichmann@npl.co.uk`
2. Fax to 020-8977 7091 marked for the attention of Brian Wichmann
3. by post to Brian Wichmann at the address on the title page.



*Recommended sale price £15.00*

1725/PPG/A1874/0.25K/9908