Quality Assurance for Software Important to Safety
QUALITY ASSURANCE
FOR SOFTWARE
IMPORTANT TO SAFETY
The following States are Members of the International Atomic Energy Agency:

AFGHANISTAN  ALBANIA  ALGERIA  ANGOLA  ARGENTINA  ARMENIA  AUSTRALIA  AUSTRIA  BANGLADESH  BELARUS  BELGIUM  BENIN  BOLIVIA  BOSNIA AND HERZEGOVINA  BRAZIL  BULGARIA  BURKINA FASO  CAMBODIA  CAMEROON  CANADA  CHILE  CHINA  COLOMBIA  COSTA RICA  COTE D'IVOIRE  CROATIA  CUBA  CYPRUS  CZECH REPUBLIC  DEMOCRATIC REPUBLIC OF THE CONGO  DENMARK  DOMINICAN REPUBLIC  ECUADOR  EGYPT  EL SALVADOR  ESTONIA  ETHIOPIA  FINLAND  FRANCE  GABON  GEORGIA  GERMANY  GHANA  GREECE  GUATEMALA  HAITI  HOLY SEE  HUNGARY  ICELAND  INDIA  INDONESIA  IRAN, ISLAMIC REPUBLIC OF  IRAQ  IRAQ  ISRAEL  ITALY  JAMAICA  JAPAN  JORDAN  KAZAKHSTAN  KENYA  KOREA, REPUBLIC OF  KUWAIT  LATVIA  LEBANON  LIBERIA  LIBYAN ARAB JAMAHIRIYA  LIECHTENSTEIN  LITHUANIA  LUXEMBOURG  MALAWI  MALAYSIA  MALI  MALT  MARSHALL ISLANDS  MARTINIQUE  MEXICO  MONACO  MONGOLIA  MOROCCO  MOZAMBIQUE  MEXICO  MEXICO  MYANMAR  NAMIBIA  NAMIBIA  NIGER  NIGERIA  NETHERLANDS  NEW ZEALAND  NICARAGUA  NORTH KOREA  NORTHERN IRELAND  NORWAY  PAKISTAN  PANAMA  PARAGUAY  PERU  PHILIPPINES  POLAND  PORTUGAL  QATAR  REPUBLIC OF MOLDOVA  ROMANIA  RUSSIAN FEDERATION  SAUDI ARABIA  SENEGAL  SIERRA LEONE  SINGAPORE  SLOVAKIA  SLOVENIA  SOUTH AFRICA  SPAIN  SRI LANKA  SUDAN  SWEDEN  SWITZERLAND  SYRIAN ARAB REPUBLIC  THAILAND  THE FORMER YUGOSLAV  REPUBLIC OF MACEDONIA  TOGO  TUNISIA  TURKEY  TURKMENISTAN  UKRAINE  UNITED ARAB EMIRATES  UNITED KINGDOM OF GREAT BRITAIN AND  NORTHERN IRELAND  UNITED STATES OF AMERICA  UZBEKISTAN  URUGUAY  VENEZUELA  VIET NAM  YEMEN  YUGOSLAVIA  ZAMBIA  ZIMBABWE

The Agency's Statute was approved on 23 October 1956 by the Conference on the Statute of the IAEA held at United Nations Headquarters, New York; it entered into force on 29 July 1957. The Headquarters of the Agency are situated in Vienna. Its principal objective is "to accelerate and enlarge the contribution of atomic energy to peace, health and prosperity throughout the world".

© IAEA, 2000

Permission to reproduce or translate the information contained in this publication may be obtained by writing to the International Atomic Energy Agency, Wagramer Strasse 5, P.O. Box 100, A-1400 Vienna, Austria.

Printed by the IAEA in Austria
December 2000
STI/DOC/010/397
QUALITY ASSURANCE FOR SOFTWARE IMPORTANT TO SAFETY
Quality assurance for software important to safety. — Vienna : International
   p. : 24 cm. — (Technical reports series, ISSN 0074–1914 ; no. 397)
STI/DOC/010/397
ISBN 92-0-102500-9
Includes bibliographical references.
   1. Nuclear reactors—Safety measures—Computer programs—Quality
   control. I. International Atomic Energy Agency. II. Series : Technical
   reports series (International Atomic Energy Agency) ; 397.

VICL 00-00253
FOREWORD

Software applications play an increasingly relevant role in nuclear power plant systems. This is particularly true of software important to safety used in both:

(a) calculations for the design, testing and analysis of nuclear reactor systems (design, engineering and analysis software); and
(b) monitoring, control and safety functions as an integral part of the reactor systems (monitoring, control and safety system software).

Computer technology is advancing at a fast pace, offering new possibilities in nuclear reactor design, construction, commissioning, operation, maintenance and decommissioning. These advances also present new issues which must be considered both by the utility and by the regulatory organization. Refurbishment of ageing instrumentation and control systems in nuclear power plants and new safety related application areas have emerged, with direct (e.g. interfaces with safety systems) and indirect (e.g. operator intervention) implications for safety.

Currently, there exist several international standards and guides on quality assurance for software important to safety. However, none of the existing documents provides comprehensive guidance to the developer, manager and regulator during all phases of the software life-cycle.

The present publication was developed taking into account the large amount of available documentation, the rapid development of software systems and the need for updated guidance on “how to do it”. It provides information and guidance for defining and implementing quality assurance programmes covering the entire life-cycle of software important to safety. Expected users are managers, performers and assessors from nuclear utilities, regulatory bodies, suppliers and technical support organizations involved with the development and use of software applied in nuclear power plants.

The IAEA wishes to acknowledge the efforts and assistance provided by the contributors and participants at the preparatory and review meetings, listed at the end of this publication.
EDITORIAL NOTE

The mention of names of specific companies or products (whether or not indicated as registered) does not imply any intention to infringe proprietary rights, nor should it be construed as an endorsement or recommendation on the part of the IAEA.
| APPENDIX V: | GENERAL TRAINING GUIDELINES | 88 |
| APPENDIX VI: | PROPOSED OUTLINES FOR TRAINING PROGRAMMES ON QUALITY ASSURANCE | 91 |
| APPENDIX VII: | CHARACTERISTICS OF DEFECT PREVENTION PROCESS | 96 |
| APPENDIX VIII: | EXAMPLES OF SOFTWARE DEVELOPMENT LIFE-CYCLE MODELS | 98 |
| APPENDIX IX: | RECOMMENDATIONS FOR DESIGN INPUT DOCUMENTATION FOR MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE | 107 |
| APPENDIX X: | RECOMMENDATIONS FOR SOFTWARE DEVELOPMENT PLANS APPLICABLE TO MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE | 108 |
| APPENDIX XI: | RECOMMENDATIONS FOR STANDARDS AND PROCEDURES HANDBOOKS APPLICABLE TO MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE | 110 |
| APPENDIX XII: | RECOMMENDATIONS ON THE CONTENT OF SOFTWARE REQUIREMENTS SPECIFICATIONS FOR MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE | 113 |
| APPENDIX XIII: | RECOMMENDATIONS ON SOFTWARE DESIGN DESCRIPTIONS FOR MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE | 115 |
| APPENDIX XIV: | RECOMMENDATIONS ON DESIGN AND DEVELOPMENT DOCUMENTS FOR DESIGN, ENGINEERING AND ANALYSIS SOFTWARE | 118 |
| APPENDIX XV: | RECOMMENDATIONS ON APPLICATION DOCUMENTS FOR DESIGN, ENGINEERING AND ANALYSIS SOFTWARE | 122 |
| APPENDIX XVI: | SUGGESTED GOOD CODING PRACTICES FOR DESIGN, ENGINEERING AND ANALYSIS SOFTWARE | 125 |
| APPENDIX XVII: | RECOMMENDATIONS ON PROGRAMMING OF MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE | 127 |
| APPENDIX XVIII: | DISCUSSION OF VERIFICATION AND VALIDATION METHODS | 130 |
1. INTRODUCTION

A fundamental principle underlying the assurance of quality of any product or service is that the processes used to create this product or service form part of a disciplined, systematic and methodical approach to achieving the required quality. In this respect, software\(^1\) should be considered as an engineered product in a manner that is no different from any other product.

Software development is different because the product is complex compared to other products of human manufacture. The complexity seems to depend on the fact that computer programs\(^2\) have many decision points and the decisions at these points lead to a large number of paths that can be followed. In a large computer program there can be an extremely large number of paths through the maze, as exemplified by the lengthy test programs required to verify the operation of these paths. It becomes difficult for the logic designer and programmer to think their way through all these possibilities. When sections of a computer program are the responsibility of different designers, it becomes even more difficult.

Software appears especially complex to the human brain because software does not have a structure in space. Humans are very good at seeing spatial relationships. From engineering drawings we can visualize how a product or building will appear. There is no natural counterpart to the engineering drawing in software development. Flow charts, indented source program\(^3\) listings and many other structuring techniques represent attempts to match a plan with the eventual product, but they all fall short of the natural simplicity of the engineering drawing.

One basic need for software quality assurance\(^4\) concerns keeping the potential for latent defects in the software to a minimum. One of the main objectives of a software quality assurance programme is to reduce the likelihood of defects getting into the executable code\(^5\) by applying appropriate, systematic techniques throughout the software life-cycle\(^6\).

---

1 Software — computer programs, procedures, rules and associated data and publications pertaining to the operation of a computer program.

2 Computer program — a sequence of instructions suitable for processing by a computer.

3 Source program — a computer program that must be compiled or interpreted before being executed by a computer.

4 Software quality assurance — a planned and systematic pattern of actions designed to provide adequate confidence that a computer program will be of the required quality.

5 Executable code — a computer program in a language that can be directly executed by a computer.

6 Software life-cycle — the period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life-cycle typically includes a concepts phase, requirements phase, design phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes a retirement phase.
Latent defects are not the only problem, however. Many computer programs do not do the job that they were specified to do. A computer program that is poorly documented or reflects complex rather than straightforward programming techniques is hard to understand, test or debug. Problems associated with the development, operation and maintenance of software also include a lack of reliability, difficult maintenance, inaccurate requirements specifications, inefficient use of computer hardware resources, inadequate testing and poor documentation.

Nevertheless, the above mentioned difficulties and other project risks, such as cost, have not prevented the successful development and application of software to most engineering areas, including the nuclear industry. The discipline of software engineering has matured greatly in the past decades, and several methods, tools and practices have been put forward to address the intrinsic difficulties of software and also its rapidly changing nature due to technological advances. A present day vision of software as a product comprises much more than its code and documentation, extending to its requirements specification, design description, testing tools, verification and validation results and all records of its changes and updates. Similarly, the software development process is viewed as a life-cycle in which several concurrent and consistent plans are put forth to systematically manage its evolution, from different points of view, such as project management, production, verification and validation, documentation, safety and quality assurance. Thus, in this context, many of the issues addressed in a software quality assurance programme are intimately related to other plans, and may appear to overlap with them. This fact, when plans are consistent and mutually supportive, is a reflection of the modern idea that quality assurance is everyone’s responsibility and that quality can only be built in, not inspected in at the end.

The assurance given by the application of a software quality assurance programme is important to safety authorities, since it provides confidence during the plant licensing process that the product is in fact of the required quality. Assurance is given by assessments such as quality assurance audits, software quality assurance programme reviews, and verification, validation and testing activities, which

---

7 Debugging — the process of locating, analysing and removing errors.
8 Hardware — physical equipment used in the execution of computer programs and the storage or transfer of data.
9 Verification — the process of ensuring that a phase in the software development process meets the requirements imposed on it by the previous phase.
10 Validation — testing conducted on a completed, integrated computer system (hardware and software) to ensure compliance with the requirements specification.
11 Testing — the process of exercising a computer program or portion thereof to verify that it satisfies specified requirements.
show that each specified process has in fact been followed properly and managed correctly, and the resultant products meet all technical requirements.

1.1. BACKGROUND

Computer programs play an increasingly important role in the design, analysis and operation of nuclear power plants. This tendency can be expected to be sustained owing to technological advances in computer based systems, the widespread availability of engineering software tools and the replacement of older analog systems. Also, new plant designs, which include computer based systems important to safety, have come into operation in recent years, thus providing operational experience and a field record for evaluation of software attributes.

Because of unsatisfactory experiences in other industries, the introduction of computer programs to perform critical functions in nuclear power plant design and in control and monitoring applications has caused significant concern, especially since the nuclear industry has very stringent safety requirements. The introduction of computer based systems in nuclear power plant applications has not been easy, since new methods and criteria have been needed to rigorously assess and judge the safety and integrity of such software applications.

Existing software quality assurance programmes established by software suppliers, nuclear designers and nuclear power plant utilities represent each organization's interpretation of what is required for the control of software. The knowledge about software quality assurance techniques varies widely across the nuclear industry. Therefore, the effectiveness of the software quality assurance programmes also varies across the industry.

Appropriate quality assurance programmes provide a framework for appropriate management systems within which the necessary work processes can be defined and implemented in a controlled environment. Quality assurance programmes also define mechanisms for demonstrating that the defined processes have been properly applied and that the resulting computer programs satisfy all known requirements.

A key role of software quality assurance programs is to provide confidence that nuclear safety is not compromised where computer programs are involved. Other important functions of the software quality assurance programme include providing confidence that the work involving software is performed on schedule and within budget. The failure to control such elements of the work environment can put nuclear safety at risk.

An effective quality assurance programme for computer based systems fits within a broader context of general plant/project quality assurance requirements. In
particular, recommendations for software quality assurance are adaptations of generally proven and accepted quality assurance principles.

The basic requirements for the quality assurance functions of managers, performers and assessors are the same as for other plant systems. In this context, the functions of a quality assurance department include:

— Giving independent advice to the line management and engineers on management processes and on technical methods without having technical responsibility for them;
— Monitoring independently that defined methods and management processes are sufficient for assurance of an identified quality and requiring corrective action where methods were not accurately followed;
— Providing impartial reports that the processes used in the generation of software were those defined by the technical departments concerned with implementation and that they were followed responsibly and completely.

FIG. 1. Quality assurance categories: requirements.
To ensure an appropriate degree of independence from those managing or performing the work, the responsibilities of the quality assurance department normally exclude software system implementation and verification and validation activities themselves, or the definition of the actual procedures followed in the management of the processes.

Figure 1, based on Ref. [1], shows the requirements applicable to the parties involved in the project life-cycle. Software calls for specific practices, tools and techniques for life-cycle management, development, verification and validation. These practices, tools and techniques change in time and according to the application's context and have demonstrated significant advances in recent years [2, 3].

Computer hardware technology continues to evolve at a rapid pace. The same is true for various computer based tools to facilitate the design, development, testing, verification, validation and use of computer programs. Therefore, related quality assurance programmes must continuously adapt to this changing environment.

Information and recommendations applicable to nuclear power plant computer based monitoring, control and safety systems, which have been issued in the last decade, continue to be supplemented at national and international levels. Information about software quality assurance programmes is closely interrelated with information on development, verification and validation.

This publication is intended to help define and implement appropriate quality assurance programmes applicable to computer programs over their life-cycle. It provides practical suggestions on how to implement effective quality assurance programmes applied to software important to safety. The information is intended to be consistent with and to supplement Ref. [1]. Additionally, it updates and complements Ref. [4]. This publication also provides information about the framework provided by software quality assurance programmes within which software engineering practices, described in Ref. [2], and verification and validation activities, described in Ref. [3], should be applied.

This publication also gives information to assist quality assurance, management and performing staff in discharging their responsibilities effectively, in the context of software important to safety.

All recommendations and information in this publication are of an advisory nature and must not be interpreted as mandatory requirements.

1.2. OBJECTIVE

This report provides information and guidance for defining and implementing quality assurance programmes applied to software important to safety.
1.3. SCOPE

This publication informs management, quality assurance staff and performing and assessing staff about their roles within quality assurance programmes applicable to the following two broad categories of software important to safety:

— Calculations for the design, testing and analysis of reactor systems (such as testing and analysis tools, and scientific and engineering tools including stress, safety and seismic analyses);
— Monitoring, control and safety functions, where such software forms an integral part of the reactor systems (such as data logging programs, databases applied to collect data such as inventories and system configurations, plant display systems, reactor control systems, safety systems and emergency shutdown systems).

1.4. USERS

The intended readership of this report includes those responsible for the quality assurance function and managers, performers and assessors from suppliers, utilities, technical support organizations and regulatory bodies.

Line managers of suppliers of nuclear power plant architect/engineering/support services and computer based products may find useful information in the report on quality assurance programme implementation, maintenance and evaluation.

Performers who are part of software development, verification and validation, operation and maintenance teams may find general reference information describing the framework of a quality assurance programme applicable to their area of work.

Assessors, including reviewers and auditors for both technical and programme issues, will find information in this report concerning the scope, type and practices for specific development or procurement stages.

The intended readership also includes software development engineers, and the managers of quality assurance or of software implementation or review groups, or members of such groups, interested in familiarization with the quality assurance methods appropriate for high quality software.

The publication reviews the methods currently used by software engineers and implementation groups to provide high quality software, and gives references to other relevant IAEA publications.

Other IAEA publications give guidance on software technology, software methods and verification and validation methods, for readers who include managers and software engineers responsible for the implementation, review and verification and validation of software important to safety. Interested readers should refer to those publications for relevant information in these areas.
1.5. STRUCTURE

This publication is organized into sections, basically arranged for sequential reading. The main sections (2, 3 and 4) follow the same sequence as those in Ref. [1], thus addressing quality assurance requirements applicable to management, performance and assessment.

Section 1 presents a general introduction to the subject of software quality assurance.

Section 2 discusses the management functions involved in the implementation of a software quality assurance programme. Several ways of classifying types of software are also discussed. Specific management issues related to software quality assurance, training and qualification, non-conformance control and corrective action, and document control and records are discussed.

Section 3 deals with software quality assurance programme aspects related to work planning and control, software design and development, procurement, and inspection and testing. Information is provided on software life-cycle models which are the basis for planning design, documentation, development, verification and validation and control activities. Some constructive approaches to achieving reliable software are discussed.

Section 4 addresses aspects related to the assessment of software quality assurance programmes. Issues related to management self-assessment and independent assessment are discussed.

Special terms are defined in footnotes in the body of this publication where they first appear. In addition, the same terms are defined in alphabetical order in a glossary in Section 5.

A set of appendices provide additional details on topics discussed in the body of the publication.

This publication should be read together with a number of other IAEA publications that provide additional information in specific areas. These include Refs [2–5].

2. MANAGEMENT

2.1. SOFTWARE CLASSIFICATION

There is a large variety of computer programs used in the design and operation of a nuclear power plant. While the general quality assurance principles are applicable to all types of software, there are differences in the detailed quality assurance
requirements and the implementation of quality assurance programmes depending on the kind of computer program involved.

The question has to be addressed in some systematic fashion which quality assurance programme is appropriate for what kind of computer program. Therefore, it is common practice to classify computer programs depending on the point of view adopted to identify distinguishing characteristics.

Software used in nuclear power plants can be classified from different perspectives: safety, development process and run-time environment. Each classification is useful to prioritize and recommend certain quality assurance activities, since life-cycles, hardware environment and the verification and validation process differ for each type.

In this publication, computer programs are classified according to three possible points of view:

— safety;
— development status;
— run-time.

These three options are well established and are widely used. However, under special circumstances other ways of classifying software may be more appropriate and would be equally valid.

A gradation according to safety importance may also be considered for design, engineering and analysis software. Those computer programs that are used to perform safety analysis at the plant, system or component level have higher importance to safety than those used to facilitate design or evaluate performance. In Appendix I, a distinction is made among them, naming them as safety analysis software and performance evaluation software.

2.1.1. Classification from the safety point of view

Reference [6] establishes the concept of classification of nuclear power plant systems according to their importance to safety, while Refs [7, 8] apply classification to instrumentation and control systems. These guides establish the distinction between the safety systems\textsuperscript{12} and safety related systems\textsuperscript{13}, which together form the systems important to safety.

\textsuperscript{12} Safety system — system important to safety, provided to ensure the safe shutdown of the reactor or the residual heat removal from the core, or to limit the consequences of anticipated operational occurrences and design basis accident conditions.

\textsuperscript{13} Safety related system — system important to safety which is not included in safety systems.
The various types of software used for design and analysis and for monitoring and control functions in a nuclear power plant have to be developed and used under a quality assurance programme that is consistent with that applied to the reactor systems and components which are designed, analysed, or operated using these computer programs. Therefore, the same concepts apply to software as to the reactor systems and components regarding their classification according to their importance to safety.

The safety categories for software which are used as part of the reactor instrumentation and control systems may be derived from the classification of the systems of which the software forms a part. Therefore, it is possible to use the following divisions [7, 8]:

- important to safety
  - safety
  - safety related
- not important to safety.

Software used in design, engineering and safety analyses and assessments may also be classed in these safety groups.

The principles of IAEA classification have been interpreted by the International Electrotechnical Commission (IEC) in the standard IEC 1226 [9], which defines a method of safety categorization. The standard identifies categories A, B and C for functions, systems and equipment of instrumentation and control systems that are important to safety. The definitions of these categories are simplified as follows:

- Category A is assigned to functions, systems and equipment which have a principal role in the achievement or maintenance of nuclear power plant safety (IEC 1226, chapter 5.2.1).
- Category B is assigned to functions, systems and equipment which have a supporting safety role to systems of category A (IEC 1226, chapter 5.2.2).
- Category C is assigned to functions, systems and equipment which have an auxiliary or indirect role in the achievement or maintenance of nuclear power plant safety (IEC 1226, chapter 5.2.3).

The remaining functions, systems and equipment are assigned to be 'unclassified' [9].

Individual countries have other methods of identification and categorization of functions, systems, equipment and items important to safety. These are defined through national and other standards. This publication should be interpreted in such cases in accordance with the classification criteria of the applicable standard or method of classification.
Additional information can be found in Refs [3, 9].

Reference [1], para. 204, and Ref. [10], paras 209–211, recommend that quality assurance programmes be designed to reflect the importance to safety of the item, service or process. Software quality assurance programmes should be constructed to reflect this principle. Appendix I provides an illustration of how a software quality assurance programme can be constructed that takes into consideration several levels of quality grading according to the safety significance of the software.

It should be recognized, however, that the classification of software and the application of an appropriate type of quality assurance programme should not be dictated solely by the applicable safety characteristics. Safety alone does not drive the requirement for software integrity. The nuclear power plant owner may place a higher requirement on the reliability of the computer system than that required by safety considerations alone. This may be due to availability requirements, the importance of the system to plant operation or the novelty of the design of the system. Therefore, the nuclear power plant owner may require the software to be treated as if it belonged to a high safety category, and to be subject to the more rigorous verification and validation which is needed for a high level of assurance of performance and reliability.

Safety categorization of the software product is a relevant factor not only for the definition of an appropriate software quality assurance programme, but also to define expected performance requirements and availability\textsuperscript{14} goals. Previous experience and economic considerations are also generally taken into account. In certain cases, a balance of these factors may lead to a decision to upgrade the quality assurance requirements for a computer based system of low importance to safety.

2.1.2. Classification from the development status point of view

Software may be divided into the following categories on the basis of its development process history (Ref. [3]):

— Custom developed software
— Pre-developed software\textsuperscript{15} that is:
  — accessible, or
  — proprietary, or
  — configurable.

\textsuperscript{14} Availability — the fraction of time that a system is actually capable of performing its mission.

\textsuperscript{15} Such software is also called ‘commercial off the shelf (COTS) software’.
Custom developed software is developed for a particular application, starting from a system specification which may extend to its hardware platform and interfaces. Its development process is tailored to its safety category, and should include all necessary management, technical development, verification and validation and quality assurance plans, activities, controls and documentation.

The use of pre-developed software is becoming commonplace in many applications, both in nuclear power plant monitoring, control and safety system software, and in the design, engineering and analysis areas. In the latter, commercial software packages are very mature, with a large installed user base and good cost advantages. A large user base also exists for proprietary software internally developed within organizations, which is reused by modifying and tailoring it to specific new applications. In monitoring, control and safety system applications, the use of pre-developed software is driven by the availability of base software from previous similar applications, its shorter development times, and the availability of configurable platforms.

Proprietary software is the most restrictive of pre-developed software types and may be subject to restrictions on its application and, in some cases, may require negotiation of specific procurement contract conditions. The verification and validation issues for these software classes are discussed in depth in Ref. [3].

2.1.3. Classification from the run-time point of view

Software is used for different applications in nuclear power plants. A broad division can be made between:

— software running in real time within monitoring, control and safety systems; and
— software applications for design, engineering and analysis uses running not in real time.

The relevant difference between these two groups is the software–hardware interface, the management and control of which is more critical in the case of monitoring, control and safety system computer programs.

Software performing monitoring, control and safety functions is found in reactor instrumentation and control systems and may have to comply with the highest safety category requirements, as it has the same safety importance as the system to which it pertains. Also, it usually has to meet stringent operational requirements. It can be categorized, following Refs [7, 8], as important to safety if it belongs to a safety system or a safety related system.
Examples of such computer based systems include:

— reactor protection systems;
— on-line core calculators;
— nuclear channel measurement and channel trip systems for safety systems;
— supervision and control, alarm annunciation, data acquisition and recording systems for safety related systems.

Examples of systems in use can be found in Refs [2, 3].

For monitoring, control and safety system software for safety applications, custom developed software is necessary with its associated custom development life-cycle. This development life-cycle should take into consideration early design concepts to meet plant and system level hardware requirements, including a structured, integrated and balanced approach for the allocation of safety functions to hardware or software.

Software used in design, engineering and analysis of reactor systems comprises computer programs for neutronic, thermal-hydraulic, shielding, mechanical, electrical and electronic purposes; environmental modelling and simulation; computer aided design tools, such as computer assisted drafting (CAD) and computer assisted software engineering (CASE) programs; computer based testing equipment; and software tools for verification and validation, maintenance, planning, data management and training. A number of computer programs in this broad class have an indirect effect on nuclear power plant safety, and thus may be important to safety. Since plant performance, availability, software functionality and cost are also important considerations for such computer programs, they should be developed, configured or purchased according to defined quality assurance requirements.

2.2. QUALITY ASSURANCE PROGRAMME

2.2.1. Software quality assurance programme purpose, definition and implementation

The fundamental purpose of any quality assurance programme is to provide a stable and controlled environment so that:

— work is performed in a controlled and predictable manner;
— completed work is examined at appropriate stages of completion to confirm that it has been performed correctly, thereby giving confidence that the outcome of any work activity meets customers' expectations.
The purpose of the quality assurance programme during the software life-cycle is to identify, to all concerned, a basis for the control of all activities affecting quality, and to ensure that the specified quality is achieved. Performance of these activities according to the defined and documented procedures and work instructions for software is an important contribution towards the ultimate success of the nuclear power plant project.

An effective software quality assurance programme consists of a system of planned and systematic activities to achieve the required software qualities. Such a systematic approach to managing the work associated with software ensures that the materials, data, supplies and services conform to established technical requirements, and that they perform satisfactorily. The essence of an effective software quality assurance programme is to prevent problems, to remove defects as they are found, and to contribute to the usability\textsuperscript{16} and maintainability\textsuperscript{17} of the software. An effective software quality assurance programme also enhances the confidence that any software development or modification can be completed within the expected time and budget.

Implementation of the software quality assurance programme requires that all participants understand the purpose of the programme and what is expected of them within that programme. It is essential that the management and staff participate actively in the definition, documentation, implementation and maintenance of the software quality assurance programme under which they perform their work activities. The implementation process should not be an initiative solely of the quality assurance function within the organization. Such an approach would have the effect that the resulting quality assurance programme is imposed by the quality assurance function on the rest of the organization.

A detailed discussion of the recommended elements of a software quality assurance programme is given in Ref. [4] and in many national and international standards, some of which are identified in the Bibliography.

\subsection*{2.2.2. Software quality assurance programme issues}

Many documents have been published providing direction and advice on the appropriate elements of effective quality assurance programmes. It is important to recognize that all these recommendations are really aimed at how the work should be

\textsuperscript{16} Usability — the ease with which the user can learn to operate, prepare inputs for, and interpret outputs from a computer program.

\textsuperscript{17} Maintainability — the ease with which a computer program or component thereof can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.
managed. Modern quality assurance programmes are really management systems. Therefore, the terms 'quality assurance programme' and 'quality management system' should be treated as synonyms. Quality assurance standards and related supporting publications are really efforts to describe models for effective management systems. In the final analysis these standards and supporting publications can be reduced to the following fundamental elements applicable to any quality management system:

— Define the structure of the organization and the roles and responsibilities of positions within the organization;
— Establish and document proven work practices used to perform all functions within the organization;
— Apply methods to verify that the work performed complies with technical and quality requirements;
— Apply methods to confirm that the organization's work practices have been applied while performing the work;
— Identify and implement improvements to the organization and its work processes.

This cycle as first suggested by W. Shewart in the 1930s and as known today as the Deming wheel is illustrated in Fig. 2.

The recommendations on software quality assurance made in this publication are simply adaptations of generally proven and accepted principles applicable to any quality management system.

FIG. 2. Quality management system cycle: Deming wheel.
Computer programs form an integral part of many components in complex products, including nuclear facilities. Computer programs are also used widely to design and to control the manufacture of these products. Therefore, computer programs can have a significant impact on the safety of such products, including nuclear facilities.

The consequences of computer program failures can be very serious. Some examples are given to indicate the severity of such consequences and to stress the importance of an effective quality assurance programme to prevent such failures.

One example is the experience with the Therac-25 medical radiation treatment machine in 1986. The Therac-25, introduced in 1983, was among the first of a generation of computer controlled linear accelerators designed for medical radiation treatment. To protect against overexposures from the electron beam without the target in place, earlier accelerators with electromechanical controls were equipped with protective circuits. In the Therac-25 both the electron beam intensity and the target position were computer controlled. When the operator switched the machine from X ray to electron mode, the computer was counted on to set the beam to low intensity before the target was withdrawn. The Therac-25 control software contained an error which under specific circumstances allowed the target to be withdrawn while the high intensity beam remained on. The faulty operating sequence was initiated if the operator first selected the X ray mode and then switched to electrons by hitting the up-arrow key and typing over the previous instruction. This software error was compounded by the design of the user interface. The Therac-25 reportedly issued as many as 40 error messages a day, usually because the beam intensity was slightly less than it should be. The error messages referred to specific problems by an identification number rather than describing the nature of the error in words. Operators could clear the error messages by typing the letter P. They learned to respond this way to any error message. The result was several fatalities and mutilating injuries.

The European Space Agency's Ariane 5 rocket launch failed in June 1996. The cause was determined to be an error in the guidance system control software which halted the operation of both inertial reference systems. An investigation found that the guidance software design, which was appropriate for the previous Ariane 4 model, was not modified to accommodate significant hardware and systems changes in Ariane 5.

In June 1994 Canadian newspapers reported that the forward turret gun on a new Canadian frigate accidentally fired four shells during a weapons test as a result of an error in a computer program. Fortunately the shells fell into the sea without causing any harm.

In the nuclear industry, just as in the defence and aerospace related industries, safety analyses are mandatory. But safety engineering practices are found to be
different in many countries and in many applications. In a general sense, safety engineering is the process of engineering analysis and management practices that control risk associated with the use of a system in a given operational context. Some of the common steps are:

— identify the hazards in a system;
— determine the underlying causes of those hazards;
— develop engineering or management controls to either eliminate the hazards or mitigate their consequences;
— verify that controls are adequate and in place;
— monitor the system after any changes and apply additional controls, as necessary.

This general top-down approach, when applied at the design phase to systems including software, can be very effective in identifying risks associated with potential software failures, including necessary restrictions at the software requirements/design input phase. Some design criteria, such as singleness of purpose, simplicity\textsuperscript{18}, redundancies\textsuperscript{19} and testability\textsuperscript{20}, can be established at an early stage when they can be cost effectively built into the software products. Other sections in this publication elaborate further on safety issues, including Section 2.2.2.2 and Appendix 18. Practical guidance in general system safety engineering can be found in Refs [11, 12].

2.2.2.2. Reliability issues

Software reliability\textsuperscript{21} is the probability of failure free operation in a specified environment for a specified time. A common measure of reliability is mean time to failure (MTTF), where time refers to computer program execution time or operation time. It is the average length of time the computer program runs before it fails.

\begin{enumerate}
\item Simplicity — degree to which a system or component has a design and implementation that are straightforward and easy to understand.
\item Redundancy — provision of alternative (identical or diverse) elements or systems, so that any one can perform the required function regardless of the state or failure of any other.
\item Testability — degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. Or, degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.
\item Reliability — probability that a device, system or facility will perform its intended function satisfactorily for a specified time under stated operating conditions.
\end{enumerate}
Software reliability depends on the number of errors remaining in the computer program after it is released for use. In order to estimate measures of reliability, such as MTTF, an estimate of the errors remaining in the computer program after release is needed. The number of remaining errors is a function of the rate of error insertion and subsequent removal during the development life-cycle. In order to plan for a specific level of reliability of the computer program when released for use, it is therefore important to estimate the rate of error insertion and subsequent removal for the software development practices used in an organization. Also, with this information, management can judge the effectiveness of defect prevention and defect removal practices.

During the operation of the vast majority of conventional products, users can recognize that a failure has occurred. The product no longer functions as intended. In contrast, both monitoring, control and safety system software and design, engineering and analysis software can fail without the user realizing it under certain circumstances. A computer program can continue to operate, even though it may have made an erroneous decision internal to its logic that is not visible to the user. For example, a finite element stress analysis program limited to predict stresses in the elastic range may carry out a number of calculations that would indicate a transition into the plastic range. If the results of these calculations are not reported to the user in the computer program output and if the program does not contain a routine to generate an appropriate warning message, the user will be unaware that the program has gone beyond its range of applicability. Therefore, users of monitoring, control and safety system software and design, engineering and analysis software should be very familiar with the nature of the physical processes that the computer program either monitors or predicts, in order to be able to judge whether the results presented by the computer program are reasonable.

IEC standard 1069 [13] provides a structured and systematic method for assessing and evaluating products for their suitability for an application. On the basis of this standard, but with adaptations, the following properties may be used:

**Functionality**

Functionality describes the extent to which one or more functions are performed, and the nature of those functions. The functionality of a system depends on the range of functions provided, the capability to execute the functions in real time (or at the required time), and the flexibility to select and implement the necessary functions if and when they are required. The functionality of pre-developed products should be evaluated in relation to the intended application. Alternatively, the products may be evaluated to identify extensions or modifications to the product to meet the intended application requirements.
The following are aspects of functionality:

— process interface functions,
— logic processing functions to make decisions,
— data processing functions required to achieve the system mission,
— communications functions between modules or to other systems,
— human interface functions,
— interface functions to operate plant systems or alarms or to provide display information.

Characteristics, or subelements, of functionality include:

— flexibility, including:
  — configurability,
  — programmability,
  — expandability,
  — segmentation,
  — standardization;
— functional coverage;
— functional capacity.

Performance

Performance describes the extent to which the functions provided can be executed under defined operational and environmental conditions. This element should also include characteristics required if a system or its components fail. The required operational conditions should be determined and compared to those designed for, or experienced in use of, the product. The consideration of operational history may be important to performance evaluation and specification.

Performance factors include:

— accuracy (zero and full scale, linearity),
— stability (drift and repeatability),
— response time in normal and loaded conditions,
— consistency of behaviour,
— any requirement for equipment qualification, to show suitable performance of all functions in the worst expected environment.

Dependability

Dependability describes the extent to which the system can be relied upon to perform its intended functions under defined operational and environmental conditions.
Applicable factors include:

— availability (reliability and maintainability),
— probability of failure on demand,
— frequency of spurious actuation,
— assessment work needed in association with probabilistic safety assessment studies,
— robustness, changeover and standby features,
— safety integrity level (hardware and software),
— features needed to withstand common cause failures,
— security against accidental or malicious actions.

Proven operational history will be relevant to dependability.

Operability

Operability describes the features of a system which allow it to be used simply and effectively and kept in use easily and without degradation. Operability depends on several factors, including procedures for access to and entry of information and data into the system, the extent of information obtained by a single user request, the information formats used, and the interface devices used. Operability is strongly influenced by human factors engineering.

Characteristics of operability include:

— actions required on failure of modules, such as:
  — system hot or warm changeover,
  — adoption of a predefined state at failure;
— facilities for modification;
— facilities for operation and maintenance;
— system safe operation during maintenance and testing;
— on-line testing.

System operational history is relevant to product evaluation.

Safety characteristics

Safety characteristics are those describing the extent to which the system will not by itself impose potentially hazardous conditions on the nuclear power plant, personnel or the environment. Safety characteristics of the system include:

— personnel safety,
— process safety.
— defence in depth features,
— the safety of the system itself.

The safety characteristics of a monitoring, control and safety system depend on the inherent safety of the system, for example the use of fail-safe engineering principles, and the use of suitable suppression systems to prevent external effects. If a system is required for a role important to safety in a nuclear power plant, the system requirements will be mainly determined by its functionality, dependability and operability, which define its capability to provide its safety function.

2.2.2.3. Organizational issues

The culture and working environment established in an organization have an important influence on the adequacy and effectiveness of its software quality assurance programme. No matter how well the software quality assurance programme is designed, constructed and documented, it can only be effective if the organization makes effective use of the established management system.

An effective and adequate quality assurance programme is needed to manage and operate suitable support systems to maintain version control, document control, configuration management and change control. Those performing the technical work of software development or using computer programs should be able to rely on such support systems and the knowledge and expertise of the staff operating these systems to achieve these essential support functions, so that they are free to concentrate on the performance of their technical tasks.

A number of organizational issues contribute to the effectiveness of the quality assurance programme:

— It is important to define clearly the organizational interfaces, both internal and external to each group. Such an environment is essential to control the flow of information between software designers, programmers, reviewers, testers, staff of regulatory agencies, quality assurance staff and end users of the computer programs, in order to ensure that current and relevant information is passed between these parties. The adequate definition of interfaces is achieved through a combination of means, including establishing the roles of various groups, establishing the types of documents and data to be transferred across organizational interfaces and the required content of those documents and data, and methods to transfer information across interfaces and to indicate its receipt. A possible procedure for controlling interfaces is outlined in Appendix II.
— A clear understanding of the roles and responsibilities of key functions in an organization is important for an effective quality assurance programme.
Managers have ultimate responsibility for the definition, effectiveness and adequacy of the quality assurance programme. They are responsible for approving and directing the implementation of the software quality assurance programme. Managers control the allocation of both human and technical resources. They determine what facilities and tools are available. They determine the schedule targets within which certain tasks are to be performed. By their behaviour they establish the level of expectation for the degree of compliance with defined and proven company processes.

Those performing the work (designers, programmers, testers, analysts, software maintainers, etc.) have a responsibility to carry out their tasks correctly and according to proven and documented work processes within the framework of the software quality assurance programme. They apply their skills and expertise within this organizational setting to produce a software product of the required quality that has the necessary functional capabilities.

— It is important that line managers have appropriate training and an understanding of the purpose of a software quality assurance programme. They should recognize that the software quality assurance programme is their tool to manage and control the execution of the work. Line managers should be able to define and implement elements of the software quality assurance programme applicable to that portion of the work that they are responsible for.

— The organization should have competent and knowledgeable staff to perform the design, development, coding, testing and qualification of software and to operate the needed support systems. Staff should have the necessary basic scientific and technical qualifications. They should have access to appropriate training to be familiar with the organization’s support systems, methods and work practices. They should maintain an awareness of current technology and its use in performing their work.

— The organization should have quality assurance specialists who are knowledgeable in the relevant aspects of the requirements and implementation of a software quality assurance programme. Such specialists are expected to advise line management on quality issues and make recommendations about the appropriate elements of the software quality assurance programme. The quality assurance staff should be familiar with the purpose and nature of the work processes in order to perform independent assessments on behalf of line management of the adequacy and effectiveness of the software quality assurance programme.

2.2.2.4. Legal issues

The evolution in some countries of a ‘law of product liability’ can have far reaching effects on any organization operating in the nuclear industry, where a
supplier of technical systems, products or components is subject to legal action for injuries, death or economic losses due to errors in its products. Some of the steps available to organizations to address such liability concerns include:

— Implementation of an effective and adequate quality assurance programme. Such a quality assurance programme is perhaps the single most important and practical tool for exercising and demonstrating the exercise of due diligence.

— Agreements with regulatory authorities about the adequacy of the selected software quality assurance standards against which the software quality assurance programmes will be evaluated.

— Agreements with regulatory authorities regarding any licensing requirements that are additional to those specified in accepted software quality assurance standards.

— Agreements with suppliers of proprietary software regarding the information to be supplied with the software, particularly as it applies to the quality assurance programmes implemented by such suppliers. Special consideration should be given to suppliers of 'off the shelf' software who are not likely to change their development processes significantly for a nuclear application. In such circumstances, assessments should be made of the adequacy of the processes used and of the suitability of the resultant software for nuclear applications.

Some countries make the final legal liability for nuclear incidents the responsibility of the State. The State then needs assurance that the nuclear installation is safe, so that it can take on the liability. This involves the state regulating body gaining suitably high assurance that the plant is safe. Therefore, when safety system or safety related system software, as well as software for the design and analysis of safety or safety related systems and components, is involved, the State regulating body is likely to seek the highest practicable assurance of its quality. Similar steps should be used to achieve this assurance as described above.

2.2.2.5. Practical approaches to software quality assurance programme planning and implementation

Software is a product like many others, whose development and use need to be managed in order to achieve predictable and repeatable results. At the same time, many organizations either developing or using software also perform other work that does not involve software. Therefore, the software quality assurance programme should be integrated with the quality assurance programmes applicable to the organization's other activities.
Any software quality assurance programme consists of three general components:

— The overall management system, usually described in a quality assurance manual;
— Procedures and work instructions which specify how individual work processes are to be performed;
— Infrastructure tools and facilities to support the implementation of the management system; such tools include facilities for document control and records management, software configuration management, verification and validation testing tools, etc.

There is no single correct way to plan and implement a software quality assurance programme. The specific path taken depends a great deal on factors such as the organization's past experience, the character and qualifications of its staff, the nature of the software products to be developed or used, the regulatory environment and the nature of the organization’s other work activities. However, some general principles should be considered in planning and implementing a software quality assurance programme.

— Take advantage of quality assurance programme elements that already exist in the organization to perform other work, such as records management systems, audit function, quality assurance programme review function, and qualification and training function. In other words, incorporate the software quality assurance programme elements into the structure of any existing quality assurance programmes.
— Identify an appropriate software quality assurance standard against which the adequacy and completeness of any software quality assurance programme can be assessed. The selection of such a standard should take into account the nature of the organization’s activities involving software, the customer’s requirements and regulatory requirements.
— Be clear about the roles played by members of units performing the work and those with quality assurance expertise. It is essential that the representatives of performing units take an active part in the definition, documentation and implementation of any software quality assurance programme. They should prepare the elements describing the software quality assurance programme, including any quality assurance manuals and procedures. It is they who have to execute the work according to the adopted procedures. It is they who have to make the management system work as represented by the software quality assurance programme. The role of the quality assurance experts should be to assist in the interpretation of software quality assurance standards, and to
provide information and advice in the development and implementation of the software quality assurance programme elements by assisting in the preparation of quality assurance manuals and related procedures, by organizing and participating in training sessions, and by reviewing quality assurance programme documentation for clarity, compliance with standards and consistency with the organization's documentation practices. Members of the quality assurance function should not be put in the position of having to document the software quality assurance programme without input from the performing units. Such an approach can lead to inappropriate instructions and the perception by the performing units of unwanted and unwarranted restraints being imposed, and usually leads to a failure of the software quality assurance programme to be adopted by the performing units.

— Involve management in the process of developing and implementing a software quality assurance programme. The managers of the performing units are responsible for directing the work under the software quality assurance programme. They have to authorize the funds and other resources to perform the work according to the established practices. Therefore, they have to understand the reasons for the provisions of the software quality assurance programme and should actively support the implementation of the quality assurance programme.

— Establish a plan for developing and implementing a software quality assurance programme. Establishing a software quality assurance programme is like a journey. To reach the desired goal, a suitable road map should be used. The implementation plan should be designed to facilitate the smooth transition to the state of a fully functioning software quality assurance programme. It should identify the documents to be produced, the workshops and training sessions to be held, the sequence to be followed in the software quality assurance programme development and implementation, and the strategy to be followed in declaring elements of the software quality assurance programme operational.

— Establish the order in which documents describing a software quality assurance programme will be developed. Often organizations choose to prepare the quality assurance manual first, followed by supporting procedures. This top-down approach has intuitive appeal because it allows the quality assurance programme development to proceed from a general overview described in the quality assurance manual to more detailed instructions. The assumption underlying this approach is that the complete software quality assurance programme can be presented to the organization and implemented in one step. Practical experience suggests that such an approach is often not the most appropriate. Different elements of the software quality assurance programme are required at different times depending on the nature of the work to be performed. It is often difficult to prepare the overview of the software quality
assurance programme in the manual without having defined the details of the work processes that the overview is supposed to describe. Therefore, an approach to the documentation of the software quality assurance programme that is often more effective is one where the most urgently needed procedures are written and implemented first, followed by the preparation of the related quality assurance manual sections, followed by the second most urgent set of procedures, and so on. This allows for work to be performed under appropriate controls, while the development of the remaining software quality assurance programme elements proceeds in parallel.

- Take into account feedback from operating experience with the existing software quality assurance programme and with the software developed, maintained and used under this programme. Information about the effectiveness of the software quality assurance programme is available from sources such as customer feedback, the auditing process and regular management reviews of the software quality assurance programme. This information should be evaluated for opportunities to improve the software quality assurance programme’s completeness, effectiveness, efficiency and economy. Information about errors in the specification, design, coding and operation of software should not only be evaluated to remove the error; consideration should also be given to what changes are necessary in the software quality assurance programme to prevent such errors from recurring.

- Take into account the availability of new technology. As more effective tools become available, such as more powerful computer hardware, more sophisticated high level programming languages, configuration management tools and CASE\(^\text{22}\) tools, the work practices that are used during design, development, maintenance and use or operation of software will change. The software quality assurance programme should be maintained current to reflect how the organization has adopted such new technology.

- Use the software quality assurance programme as a framework for performing work involving software. Work should be authorized by assigning an approved scope, budget and schedule. For projects of any degree of complexity, the work should be divided into discrete work packages of a size that can be managed and controlled by the responsible organizational unit. The system used for defining the work breakdown structure should provide for the unique identification of work packages, scheduled completion dates, budgets and input/output requirements. Overall project planning tools, such as Gantt charts, should be used to indicate the relationships between various work packages and should show the progress made as the work proceeds.

\(^{22}\) CASE tools — computer assisted software engineering tools.
The environment in which organizations operate does not stay static. The structure of the organization and its work assignments change. The technology available to perform and manage the work changes. Governing technical and quality assurance standards change. Regulatory requirements are revised.

Under such conditions a static quality assurance programme would soon become a hindrance to the achievement of quality. The software quality assurance programme should be able to adapt to new circumstances while maintaining its ability to control the work effectively so that the organization can provide products and services of the required quality. This requires controlled and orderly processes to effect changes to the software quality assurance programme.

Most quality assurance standards, including those aimed at software, recognize this need for orderly change processes. Some of the ways in which such standards encourage orderly change include requirements such as:

— Defining processes for the orderly revision and approval of the organization's quality assurance manual; changes to the manual should be documented by those most familiar with the change in the organization, work process or infrastructure system. Management should actively participate in the review and approval of such changes to ensure that they reflect their intentions for directing the organization and how work is managed.

— Conducting regular management assessments of the organization's software quality assurance programme. Such reviews should lead to specific action plans for changes to the software quality assurance programme.

— Identifying deficiencies and weaknesses in the software quality assurance programme during independent assessments, which should lead to specific corrective actions to improve the software quality assurance programme.

— Implementing a system for receiving, evaluating and implementing staff suggestions for changes to the software quality assurance programme.

— Implementing a system for receiving, evaluating and responding to customer and operational experience feedback.

— Implementing a system for addressing the observations and recommendations resulting from assessments performed by customers and regulatory agencies.

— Providing for a smooth transition, during organizational restructuring, between the quality assurance programmes and infrastructure systems of the original and the new organization.

Such mechanisms encourage continuous adaptation of the quality assurance programme to changing circumstances and requirements in an orderly way that is prompt and effective.
2.3. INFRASTRUCTURE SYSTEMS

2.3.1. General

Infrastructure systems include systems for achieving software configuration management, software version identification, software change control, document control and computerized tools to perform some of these functions. While such systems can be based on paper records, the use of computer based tools to perform these functions is becoming widespread. Well designed computerized tools can automate many of the functions needed.

It is essential that the work processes and the procedures describing these processes be compatible with the characteristics of the infrastructure systems. The procedures should be designed to let the working staff interact smoothly with the infrastructure systems, where information, data and documents have to be exchanged with the infrastructure systems.

2.3.2. Assessment, selection and implementation of computerized tools

Any computerized tools used to support infrastructure systems and any CASE tools should be subject to a suitable form of assessment to ensure that they meet the organization's requirements and can perform the intended functions. The effectiveness of any assessments is enhanced if the organization establishes functional specifications for such tools which capture the essential capabilities and features of the tools.

The selection of appropriate computerized tools should follow a systematic process. Such a process is outlined below.

During the initiation process for the evaluation and selection of computerized tools, the general objectives and requirements of the intended evaluation and selection of the tool should be defined to establish the high level direction and to establish the management aspects of the effort such as schedule, resources and cost. The initiation process typically consists of three phases:

— goal setting: provides the rationale and general policy for the evaluation and selection of the tool;
— establishing selection criteria: provides criteria to be used in the subsequent selection process;
— project planning: results in a plan which includes generic planning information and also information which defines the structure of the evaluation and selection effort.

The structuring process elaborates a set of requirements based on the tool characteristics against which such tools should be evaluated and also the necessary
information on the tools to permit evaluation. The structuring process typically includes three main activities:

— requirements analysis: transforms organizational needs into measurable structures;
— tool information gathering: captures a snapshot of the current state of the art in tools;
— identifying final candidate tools: candidate tools are identified for evaluation using the results of the previous two activities.

The evaluation process leads to technical evaluation reports that form the main input to the selection process. Each evaluation captures a profile of the quality and other characteristics of the tool. Typically, three activities are involved in such evaluations:

— preparation for evaluation: finalization of the various details of the evaluation in an evaluation plan;
— evaluating the tools: measurements, ratings and assessments;
— evaluation reporting: an evaluation report is prepared which provides the results of the evaluation for each tool under consideration.

During the selection process the most suitable tool is identified from the candidates under consideration. The selection process should ensure that the recommended tools meet the original goals. The selection process should compare the results of the evaluations of the candidate tools to determine which is the most appropriate for selection. The selection process typically consists of four activities:

— preparing for selection: the selection criteria are finalized and the selection algorithm is defined;
— assessing the evaluation results: the selection algorithm is applied to the evaluation results;
— recommending a selection decision: the best of the candidates is determined;
— validating the selection decision: the recommended selection is validated against the original goals.

The implementation process should be carried out in two basic steps. The selected computerized tools should be implemented on a trial basis in one area of the organization to confirm in the field that they function as intended and to gain experience in their use. After such a trial period, such systems can then be applied across the whole organization, knowing that implementation problems will be minimized.
Detailed information about the elements of an appropriate selection process are available in publications such as ISO/IEC 14102 [14]. Standard IEEE 1209 [15] also provides useful information.

Additional information about the acquisition of computerized tools is given in Appendix III.

2.3.3. Considerations before acquisition of computerized tools

2.3.3.1. Types of computerized tools

The types of computerized tools used in the management of software can be collected into four broad groups. They include tools to assist in:

— computer program understanding and reverse engineering;
— testing;
— configuration management;
— documentation and measurement.

2.3.3.2. Computer program understanding and reverse engineering tools

Computer program understanding involves having a general knowledge of what a computer program does and how it relates to its environment, identifying where in the system changes are to be effected, and knowing how components work that are to be modified. Reverse engineering permits analysis and different representations of the computer program to promote that understanding. The majority of tools in this category assist programmers to form a mental model of the computer programs under examination. Examples include the program slicer, static analyser, dynamic analyser and cross-referencer.

A brief description of the functions of these tools is given in Appendix IV.

2.3.3.3. Configuration management tools

Effective configuration management is not possible without some kind of automated support tools in most environments. Keeping track of modifications to a computer program involves very large numbers of files and associated documentation. Configuration management and version control tools act as a repository for the components that make up the software system.
2.4. TRAINING AND QUALIFICATION

It is a matter of long standing experience that quality work requires competent, knowledgeable and qualified staff. Quality assurance standards reflect this fact by containing some general requirement that staff be appropriately qualified and trained.

Usually, qualification and training are associated with technical and professional qualifications, such as post-secondary education and participation in conferences and seminars in order to keep up to date with technical advances. Such training is essential to achieve and maintain the technical competence of the organization. However, additional training is needed to introduce members of the organization to the requirements of the quality assurance programme. Such information is not normally conveyed during technical training at the post-secondary-school level and in technical seminars and conferences.

Training in quality assurance topics is usually provided as part of an organization's staff development programme. The nature and content of such training programmes are generally unique to the organization, given the nature of its work, its past history, the attitudes of its management and the mix of its staff. Some general information on quality assurance training for organizations dealing with software is provided in Appendix V.

Some examples of training programme outlines are given in Appendix VI.

Reference [2], Section 14, and the references quoted therein provide additional information on qualification and training of personnel.

2.5. NON-CONFORMANCE CONTROL AND CORRECTIVE ACTIONS

Software development generates non-conformances and related corrective actions in the same way as other development processes within the organization. In most cases, non-conformance control and corrective actions can be carried out under the organization's general quality assurance programme. Procedures and their supporting infrastructure systems should be available to:

— detect, record and investigate the causes of non-conformances;
— define, apply and follow-up on effective corrective actions;
— initiate preventive actions to deal with potential problems to a level corresponding to the risks encountered.

Also, the systematic application of the cycle of non-conformances and corrective actions is the basis for process improvement within organizations. These topics are further elaborated below. Other applicable comments and sample records can be found in Ref. [4].
2.5.1. Estimating the number of software errors

A traditional manufacturing quality assurance programme includes steps to prevent as much as possible and identify and remove errors at intermediate stages of the manufacturing process. This is achieved through a variety of inspections and dimensional checks. The same principle applies to the development ('the manufacture') of software. Clearly the traditional manufacturing tools to prevent, or identify and remove errors do not apply to software.

Estimates of software errors\textsuperscript{23} should have a common basis in order to make meaningful comparisons between various estimates. Therefore, a common definition would be useful for what represents a software error. A simple definition is that it is a deviation from specification.

The desired result of a computer program may not be achieved despite the fact that the software fully complies with its specification. This should be considered an error in the specification and not in the software.

Errors should be distinguished from software failures\textsuperscript{24}. A software failure can be defined as the departure of the external results of computer program operation from requirements. A software failure can be defined more precisely as a software error that causes a deviation from the required output by more than a specified tolerance. Also, the software should produce correct outputs only for inputs that are within specified limits.

Failures resulting from errors in systems external to the software under consideration should not be counted.

An important step towards establishing methods for preventing or identifying and removing errors is to estimate the likely number of errors to be expected. Errors may occur in requirements specifications, design or computer code. Current industry experience suggests that the number of errors in software can range from 30 to 95 errors per thousand lines of code. Most of these errors are removed through self-checking, design reviews, walkthroughs\textsuperscript{25}, inspections, module testing and integration testing, with a typical removal efficiency of 85%.

In software development we are concerned with another element — an error which covers a broader range than just a fault in the code. Errors may occur in requirements, specifications, design or code. Developers find these errors by self-checking, walkthroughs, inspections, module testing, etc.

\textsuperscript{23} Error — the difference between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition.

\textsuperscript{24} Failure — the inability of a system or component to perform its required functions within specified performance requirements.

\textsuperscript{25} Walkthrough — a review process in which a designer or programmer leads one or more members of a review team through a computer program or its design, or portions thereof.
The following error types can be defined:

- requirements errors
- design errors
- algorithmic processing errors
- interface errors
- performance errors
- documentation errors.

2.5.2. **Suggested criteria for categories of software defects**

In computer based systems important to safety in nuclear power plants, software which does not perform as expected should be subject to a report. But the 'defects' reported by verification and validation, the test engineers or the plant may be due to misunderstanding of the system operation or the design. They may be due to associated hardware defects or due to setting to work activities, and therefore not be true defects. But some will be defects causing faults which require analysis and correction. As a term suitable to cover both real and potentially unreal defects, they may be referred to as 'findings'.

A number of findings in the software documents and code may be of direct safety concern. This is so only if a finding is in fact a defect, and degrades the safety functions of the system. Some findings may be defects which degrade the software functions, or degrade auxiliary functions or the quality record, but with no direct functional significance to safety. Some findings may be defects of no real quality or functional significance, and concern only minor matters such as comments in the code. Some findings, on investigation, may be misunderstandings arising during the review or analysis process, and should be discounted as not relevant, although they will remain on the records of the project.

The total number of defects known at some point in time may have no direct functional significance if the system is still under development and not in service. Some projects may use the number of defects in reliability growth models to forecast a time of removal of significant defects. A real defect of function or of quality control revealed by verification and validation or some site incident is of great concern. In a well managed project, real defects will be very few.

It is important to prevent all such findings and defects being lumped together without differentiation. A generic classification of errors in terms of their severity may be as follows:

---

26 Defect — see definition for 'error' (see footnote 23).
27 Code — in software engineering, computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler or other translator.
(a) Critical — prevents further execution; non-recoverable. Must be fixed before program is used again.
(b) Serious — subsequent answers grossly wrong or performance substantially degraded. User could continue operating only if the poor results are allowed for. Should be fixed soon.
(c) Moderate — execution continues, but behaviour only partially correct. Should be fixed in this release.
(d) Cosmetic — tolerable or deferrable, such as errors in format of displays or printouts. Should be fixed for appearance reasons, but fix may be delayed until convenient.

Errors should be classified in order to aid in the analysis of causes.
All findings should be reported and analysed after some chosen point in the software life-cycle, such as start of final acceptance test, or start of on-line operation. There should be a formal response to each finding, placing it into a class depending on its significance. The classes could be described as follows.

A Class 1 defect is one which either:

(i) Directly prevents a function important to safety from operating;
(ii) Directly causes a function important to safety to operate when not required;
(iii) Directly reduces the system reliability.

A Class 2 defect is one which is not Class 1 and either:

(i) Reduces the possible reliability of a required function important to safety, by degrading a support, maintenance, test or calibration function;
(ii) Prevents information transfer or provides incorrect information transfer to an external system of lower safety category;
(iii) Is software information, configuration information or software issue level and is incorrect, such that interfacing, modifications or revisions would be incorrect, but without other on-line implications;
(iv) Implements a function in code or appears as documentation significantly below the agreed standard for the system concerned;
(v) Implements a function in a manner which could fail in expected abnormal hardware conditions, but does not fall into Class 1.

A Class 3 defect is one which is not Class 1 or 2 and either:

(i) Could significantly impair documentation clarity or quality, where no functional defect exists;
(ii) Could impair clarity of auxiliary information on performance, operation or function;
(iii) Could impair auxiliary aspects not central to the functions important to safety.

If the finding does not fall into the above classes, then it is identified as not relevant.

2.5.3. **Software development process improvement**

The organization developing software should have in place as part of its software quality assurance programme processes to identify and implement improvements to its software development processes. An example of such a continuous improvement process is the defect prevention process. It is based on three steps:

1. Analyse existing defects and errors to trace the root causes.
2. Suggest preventive actions to eliminate the defect root causes.
3. Implement the preventive actions.

The defect prevention process is a real time process, integrated into every stage of the development process. Compared to ‘post-mortems’, this approach helps ensure that meaningful discussion takes place when things are fresh in everyone’s mind. It focuses on defect related actions and process oriented preventive actions. The defect prevention process provides a systematic, objective, data based mechanism for action implementation. It is a bottom-up approach. The implementation of the defect prevention process can reduce error rates significantly. It is repeated in Ref. [16] that IBM, at one of their facilities, achieved a 54% reduction in error injection during development and a reduction of 60% in field defects on one software product, after a defect prevention process was implemented.

The defect prevention process can be applied to any development process, whether waterfall, prototyping, iterative, spiral, cleanroom or others.

Appendix VII provides additional information about the defect prevention process.

2.6. **SOFTWARE QUALITY ASSURANCE PROGRAMME DOCUMENTATION**

Generally accepted practice for documenting any quality assurance programme is to structure the documents in three main tiers. The most senior document is the quality assurance manual. Further details about individual work practices and processes are given in procedures, which form the second tier of the quality assurance
programme documentation. The third tier consists of detailed work instructions giving directions how to operate specific pieces of equipment, fill in a specific form, operate an administrative system, etc. The same approach is recommended for the documentation of a software quality assurance programme.

The quality assurance manual is the top tier document. The manual should be brief and concise. The main body of the manual should not exceed 25–35 pages. The manual should be designed to provide an overview of the organization’s software quality assurance programme, including a description of its organization and the principal roles and responsibilities within the organization. The manual should also serve as a road map leading the reader to relevant second and third tier documentation that is needed to perform work according to established procedures.

The manual should address all the elements of the applicable software quality assurance standard. The manual need not get into details of how the organization has addressed the quality assurance standard requirements. But the manual should demonstrate that the organization has properly addressed all the software quality assurance standard requirements. It is recommended that the manual be structured such that its sections correspond to the clauses in the applicable quality assurance standard. This makes it easy for any reviewer to recognize the degree to which the software quality assurance programme complies with all the requirements of the governing quality assurance standard. It is also recommended that the manual contain a cross-reference table, perhaps as an appendix, which links the clauses in the governing quality assurance standard, the sections in the manual and the applicable procedures and work instructions. A possible layout is shown in Table I.

TABLE I. CROSS-REFERENCE TABLE: A PROPOSED LAYOUT WITH TYPICAL ENTRIES

<table>
<thead>
<tr>
<th>QA standard clause No.</th>
<th>QA manual section No.</th>
<th>Procedural publication title</th>
<th>Procedural document identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Identification and assessment of personnel qualifications</td>
<td>00–261.1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Computer program design and development</td>
<td>00–451.1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Configuration management for development, maintenance and use of design, engineering and analysis software</td>
<td>00–552.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Change control for development, maintenance and use of design, engineering and analysis software</td>
<td>00–551.1</td>
</tr>
<tr>
<td>etc.</td>
<td>etc.</td>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>

35
The second tier procedures should provide instructions on how to perform a
certain process, such as the preparation, review, approval, distribution and revision of
a design document. Individual procedures should not exceed 5–6 pages. The
procedures should form part of a procedural system which assigns unique identities
to each procedure. The structure of the procedure system should be such that it
complements the structure of the quality assurance manual. The documentation of the
second tier procedures should follow a standard format, including recommended
headings. The language should be simple and concise. Long explanations about the
reasons for a particular process should not be included in a procedure. Information in
procedures should focus on how an activity is to be carried out and by whom. The
main body of a procedure should be the description of a work process. Two formats
are recommended. The process can be described in the form of a flow chart. Alternatively, the process can be documented in textual form using the play-script format28. In either format the participants in a process, their actions and the sequence
of steps are clearly identified.

The third tier of instructions is represented by all those documents that provide
detailed technical directions for the execution of specific tasks. Instructions usually
supplement information contained in second tier procedures. Instructions can provide
more detailed information regarding a particular aspect of an administrative
procedure. In this case such instructions should follow a format that is similar to that
for second tier procedures.

Instructions can also be in the form of detailed manuals, such as programmers' 
handbooks describing recommended programming practices, or users’ manuals for
operating test facilities. No specific recommendations can be made regarding the
format and structure of such manuals. Some general recommendations include:

— assign unique document identities for reference;
— implement a system of revision control;
— structure the manual so that individual topics are easily found under descriptive
  headings;
— use simple sentence structure;
— avoid long explanations of why a certain step is necessary;
— include a table of contents and an index to facilitate finding information on a
  specific topic easily.

Reference [10] provides additional information.

28 ‘Play-script’ format refers to a style of specifying work process steps sequentially
and identifying those required to carry them out that is similar to the way the dialogue is
recorded for a theatre play, where specific actors are assigned to speak specific words and are
directed to take specific actions.
3. PERFORMANCE

3.1. WORK PLANNING

3.1.1. Life-cycle phases

Although some professionals working in software development still express opinions to the contrary, software should be considered an engineered product like many other products used in industry and the service sector and by consumers. Engineered products have defined characteristics, features and functions. They are the result of a series of controlled processes and operations that ensure a predictable outcome leading to the desired product or service. Components of the product enter any particular stage of such a series of processes, they change state in a controlled manner as a result of the operations performed at that stage, and then move on to the next stage.

A life-cycle model for software is a structured description of the activities, products and controls from software conception to retirement from use. Life-cycle models now have widespread use in software engineering. According to modern practice, management, development, independent verification and validation, and quality assurance plans for software important to safety are life-cycle based. The establishment of a life-cycle model for a software project is necessary so that work can be ordered into stages, each with tangible output materials and documents which can be subjected to controls. References [2–4] should be consulted for basic descriptions of each life-cycle phase.

The selection of a particular life-cycle model is done at the beginning of a project, and depends on the software type, the project context, and the software and hardware development environment. The relation between software type and life-cycle stages is illustrated by considering custom developed software for monitoring, control and safety systems and for design, engineering and analysis applications.

Typical life-cycle phases for custom developed monitoring, control and safety system software include:

- system requirements specification
- computer system requirements specification
- software requirements/hardware requirements
- software design plan
- computer system design/prototyping
- software design/hardware design
- software module implementation/hardware module fabrication
- software integration/hardware integration
— computer system integration/instrumentation and control or testing equipment installation
— system integration and testing
— system commissioning
— system operation and maintenance
— software retirement from use.

A typical life-cycle for monitoring, control and safety system software is illustrated in Fig. 3.

For monitoring, control and safety system software it is important to recognize that the software life-cycle is a part of a larger system life-cycle, especially after the software is installed as part of the nuclear power plant system. For example, during the commissioning, operation and maintenance of a monitoring, control or safety system, activities such as modifications to the system hardware and software will have interactions that have to be controlled so that the hardware and software continue to operate as intended.

For monitoring, control and safety system software it is important to recognize that the software life-cycle is a part of a larger system life-cycle, especially after the

![Diagram of I&C system life-cycle](image-url)
software is installed as part of the nuclear power plant system. For example, during the commissioning, operation and maintenance of a monitoring, control or safety system, activities such as modifications to the system hardware and software will have interactions that have to be controlled so that the hardware and software continue to operate as intended.

Typical life-cycle phases for custom developed design, engineering and analysis software include:

- application requirements
- computer system requirements specification
- software requirements/hardware requirements
- software design/hardware selection
- software module implementation/hardware procurement
- software integration/hardware installation and commissioning
- software installation
- software testing
- software commissioning
- software and hardware operation, upgrades and maintenance
- software retirement from use.

In the case of pre-developed software, the type and scope of each stage will change according to the extent of use of pre-developed base software and the pre-existence of the hardware platform. A suitable life-cycle could be defined in each case, which in general terms could clearly separate verifiable work stages.

In order to understand and describe the stages and processes involved in the development, maintenance and modification of software, several models have been proposed to describe the stages and processes that are used to create, maintain and modify software. Appendix VIII describes some of these proposed models. They are discussed here to indicate various ways of perceiving the stages and processes that a software product passes until it reaches operational use. This discussion should not be interpreted as indicating a preference for any one of these models.

3.1.2. Possible development process models for computer programs

Early experience with developing software has been that the results were unpredictable in terms of the capabilities and reliability of the software, the actual versus the planned completion of the development, and the cost of developing the software. Today, there is a general consensus that such results are due to a lack of understanding of the processes used to develop software and therefore a lack of control of such processes.
This guide does not make any specific recommendations in favour of one or another development process model. Information is presented here to make the reader aware of several models and their characteristics. The choice of an appropriate development process model depends on the nature of the development task, the structure and experience of the development organization, and the technical resources available.

More detailed discussions about software development life-cycle models can be found in the published literature. One example is Ref. [16]. The standards IEC 880 and IEEE 1074 also discuss life-cycle processes. A brief review of some common software development life-cycle models is given in Appendix VIII.

3.1.3. Choice of development life-cycle models in relation to importance to safety

Specific recommendations for the selection of a particular development life-cycle model are difficult to state. Much depends on such factors as the nature of the organization, the level of skill and experience of its staff, and the infrastructure systems (configuration management systems, records management systems, planning and resource loading tools, available CASE tools, etc.).

The selection of a development life-cycle should consider the following general guidelines:

(a) For monitoring, control and safety system software, and for design, engineering and analysis software that requires a high degree of reliability to achieve the required level of safety, it is important to achieve early in the development process as complete a definition of the software requirements as possible. Therefore, this consideration would favour development process models like the waterfall and spiral model.

(b) It may be necessary to explore possible software design approaches before formal development begins. In these circumstances it may be valuable to develop a limited number of key modules quickly on a trial basis to help determine whether a particular design approach is feasible. Under such circumstances the use of a prototyping approach or iterative model may be suitable. Once a suitable design approach has been found, the formal software development should be based on a development life-cycle model that requires the formal definition of a complete set of requirements, such as the waterfall or spiral models.

29 The figures in this publication illustrating various software development life-cycle models are based on those in Ref. [16].

30 Configuration management — the process of identifying software components, controlling changes, and maintaining the integrity and traceability of the configuration.
(c) If the organization has staff with extensive skills and experience and the
development task is small, the organization may consider using a cleanroom
approach (see Appendix VIII).

3.1.4. Post-development life-cycle phases

The life-cycle models discussed above focus on the development process used
to create new software. These models do not discuss the phases dealing with
commissioning, operation, maintenance and retirement from use, referred to here as
the operating life-cycle phases.

The commissioning process for monitoring, control and safety system software
involves essentially a set of trials to determine whether the integrated hardware and
software system behaves as specified in the system design. Such tests may identify
latent software defects that need to be corrected. The required software changes
should be developed and implemented in a controlled manner. The creation of these
software changes involves the application of design and development activities that
can be considered part of a development life-cycle. An appropriate software quality
assurance programme should be in place, based on a development life-cycle model,
that the organization making the changes is familiar with. A similar approach applies
to design, engineering and analysis software when such software is first introduced in
an organization and trial runs are carried out to confirm that the software behaves as
expected on the computer platforms on which it has been installed.

The normal operation of monitoring, control and safety system software and of
design, engineering and analysis software is a relatively stable period in the total
software life-cycle. The software functions within a stable environment that is
controlled under an appropriate quality assurance programme. Such a quality
assurance programme needs to ensure that the software is used correctly.

In the course of operating monitoring, control and safety system software and
design, engineering and analysis software, new requirements may be identified or
latent software defects may be detected. Remedial action involves changes to the
software. The design and development activities and their control should be similar to
those used to create software changes during the commissioning phase.

3.2. ANALYSIS, DESIGN AND IMPLEMENTATION

3.2.1. Design and development

The creation of a computer program is usually motivated by the need to satisfy
a set of functional requirements. Such functional requirements are complemented
with other requirements specifying the conditions under which the software has to
operate, such as the computer, size and memory limitations, operating speeds and programming language requirements.

The objective of the design phase is to produce a detailed description of the computer program design that demonstrates how the software requirements are addressed in that design. The design description documents should describe the design in sufficient detail so that no further refinements of the module\textsuperscript{31} structure, module interfaces, data structures\textsuperscript{32} or databases\textsuperscript{33} are required during subsequent phases of software development.

Software design consists of the set of activities that determine a specified set of requirements for a computer program to be implemented. The design process can be divided into two main phases, the high level design phase and the detailed level design phase. During the high level design, the main structure and organization of the software modules are proposed. The requirements are grouped and associated with a particular major computer program component. During the detailed design phase, a more detailed description is developed of the organization and structure of each major computer program component in terms of individual modules. The relationships between the modules are determined. The general structure of the computer program is established.

During the design phase, the thoughts and ideas about the structure and organization of the computer program are described in one or more documents, usually called ‘design descriptions’. Design descriptions describe the approach taken to translate the requirements into a functioning computer program using natural language text and pictorial representations such as flow charts. The documents describing the computer program design usually do not contain any coding.

Adequate documentation describing the design, structure and use of computer programs is an essential quality characteristic. There is a wide variety of documentation tools including data flow and control chart generators, requirements tracers and CASE tools.

The structure and content of the design documents should follow a standard format. Recommendations for the structure and content of various design documents have been published by standards writing organizations (see Bibliography). These should be consulted to develop a standard format and publication structure for the developer organization’s design documents.

\textsuperscript{31} Module — a discrete and identifiable part of a computer program that can be separately compiled and assembled.

\textsuperscript{32} Data structure — a representation of the logical relationships among individual data elements.

\textsuperscript{33} Database — a collection of interrelated data stored together in one or more computerized files.
Some design methods suggest trying several alternative designs before settling on one specific approach. Experience indicates that such an approach does not produce the most effective product. Good designs evolve, they do not appear by spontaneous generation in their final form.

Achieving a good design is essential to successful software development. Design can represent up to 60% of the total development effort. Therefore, it is important to develop methods to measure software design quality. Only with the introduction of measurements can software design move from an art to an engineering discipline. It is important to recognize that no single approach to measuring software quality can fit all possible software development methods and organizational environments. Each organization has to develop its own measurement programmes.

A good design is achieved if:

— there is a clear traceability\(^{34}\) from the software requirements to the design;
— it is easy to verify the requirements specifications against overall reactor system requirements for monitoring, control and safety system software;
— it easy to verify the requirements specifications against the nature of the analysis problem for design, engineering and analysis software;
— it is easy to verify individual design documents against the corresponding requirements specifications;
— it is easy to verify the correctness of the program details during testing against the applicable design publications;
— it is easy to implement the design accurately in a computer program;
— it is easy to maintain or modify the computer program during the operational life-cycle phase.

Design methods provide rules for system construction but generally do not identify corresponding measures of quality. Many design measures have evolved in association with design heuristics\(^ {35}\). Specifying quality objectives like modularity\(^ {36}\) and simplicity\(^ {37}\) does not ensure quality products unless more specific information is given. That information often takes the form of software standards and rules of thumb.

\(^{34}\) Traceability — characteristic showing there is a high probability of finding the source of information in the products of the preceding life-cycle phase.

\(^{35}\) Heuristics — rules for developing good systems.

\(^{36}\) Modularity — the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
Some common design heuristics for achieving modularity, considered to contribute to improved produceability, are:

— small modules
— limited data coupling
— medium span of control
— singleness of purpose.

The functional complexity\(^{37}\) of a software system is driven mainly by the nature of the problem to be solved. It is very difficult to control that complexity at the functional level. Therefore, the complexities of the output from the subsequent phases of the development process have to be minimized.

Recent studies show that system complexity effectively predicts the total error rate for development projects. Complexity has been found to account for over 60% of the variation in error rate.

Recent studies and empirical models suggest four strategies for minimizing design complexity:

— Optimize module complexity (keep the number of calls from a module to three or less).
— Minimize data variables\(^{38}\) (use common data and rearrange module call structures to minimize data complexity).
— Minimize connections (the number of calls from a module, the number of input/output variables, and the count of the number of modules should conform to similar S-shaped curves when the cumulative percentage of these parameters is plotted against the level of the modules in the computer program hierarchy).
— Increase design efficiency (modern programming practices should be used).

A detailed discussion on design heuristics is provided in Ref. [19].

Design reviews are a common tool used to ensure that the design meets specified requirements. Reviews compare a software product against predetermined criteria and identify actions to address identified problems. Such reviews bring together participants with the expertise to deal with assessment of the work in terms of satisfaction of requirements (traceability), ease of production (produceability) and conformance to standards. As part of an appropriate management system the quality assurance organization verifies compliance to standards, managers question budgets

\(^{37}\) Complexity — the degree to which a system or component has a design or implementation that is difficult to understand and verify.

\(^{38}\) Variable — a quantity that can assume any of a given set of values.
and schedules, and technical staff explain their approach to requirements satisfaction and produceability. The client explains his/her view of software requirements. Design reviews are a data collection and analysis process. During design, satisfaction of requirements can be demonstrated by showing traceability of requirements to design components.

Reviews and inspections should occur throughout the software life-cycle. They should be planned and systematic. Review results should be captured and documented in problem reports, corrective actions and budgets/schedule information. The information generated during design reviews should also be used to evaluate the design process and to plan for new projects.

A design and code walkthrough is another verification procedure. The purpose of walkthroughs is to detect and eliminate errors early in the development process. The key feature of such a formal review session with technical colleagues is that development team members present their work, explain the rationale behind the design decisions and simulate program operation for typical cases. The reviewers question the decisions, suggest improvements and point out errors and areas warranting improvement.

Programming standards help ensure satisfaction of requirements and ease of production in the completed design. Standard notation improves understandability and communication. The application of programming standards should be subject to a certain degree of balance. Concern for standards compliance should not be pursued to the detriment of the technical quality the standards were intended to promote. Development organizations should rely on a small set of key programming standards. Programming standards are a simple mechanism that can never substitute for technical understanding of the design process and application area.

An effective software quality assurance programme relies on measurements to determine whether a process is under control. During software design and development one such indicator is the error rate as a function of development phase. Measuring error rates during testing helps determine whether the software development process is under control. Within large projects, the error rate tends to stabilize over time. Plotting error rates on a control chart provides one mechanism for monitoring process performance.

The increasing use of design tools permits designs to be stored in a machine readable form like source code\(^\text{39}\). This makes design more amenable to analysis. Such tools strengthen the effectiveness of the software quality assurance programme since they facilitate the determination of quantitative measures for parameters such as error rate and produceability.

\(^{39}\) Source code — computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler or other translator.
As is true for other activities, tools and systems cannot totally overcome the influence of the human element. Experience shows that the total error rate as one measure of software quality is also affected by the level of experience of the development team. Less experienced programmers will produce higher error rates.

3.2.1.1. Monitoring, control and safety system software

For monitoring, control and safety system software ('real time' software), although the software engineering process is a distinct activity, it should not proceed in isolation from the engineering of the whole system including hardware, and selection and incorporation of pre-developed software. The software engineering process should be managed as an integral part of the overall system design process. A suitable software development plan (SDP) should be developed and followed, subject to necessary revisions as the project is executed.

General guidance for the preparation of SDPs can be found in IEEE standard 1058 [20]. Some recommendations for SDPs applied to the development of monitoring, control and safety system software are given in Appendix X.

The flowchart in Fig. 4 provides an overview of the typical design and development, installation and commissioning, and operation and maintenance phases for monitoring, control and safety system software.

Design and development practices

As a first step, documents should be identified that contain information about the engineering system, hardware, and pre-developed software requirements and design specifications. This material can be referred to collectively as design input documentation (DID) and represents an input to the software engineering process. To facilitate the software design and development process, the DID should satisfy certain criteria. General recommendations for the format and content of the DID are given in Appendix IX.

It is important to establish uniform and consistent conventions, work practices and procedures that all members of the development team can follow. This information should be available to all team members in a form that is readily accessible. Typically, this information is collected in a document referred to here as the standards and procedures handbook (SPH). The SPH may be made available to members of the development team in the form of one or more hard copy volumes of instructions. Where the facilities exist, the information can also be provided in electronic form using shared file directories or on the organization's internal intranet.

Some recommendations for the content of an SPH applied to the development of monitoring, control and safety system software are given in Appendix XI.
Software requirements specification

The software requirements specification (SRS) should fully specify the overall function to be performed by the software. The SRS should contain all relevant requirements from the DID and any other software specific requirements which arise as a result of the environment in which the software is expected to operate. The SRS should be written in terms of requirements and design constraints and should limit the range of valid design solutions without specifying a particular design.

General guidance for the preparation of SRSs can be found in IEEE standard 830 [21]. Some recommendations are given in Appendix XII for the content of software requirements specifications applicable to monitoring, control and safety system software.

Software design description

The objective of the software design process is to produce a detailed software design which satisfies the requirements in the SRS and provides the basis for the software programming activity. The output of the software design process is one or a set of documents which represent the design. This is the software design description (SDD). The SDD should describe the design to a sufficient level of detail so that no further refinement of the module structure, module interfaces, data structures or databases is required during software coding.

The primary purpose of the SDD is to guide the software coding and testing phases. It is advisable to revise the SDD after the software development and commissioning activities are complete in order to reflect the actual software in its operational state. The SDD in this form is an important document for future maintainers of the software system. It can also serve as a product baseline from which all future changes can be made to correct defects found during operation, and to add enhancements to the software as these become necessary.

General guidance for the preparation of SDDs can be found in IEEE standard 1016 [22]. Some recommendations for the content of SDDs applicable to monitoring, control and safety system software are given in Appendix XIII.

3.2.1.2. Design, engineering and analysis software

Computer program design and development

The design and development process begins typically with a recognition that a particular reactor system design or performance characteristic requires a solution using computer analysis. If a suitable computer program is not available as an analysis tool, one has to be developed.
The first step in this development process should be the description and documentation of the problem to be solved. This problem definition should provide as much detail about the problem as possible. The problem can be expressed in natural language format describing the physical systems and phenomena involved. Where possible, applicable mathematical models and relationships should be provided using conventional mathematical notations.
It is important that a clear software development plan (SDP) be created at the beginning of the development task as this will help in the allocation of time and resources. The SDP provides a description of all activities associated with the production of a new computer program.
The theory that forms the basis for the computer program should be documented. The solution methods that are to be employed in the program should be clearly stated in a theory manual along with the reasons for choosing the methods. Also, the physical or mathematical basis for the methods should be presented. The theory manual should be started during the design and development phase to serve as a design and development document. It should be completed after the design and development stage has come to an end to reflect the final versions of solution methods employed and to serve as a user's reference document.

The requirements specification describes the features and capabilities that the computer program is expected to have. It provides the reference against which the design and coding of the computer program are eventually evaluated.

Before any coding is undertaken, the design of the computer program should be documented in a design description in such a manner that the design can be clearly related to the requirements stated in the requirements specification.

The coding phase consists of translating the detailed design into computer language, debugging the resulting computer program and integrating computer program modules. Module coding and debugging should be performed so that each module performs the functions identified in the computer program design. Integration of the modules should be done to ensure that the integrated computer program performs as specified in the design description.

Good programming practices should be followed. This will ensure a consistent approach to the formulation of computer instructions and a minimum of difficulty in coding, debugging, testing and integrating the resultant computer program.

Additional recommendations are given in the following sections and in Appendices XIV and XV.

**Documentation requirements for design, engineering and analysis software**

Documentation for design, engineering and analysis software falls into two principal groups:

— design and development documents
— application documents.

The following design and development documentation is typical for design, engineering and analysis software:

— problem definition
— development plan
— theory manual
Documentation may be presented in one or more separate documents, as appropriate. Additional recommendations on design and development documentation are given in Appendix XIV.

Application documents for design, engineering and analysis software

The following application documents are typical for design, engineering and analysis software:

- computer program abstract
- theory manual
- user's manual
- validation report/manual
- version tracking record.

The application documents are intended to provide the program users with instructions to install, operate and manage the released computer program package. Additional recommendations on application documents are given in Appendix XV.

Configuration management

Scientific and analytical computer programs should be subject to a configuration management system. The configuration management system should have the capability to identify configuration components and to maintain the integrity and traceability of a particular set of configuration components.

Configuration components include:

- the source program
- system software components (such as operating system, compiler and library functions) and the executable computer program
- computer program documents.

Each version of a configuration should be uniquely identified using a defined naming convention. Any change to one or more configuration components constitutes a new configuration version.
Those configuration components which are required to recover any previous configuration version and are under the control of the organization should be preserved. Ageing of storage media and changing technologies should be taken into account.

The source program version should be identified both in the coding and in the computer program output.

Changes should be performed in accordance with a defined change control process making sure that any changed computer program modules are identified as such.

**Change control**

A system to control changes should be implemented. A change control system should provide that:

— reasons for changes are identified;
— the version to be modified is specified and a new version identification is given to the changed configuration;
— proposed changes are reviewed and approved;
— a change control plan (i.e. SDP for changes) is produced for significant changes;
— a software requirements specification is produced for significant changes;
— changes and their verification are documented in a version tracking record;
— the new version is archived and released for use in a controlled manner.

3.2.2. Coding

During the coding phase, the design is translated into computer language. The resulting code is debugged and integrated into computer program modules. During the coding phase, all necessary databases are also created, including any appropriate initial data. These databases either contain the data necessary to execute the computer program or represent the structures into which appropriate data are read during computer program execution.

---

40 Coding — translating a computer program design into a programming language.
41 Integration — the process of combining software components, hardware components, or both into an overall system.
In the coding of a computer program, adherence to a set of good programming practices is recommended. A possible set of good coding principles is presented in Appendix XVI.

3.2.2.1. Programming monitoring, control and safety system software

The SDD is the input to the process which converts the design into a computer program. A detailed discussion on programming issues, experience and current practices is given in Ref. [2]. The standard IEC 880 [17] provides useful recommendations regarding programming practices. Additional recommendations are given in Appendix XVII.

3.2.2.2. Recommended programming practices for design, engineering and analysis software

There are many programming languages available in which to write computer programs that can be processed and executed by a computer. New languages are introduced at a rapid rate. Also, the complexity and size of computer programs vary over a very large range. Therefore, it is difficult to make specific recommendations on good programming practices. However, some general guidelines can be provided.

In the coding of a computer program, it is recommended that a set of good programming practices be followed that are suitable for the programming language in use and that are appropriate for the level of complexity and size of the computer program. The following describes one possible set of principles that can be applied when establishing good programming practices:

(a) Program organization
   — clear and easy-to-follow overall program structure;
   — overall program control centralized in a single module;
   — orderly progression of calculation flow;
   — single, well defined function for each module;
   — uniform module layout;
   — input data acquisition centralized in a single program unit (or in a single program unit per module in a multi-module computer program);
   — displaying and saving of major calculation results.

(b) Programming language
   — adoption of a standard programming language;
   — adherence to an applicable programming standard;
   — limited use and thorough documentation of extensions to the standard programming language;
   — use of assembly language only where absolutely necessary.
(c) **Data transfer**
- use of a uniform data transfer technique throughout, unless a data structure requires special handling;
- documentation of data transfer technique(s).

(d) **Input/output**
- input data identification, preparation, organization;
- input checking;
- echoing of input data for visual inspection;
- echoing of computer program version identification in output;
- reports on calculational path, intermediate results and calculation progress;
- issuing of warning and error messages in the event of abnormal conditions;
- adoption of user friendly input and output formats and user’s options.
- use of a single module to read or write a data file;
- documentation of data file content and structure.

(e) **Source statements and variable names**
- specification statements for variables;
- adoption of naming conventions and/or meaningful variable names;
- liberal use of comment statements;
- single purpose use of variables;
- checking of parameter types.

(f) **Hardware and operating system dependencies**
- independence from unique hardware and operating system features;
- documentation of dynamic memory management techniques.

(g) **Other recommended programming techniques**
- initialization of arrays, variables and default parameters;
- clear identification of range, beginning and end of program loops;
- checking of array bounds;
- unambiguous branching decisions;
- standards on the use of subscripted variables;
- avoidance of mixed mode operations;
- standards for structuring arithmetic and logical expressions for clarity;
- testing of intermediate and final results for validity;
- standards on the use of nested constructs;
- manageable module sizes;
- limitation of module parameters to the minimum number required;
- standards for module and loop entry and exit points;
- standards on computer program readability;
- standards on efficient and reliable programming techniques.

(h) **Standard case set**
- Development of a standard set of test cases against which changes to a computer program can be evaluated.
3.2.3. Use of pre-developed software during the software life-cycle

For monitoring, control and safety system software and for design, engineering and analysis software it is possible to make use of pre-developed software modules during the development and maintenance phases of the software life-cycle. The use of pre-developed software modules can accelerate the development of new software or the modification of existing software. To preserve the integrity and reliability of the main computer program, the following actions should be considered when using pre-developed software modules:

— Use modules with a known development history so that the nature of the software quality assurance programme applied during their development can be assessed.
— Use modules with a known version identity taken from a well controlled software library.
— Have access to the design and development documentation so that the proper integration of the modules into the main computer program can be achieved.
— Verify that the design of the modules addresses the applicable components of the main computer program specification to confirm the modules' capabilities and limitations with respect to that specification.
— Verify the proper incorporation of pre-developed modules into the main computer program through a suitable set of unit and integration tests to confirm that the main computer program works as intended.

3.2.4. Use of design, engineering and analysis software

Users of computer programs should be familiar with the features of the computer program and should have training in the preparation of input data sets. They should also be familiar with the nature of the output so that it is interpreted correctly. Users should have access to relevant user's manuals that describe the computer program and its characteristics.

Computer program versions approved for use should be held in a controlled software library. Users should perform calculations only with an approved version from that library.

42 Unit — a separately testable element specified in the design of a computer program component.
The file containing an approved computer program version supplied by the software library to the user should normally be an executable file. If only a source file is available, the user should not make any changes to the computer program and should use the same system components to preserve the configuration.

The user should use computer programs that are validated for the intended use and range of applicability. Validation results should be documented according to defined requirements.

The user should employ input data (e.g. models) that:

— adequately represent the system or process being analysed;
— are documented;
— are verified to ensure that they meet both these requirements.

The user may employ input files that to a large extent have already been verified and documented; in this case, the user should ensure that any changes made to the input (or the revised input file as a whole) still meet the above requirements.

The user should examine the results of the computer program output to ensure that the results produced are reasonable. The user should also ensure that the results and documentation receive appropriate independent review and should file the analysis records in a retrievable manner.

3.2.5. Considerations for legacy codes

The detailed recommendations on design and development discussed in this publication obviously cannot be applied to software that already exists, unless costly and time consuming programmes of reverse engineering are undertaken where the existing program is essentially redeveloped. Particularly for design, engineering and analysis software, it is also a fact that many such computer programs were developed using less formal design and development practices, possibly 10 to 20 years ago. Yet versions of such software remain in use today.

It is generally recognized that the quality attributes of such legacy codes are a concern. A universal consensus has not yet emerged on acceptable practices to address this concern. However, some general guidelines have been proposed.

The following actions should be considered:

— Establish and maintain a software development library;
— Implement a well defined configuration management system, including a software release system;
— Establish a unique identification for the software version in use and place it under configuration control;
— Develop a set of acceptance test cases;
— Carry out a series of tests to validate the identified version for use in defined applications and document the test results;
— Apply up-to-date software quality assurance practices to the design and development of any significant changes to the legacy code; examples of significant changes are:
  — change in theoretical background,
  — change in solution technique,
  — additional calculational capability,
  — change in data structure,
  — change in programming language,
  — change in computer program structure,
  — change in embedded data, such as empirical correlations,
  — change in computer system, assembler or compiler;
— Implement a non-conformance and corrective action system.

3.3. PROCUREMENT

3.3.1. General

The purchase of software may involve a contract to deliver a software package developed according to a set of specifications unique to the procuring organization. Alternatively, the software purchase may involve a software package that has been developed previously and is purchased off the shelf.

There are many circumstances where a decision is made to purchase software components or an entire computer program rather than to develop such software in house. The purchasing organization may not have the expertise, time or resources to carry out any software development. A suitable software package may already exist so that developing another computer program for the same purpose would be a duplication of effort.

Certain controls should be placed on the procurement process in order to ensure that the software purchased possesses the necessary functional capabilities and is compatible with the hardware with which it is expected to interact, and that the appropriate records are available to demonstrate that all required functional and quality characteristics are incorporated in the software.

For any purchase, appropriate software quality assurance requirements should be imposed on the software supplier. This can be achieved by including appropriate software quality assurance requirements in the request for proposal and monitoring the supplier’s contract execution against these requirements.
A similar procurement process should be applied to the acquisition of software upgrades subsequent to the original purchase.

3.3.2. Software requirements specification

The purchasing organization should be clear about the functional and quality requirements for the software to be purchased. An essential component of any purchase order should be a software requirements specification that provides enough details about the nature of the software to be purchased so that any supplier can understand the requirements to be satisfied and so that the purchasing organization can evaluate any bids against the defined requirements. Recommendations for the format and structure of the software requirements specification are given in many published documents, such as IEEE Standard 830 [21].

3.3.3. Vendor qualification

A key step in the procurement process is the establishment of a list of qualified suppliers whose names would be placed on an approved vendors list. A list of possible suppliers should be prepared on the basis of information that they have supplied software to the market that is similar to the kind that is proposed to be purchased. Information should be requested from potential suppliers about their products, the nature of the software products delivered recently, the history of the supplier’s organization, their financial health, and the software quality assurance programme in place in the supplier’s organization. Such information can be gathered from potential suppliers using a standard questionnaire. The supplier evaluation should include examining a copy of the supplier’s software quality assurance manual and determining whether the manual complies with any required quality assurance standard. Assessments of a potential supplier’s capabilities may also include visits to the supplier’s location in order to carry out surveys and qualifying quality assurance audits.

The information gathered in this way should be evaluated against the purchasing organization’s requirements by a multidisciplinary team with technical, financial and quality assurance expertise. The results of such evaluations should be documented, including the decision to place a certain supplier on an approved vendors list. The approved vendors list should be kept current by incorporating new information about potential suppliers and by reflecting the experience with current suppliers.

The evaluation of suitable suppliers should take into account the degree of compliance with any required software quality assurance standard and should include the following considerations:
— The expected interactions across the interface between the supplier and the purchasing organization and the expected effort that the purchasing organization will have to apply to manage the interface.
— The adequacy of the supplier's software quality assurance programme as described and the level of assurance that the declared programme is implemented.
— The adequacy of the supplier's problem reporting and corrective action processes.
— The adequacy of the supplier's configuration management system.
— The adequacy of the supplier's systems for ensuring that the procuring organization's software requirements are satisfied.
— The suitability and completeness of the supplier's typical documentation delivered with the software.

The supplier's software quality assurance programme should comply with a quality assurance standard acceptable to the purchasing organization. Modern software quality assurance standards are likely to include requirements such as those listed below:

— A definition of the software life-cycle with suitable intermediate milestones.
— A commitment to supply specific documentation to the purchasing organization.
— A commitment regarding the content of the documentation and the level of detail of the information.
— Defined review processes applicable to the software design phase.
— Effective verification and validation processes.
— The identification of any software development tools and techniques used during the development effort.
— An effective software configuration management and change control system.
— Evidence that the supplier is actually performing the design and development tasks according to the declared software quality assurance programme.

3.3.4. Request for proposals

The request for proposals (RFP) is the first formal step in the procurement process. The RFP should include the software requirements specification to define the technical requirements. The RFP should also specify the desired commercial conditions, delivery dates, the documentation to be submitted to the purchaser for review, approval and retention, and the right of access to the supplier's premises to carry out audits and surveillance, and to witness crucial tests. The RFP should specify a date by which suppliers are expected to reply and should identify any format requirements applicable to such replies.
For software subject to a regulatory evaluation process before its approval for use, the inclusion in the RFP of a special clause to this end should be considered. A requirement can be specified for a quotation on a final price, inclusive of regulatory approval. Alternatively, separate pricing can be requested for services in support of the regulatory process. The details of this negotiation, and the risk sharing between vendor and client, can be very dependent on the details of the application, the regulatory requirements, and the organizations' previous experience on this issue.

The responses received should be evaluated by a multidisciplinary team to help select the most advantageous proposal. All relevant factors should be considered, including compliance with technical, quality assurance, commercial and delivery requirements. While price is a very important consideration, it is important not to assign such high weight to the price factor that it outweighs all other considerations. A higher purchase price may prove to be cheaper in the long run if the supplier delivers a superior product, better after-sales support or fewer latent defects in the software.

3.3.5. Purchase order

The purchase order for custom developed software should identify the technical, functional and quality assurance requirements applicable to the software and the technical and quality assurance documentation to be provided. The purchase order should also specify the arrangements to manage the interface between supplier and purchaser. This includes the exchange of information about the progress of the work and the purchaser's right of access to the supplier's premises for surveillance, audit and witnessing purposes. The purchasing organization should identify the verification and validation tests it wishes to observe. The purchase order should also address the management of change requests by the purchasing organization after the order is placed and the disposition of concessions requested by the supplier.

The purchase order should also:

— specify the actions in the event that the supplier fails to meet the requirements of the purchase order;
— spell out the methods for transferring to the purchaser ownership of the software and its documentation;
— establish the delivery schedule for the software product and related documentation;
— define the acceptance criteria under which the purchasing organization will receive delivery;
— establish the level of after-sales support to be provided, particularly with respect to latent defects detected after delivery.
For the purchase of off-the-shelf software the content of the purchase order is similar. However, some elements of the purchase order have to cater to the nature of off-the-shelf software. For example, requirements for surveillance, audit and witnessing are not appropriate, since the development process has already taken place. In this case the purchase order should ask for evidence that the supplier has applied an appropriate software quality assurance programme during the design and development phases. The supplier should provide evidence of other users' experience with the software and adequate demonstrations that the software does meet the technical and functional requirements and is fit for the intended use. Such evidence may be supplied by supplying the results of a spectrum of defined test cases demonstrating that the software functions properly over the required range of parameters.

In the procurement function, attention should be paid to obtaining suitable licences for the use of the software. The procurement function should ensure that software documentation is provided for the whole assembly of software supplied. This may be divided amongst the following categories:

1. Manuals and handbooks for proprietary software;
2. Configuration information for the application of proprietary system software whose detailed code may not be available to purchasers;
3. Manuals and handbooks for application software produced for the project;
4. Detailed requirements documentation defining the functional and performance requirements for the project;
5. Detailed design documents defining the software design for the project, with details of both the proprietary software and application software design intended;
6. Detailed listings of the software code for all application software produced for the project;
7. Detailed configuration information for all application software, for the definition of input and output characteristics, system interconnections, system messages, control functions and the like;
8. Reports of verification and review activities, which should be accessible to the purchaser;
9. Reports of the factory tests and site tests done to show the validity of the software operating and maintenance procedures and methods of testing.

3.3.6. Receiving inspection

Upon delivery of the software, the purchasing organization should carry out a number of receiving inspections. These should include verification that software and documentation have been delivered in good condition and in the required quantities. The purchasing organization should also carry out a series of tests to ensure that the software performs as expected on the purchasing organization's computer hardware.
Suppliers' representatives should be allowed to witness such testing and should assist in resolving any difficulties encountered.

3.4. INSPECTION AND TESTING

3.4.1. General

Inspection and testing of a software product, as of any other product, is designed to identify any defects and to check that they are corrected to a sufficient extent to ensure that the software product performs satisfactorily. The main activities of this work, for software, are called verification and validation. An important guide to the principles and methods of software verification and validation is given in Ref. [3].

Verification refers to the checking of the software documents and code. Documentation should be carefully checked to show that, at each stage of the design breakdown to code, the lower stage, which contains greater detail, reflects accurately the stage above. Verification may involve visual or automated checks of documents or of code against good practice and the documentation of the design stage above. It should involve document reviews by knowledgeable individuals other than those who prepared the document, and review meetings. Verification also involves testing of modules and subsystems of code to show that the documents or code meet the design, functions and performance requirements placed on them. Such testing may be required to be independently defined and performed. Verification results and the resolution of any issues raised during verification should be documented.

Validation refers to the testing of the fully integrated software system to show that it meets the requirements originally placed on it. Validation for monitoring, control and safety system software should involve tests of the system at the supplier's premises, followed by tests of the complete installed system on the site. Such testing is normally a condition of the contractual takeover by the nuclear power plant owner or operator. Validation should be carried out against a written and previously defined set of test documents. These test documents should be prepared by staff who are independent of the original design and development team. Validation results should be reported in writing.

Part of the end-to-end process of verification and validation is to record and correct any defects found. Software documentation review and verification can follow established processes applicable to any engineering documents in an organization. Software defect reporting and correction for software may be informal during the very first stages of coding. An informal approach may be useful initially, if it is
necessary to try out and correct workable procedures. However, a formal system should be imposed once suitable work practices are established to carry out the software design and development, with defined reports, logs and records of correction for software that is important to safety.

3.4.2. General recommendations on verification and validation methods

Verification uses processes to ensure that the outputs of one life-cycle phase satisfy the requirements of the previous phase. Verification activities are intended to provide a thorough examination of computer programs and related documents during the design and development phases to demonstrate that the required quality characteristics are achieved, including software reliability and dependability. Verification activities should be performed at appropriate points throughout the software development life-cycle. Verification activities should be applied as early as possible during the software development life-cycle to achieve high quality software at minimum cost.

Validation uses processes to test and evaluate the integrated computer system (hardware and software) to confirm strict compliance with its functional and non-functional requirements and to confirm the absence of unexpected behaviour. Therefore, validation represents a final check of the software product to determine whether all user and safety requirements have been met both by the specifications and by the final product. Validation activities are applied at the end of the software development life-cycle.

The methods used to perform verifications and validations are similar. In both cases various forms of reviews and testing are used.

During any particular software development life-cycle the verification programmes and validation programmes should be designed to complement each other rather than be a substitute for one another.

Additional information about specific verification and validation methods is given in Refs [2, 4, 23]. Appendix XVIII also provides a discussion of verification and validation methods.

3.4.3. Verification of monitoring, control and safety system software

Verification activities and the verification processes to be used for a particular software project should be identified completely in the SDP. Verification processes should be defined in applicable procedures and work instructions.

The results from each verification activity should be recorded in appropriate reports. Verification reports should:
— identify the revision of the documents under review;
— summarize the review and verification activities performed and the methods and tools used;
— list the deficiencies found;
— list the positive findings;
— record the conclusions and recommendations;
— identify the review or verification participants;
— report the detailed results of the verification;
— satisfy the format requirements established for verification reports.

Additional recommendations are given in Appendix XIX regarding characteristics of specific verification and review reports.

3.4.4. Commissioning practices for monitoring, control and safety system software

The commissioning programme for all systems and components of a nuclear power plant should be executed within an established commissioning quality assurance programme. The commissioning of systems containing monitoring, control and safety system software should be performed within this commissioning quality assurance programme.

Within the commissioning quality assurance programme, there are three sets of commissioning processes that are of particular interest related to reactor systems containing monitoring, control and safety system software. They are illustrated in Fig. 5.

FIG. 5. Software commissioning processes.
Additional recommendations regarding commissioning tests are given in Appendix XX.

3.4.5. Verification of design, engineering and analysis software

Both design and computer programming activities should be verified.

The design should be verified to confirm that it satisfies the requirements specification. The set of design verification activities should be performed as defined in the development plan. Design verification methods can include:

— peer reviews — a co-worker checks that the design document is basically correct;
— walkthroughs — a group of peers review the design as the designer describes it;
— formal design reviews — a team of qualified independent peers review a design documents package for compliance with requirements; deficiencies are formally recorded and subsequently followed up to a successful resolution.

The coding should be verified to confirm that it implements the design correctly. The set of computer program verification activities should be performed as defined in the development plan. Code verification methods can include:

— walkthroughs applied to the computer program text;
— mathematical analysis of computer program functions;
— testing of computer program components at various stages of integration.

Additional recommendations are provided in Ref. [4]. A number of sources listed in the Bibliography should also be consulted.

3.4.6. Validation of design, engineering and analysis software

A specific computer program should be validated to determine and to demonstrate that it is suitable for the intended application and to provide an indication of the uncertainty allowances that should be taken into consideration when the program output is interpreted.

Examples of validation methods include comparison of the computer program results with:

— experimental data,
— commissioning data and operating experience,
— results of hand calculations,
— solutions to standard or benchmark problems,
— closed mathematical solutions,
— results of another validated computer program.
The conclusions regarding the validation results should take into consideration the accuracy of the information against which the computer program is validated. Validation activities should be performed according to a documented plan. The plan should provide information on:

— identity of the computer program version being validated,
— description of the application for which the computer program is to be used,
— description of the validation methods to be used,
— acceptance criteria to be used in evaluating computer program accuracy and uncertainty.

3.4.7. Testing tools

Testing is one of the most expensive and demanding tasks in software development and maintenance. The main purpose of testing is to identify any errors that are present in the computer program. Several tools are available to support testing activities.

A simulator establishes a controlled environment for the testing process. The computer program to be tested is simulated in this environment and the specified set of tests is carried out. A test data generator supplies the test data needed to test the functionality of the computer program being modified. The programmer sets the criteria for generating the test cases.

Before integration and unit testing, it is important to know the potential data flow and control flow paths that may be affected by a change in a computer program. Test path generators can be used for this purpose.

Appendix XVIII provides additional guidance on testing. These recommendations should also be considered when testing tools are evaluated for their functionality.

4. ASSESSMENT

4.1. MANAGEMENT SELF-ASSESSMENT

4.1.1. Effect of management actions

The error rate experienced during the life-cycle of a computer program is influenced by the size of the program, the time taken to develop the program, the staff
size and effort, process productivity, project complexity and the personnel build-up rate. Each of these parameters has an effect on the number of errors created, and therefore on the reliability of the computer program.

The effect of most of these parameters on the error rate can be influenced by management policies and actions. Management decisions about the way a software project is carried out affect the reliability of the software. The influence of these parameters on the error rate is summarized below.

**Effect of size** — the incidence of errors grows at a nearly linear rate (on a log-log plot) with increase in system size. The error rate also increases with system size. The mean time to failure becomes poorer with increasing system size.

**Effect of development time** — extending the planned development time beyond the minimum development time reduces the number of errors created. Error models discussed by Putnam and Myers [24] predict this behaviour. Allowing enough time to do the work right reduces the number of errors committed.

**Effect of staff size** — the average number of staff and the number of expected errors decrease with lengthening development time.

**Effect of productivity** — the number of errors declines rapidly as the organization's productivity (as measured by a suitably defined productivity index) improves.

**Effect of application type** — different types of applications have different average productivity indexes. The productivity index involves the complexity of the work being done. Therefore, the more complex the software product, the greater the number of errors.

**Effect of personnel build-up** — the number of errors increases as the rate of building up personnel increases.

**Effect of remaining errors** — in general, the remaining errors are proportional to the total number of errors. The approximate number of errors remaining at any one time can be estimated using models proposed by Putnam and Myers [25]. Because the remaining errors have a mathematical relationship to the total number of errors, the same trade-offs that reduce total errors also reduce the remaining errors.

A detailed discussion of proposed mathematical models describing the relationships of various parameters affecting the error rate is provided in Ref. [25]. This source also describes the collections of empirical data on which the proposed mathematical models are based.

### 4.1.2. Quality measures as a management tool

If software is to be considered as an engineered product using proven and well controlled production processes that form part of an effective quality assurance
programme, management of an organization involved with software needs to obtain acceptable answers to the following questions:

— Can we do it?
— How long will it take?
— How many people will it take?
— What is the risk?
— What are the trade-offs?
— How many errors will there be?
— Can we measure process improvement?

For these questions to be answered with confidence, they need to be answered in quantitative terms. A number of quality measures have been proposed to help provide the required answers.

The measures should provide information throughout the life-cycle of the software. They should be designed to achieve optimum operational reliability levels and to provide a measurement of actual reliability levels achieved. The basic goal is to provide the elements of a measurement programme that supports a constructive approach for achieving optimum reliability of the end product.

The measures should assist in building consistent methods for software development and use. The measures should also support management in directing product development and support towards specific reliability goals. The purpose is to provide a common set of measures through which a meaningful exchange of data and evaluations can occur. Successful application of the measures is dependent upon their appropriate application in a specific environment.

4.1.3. General guidance on the use of quality measures

Measurement is a basic tool of engineering. Measures only become accepted and applied by practitioners when a sufficient foundation of empirical evidence accumulates that reaffirms their validity. Only under these circumstances should collecting and analysing measures form part of software development as an engineering or scientific discipline.

Measurement provides the tools to identify critical problem areas. Using the Pareto principle that 20% of causes create 80% of problems one must find the dominant 20% to achieve significant improvements. These are the leverage points for process improvement.

Part of the difficulty in defining suitable measures for the software development process is inherent in the intellectual, non-deterministic nature of the process. But much of the difficulty is also due to the nature of the measures proposed. Most early
measures were related to source code characteristics. Many measures of software complexity are proposed without a clearly defined theory or model of the activity being measured. Many measures have no obvious practical application. Few measures have been validated empirically.

An approach to measurement for practitioners emphasizes two lessons:

— Focus on a few key quality characteristics rather than attempt to measure everything.
— Rely on simple measures extractable from common design products and other sources.

The recommended measurement approach follows five steps:

(a) Define the object of measurement (i.e. software design). Often this means developing a model of the object.
(b) Identify the characteristics or attributes to be measured.
(c) Specify the purpose or intended use of the measurement results.
(d) Collect the appropriate data.
(e) Verify and/or modify the model on the basis of an analysis of, and experience with, the collected data.

The following general guidelines should be considered when selecting quality measures for software:

(1) For an estimating method that works, use strategic variables that are already in wide use. They are available in software databases, as indicated in Ref. [24].
(2) Define an objective measure of productivity because it is commonly an ingredient of estimating equations. The traditional measure of source lines of code per person-month is notorious for its wide variability. It may not be a good choice.
(3) There is evidence that software reliability (or number of errors) is related to productivity. When productivity improves, errors seem to decline, or with more emphasis on quality, productivity increases. Find out what this relationship is and incorporate it into the system of quality measures.
(4) Common measures involving software are noisy. They have large deviations from what the numbers might be if taken under ideal conditions. Any proposed set of equations used to calculate quality measures should be capable of operating in the presence of noise.
(5) If the input to the equations is noisy, the output will also be noisy. This uncertainty is related to the risk that the actual values will deviate from the estimate as the project proceeds.
Measurements are not only used to control the processes to create and use a computer program. Measurements should also be used to assist management in planning the activities of their organization. Measurements are important in this respect to assist managers in this planning activity when using simulations to predict expected work results. Simulation contributes significantly to illustrating the dynamics of behaviour of the software process. Higher level managers need to understand this dynamic behaviour. By understanding the behaviour of development processes broadly, they have less need to learn the many details of software techniques. The reason for widespread overruns and slippages is that organizations have not understood the real behaviour of software development, and how sensitive software development is to management influences. The patterns in historical data show that there is a linkage between system size, as measured in source lines of code, and development time, effort (person-months, person-years, costs), personnel, productivity and number of defects. These relationships can be made more precise when the database of all software applications is stratified by type of application.

4.1.4. Functional classification of measures

Measures can be divided into two functional categories:

— Product measures: these are applied to the software components. Product measures are divided into six subcategories.
— Process measures: these are applied to the activities of development, test and maintenance. Process measures are divided into three subcategories.

4.1.4.1. Product measures

Product measures address the cause and effect of the static and dynamic aspects of both projected reliability prior to operation and operational reliability. For example, reliability may change radically during the maintenance effort owing to the complexity of the system design. These product measures cover more than the correctness aspect of reliability; they also address the system utility aspect of reliability.

The following six product measure subcategories address these dimensions of reliability:

(a) Errors — count of defects in terms of human cause, program bugs, observed system malfunctions.
(b) Mean-time-to-failure; failure rate — derivative measures of defect occurrence and time.
Reliability growth and projection — the assessment of change in the degree of freedom from failures of the product under test and in operation.

Remaining product faults — the assessment of the degree of freedom from faults of the product during development, test or maintenance.

Completeness and consistency — the assessment of the presence and compatibility of all necessary software system parts.

Complexity — the assessment of complicating factors in a system.

4.1.4.2. Process measures

The three process measure subcategories are directly related to process management:

1. Management control — the assessment of guidance of the development and maintenance processes.
2. Coverage — the assessment of the presence of all necessary activities to develop or maintain the software product.

4.1.5. Specific guidance on the use of quality measures

Information about the establishment of a programme to collect specific measures is provided in IEEE 1061 [25].

Detailed definitions of many measures evaluating the reliability of software can be found in IEEE 982.1 [26]. An extensive guide for the use of these measures is provided in IEEE 982.2 [27].

These, or similar, documents should be reviewed when an organization is considering the introduction of specific quality measures. When an organization embarks on such a course of action it is advisable that it begin with measures that are easy to understand and collect. It is important that the organization gain experience with an initial basic set of measures before introducing more complex ones.

The subject of quality metrics is discussed at length in a number of books listed in the Bibliography.

4.2. INDEPENDENT ASSESSMENT

Independent assessment by a quality assurance organization gives confidence that the software developer or user organization has indeed followed the defined quality assurance programme and has performed work activities according to
prescribed procedures which have been agreed for the design, development or use of the software. Such independent assessments increase the confidence that the software will function as required and will provide reliable results. The independent assessors may raise corrective action notices if the prescribed procedures are not followed suitably.

In software, the process of independent assessment may also include technical reviews of all the software, possibly also with hardware design, or of samples of the software. The aim may be to determine if the agreed technical standards have been applied suitably, or to assess the software in its own right as suitable in design and technical implementation. Such technical assessments should themselves be carried out under suitable procedures, with suitable plans. Such technical assessments are not of themselves activities performed by quality assurance experts.

In some countries, independent third party assessments may be undertaken by the regulator or a body acceptable to the regulator. They assist in providing confidence in the licensee's/supplier's software development processes. The extent of the third party assessments necessary to provide the required level of confidence should be carefully considered and the assessment plan should be agreed to by all parties so that the appropriate resources can be made available at the desired time.

Some of the third party assessments should involve examination of the development, verification and validation processes using techniques such as quality assurance audits and technical inspections.

Third party assessments may involve examinations of the software product using methods such as:

- static analysis
- dynamic analysis
- code/data inspections
- test coverage analysis.

The software product assessments should be undertaken on the final production version of the software.

All assessment results should be recorded and reviewed.

Where it is determined that a change is required to the computer program documentation and/or source code, appropriate change procedures should be applied.
Appendix I

ILLUSTRATION OF A GRADED SOFTWARE QUALITY ASSURANCE PROGRAMME

A suggested outline of graded requirements for software quality assurance plans is given. The activities are summarized and indicative only. The tables follow the organizational headings of Ref. [1]. Activities are graded for different integrity levels of software. The software groups are for off-line software for safety analysis or performance evaluation, and for on-line software in Categories A, B or C.

Five software classification groups are used here for stating graded requirements. Consistent with Section 2.1, they are defined as:

- Monitoring, control and safety system software ('real time')
  - Category A is assigned to functions, systems and equipment which have a principal role in the achievement or maintenance of nuclear power plant safety [9].
  - Category B is assigned to functions, systems and equipment which have a supporting safety role to systems of Category A [9].
  - Category C is assigned to functions, systems and equipment which have an auxiliary or indirect role in the achievement or maintenance of nuclear power plant safety [9].

- Design, engineering and analysis software (not 'real time')
  - Safety analysis software: software which is used to carry out safety analysis and assessment of nuclear power plant systems and equipment.
  - Performance evaluation software: software which is used to carry out design or to evaluate performance of nuclear power plant systems and equipment.

The abbreviations in the body of the tables have the following meanings:

- AHARP — as high as reasonably achievable,
- HR — highly recommended,
- R — recommended,
- '—' — 'no recommendation,
- ALARP — as low as reasonably practicable,
- SAT — site acceptance test,
- FAT — factory acceptance test,
Sometimes a comment is given. Many international and other standards make the equivalent activity to that shown with ‘HR’ mandatory. Reference should be made to Ref. [3] for specific methods in many cases.

The margin between Categories B and C is flexible, and shown dotted.

### 1. Management: quality assurance programme

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not ‘real time’</th>
<th>‘real time’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td>Cat. A</td>
<td>Cat. B</td>
</tr>
<tr>
<td>Software quality plan</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>Software safety review</td>
<td>HR</td>
<td>—</td>
</tr>
<tr>
<td>Software safety justification</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ISO 9000 series</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>Ref. [1]</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Ref. [4]</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

### 2. Management: training and qualification

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not ‘real time’</th>
<th>‘real time’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td></td>
<td>Cat. A</td>
<td>Cat. B</td>
</tr>
<tr>
<td>Graduate qualification in an appropriate discipline</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Normal software training</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Software experience</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Specific nuclear safety awareness or experience</td>
<td>HR</td>
<td>—</td>
</tr>
<tr>
<td>Supervision with software experience</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>
### Management: non-conformance control and corrective action

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not ‘real time’</th>
<th></th>
<th>’real time’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
<td>Cat. A</td>
</tr>
<tr>
<td>Established procedure for defect review and retest</td>
<td>HR</td>
<td>R</td>
<td>100%</td>
</tr>
<tr>
<td>Defect reports from in-use stage onwards</td>
<td>HR</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Formal procedures for change control, from FAT or use stage onwards</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Verification of changes to same standard as original design</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Regression testing to previously identified scope</td>
<td>HR</td>
<td>R</td>
<td>HR</td>
</tr>
</tbody>
</table>

### Management: document control and records, configuration control

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not ‘real time’</th>
<th></th>
<th>’real time’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
<td>Cat. A</td>
</tr>
<tr>
<td>Formal control over all design and code documents</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Operation manuals</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>User manuals and maintenance handbooks</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Use of automatic document configuration control tools</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Use of automatic code configuration control tools</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Use of automatic data configuration control tools</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Log of all defects and corrections</td>
<td>R</td>
<td>—</td>
<td>HR</td>
</tr>
<tr>
<td>Plant life retention of documents, code</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
</tbody>
</table>
5. **Performance: work planning**

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not 'real time'</th>
<th>‘real time’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Defined development and implementation life-cycle</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Requirements shown complete, correct, not ambiguous</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Requirements, design, code of all application software available to utility</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Requirements, design, code of all system software available to utility</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>Proprietary code, available, 100% transparent to user</td>
<td>—</td>
<td>HR</td>
</tr>
<tr>
<td>Proprietary code not available to user</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>IEC 880 applied</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>IEC 880 base, with stated interpretation</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Removal of complexity</td>
<td>ALARP</td>
<td>ALARP</td>
</tr>
<tr>
<td>Rejection of interrupts, pointers, recursion</td>
<td>ALARP</td>
<td>R</td>
</tr>
<tr>
<td>Design and code standards manual applied</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>Selection of language, typed, structured</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Language with known syntax, proven</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Limited use of assembler</td>
<td>ALARP</td>
<td>—</td>
</tr>
<tr>
<td>Optimized compilation of code</td>
<td>No</td>
<td>—</td>
</tr>
<tr>
<td>Independent verification and validation planned</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Tool output verified and validated, or tools otherwise written to application quality</td>
<td>R</td>
<td>—</td>
</tr>
</tbody>
</table>
### 6. Performance: analysis, design and implementation

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not 'real time'</th>
<th>'real time'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Structured documentation required to control design and implementation</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>Requirements in formal logic, or modelled, simulated</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Requirements in algebraic and algorithmic forms</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Human factors review of HMI design</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Software requirements document</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>Software design document</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Data structure design document</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Verification of documents to level above</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>Verification of application code to design</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Verification of application configuration data to sources</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Independence of verification team from implementation</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Independent organization for verification</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

### 7. Performance: procurement

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not 'real time'</th>
<th>'real time'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Pre-qualification tender list</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>Vendor of proven ability for high integrity software</td>
<td>HR</td>
<td>—</td>
</tr>
</tbody>
</table>
### 7. Performance: procurement (cont.)

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not ‘real time’</th>
<th>‘real time’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Vendor of known ability for software</td>
<td>—</td>
<td>R</td>
</tr>
<tr>
<td>Vendor’s QA approved to that of application</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>Evaluation and assessment report of vendor’s products</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>On-line operational experience reports of products</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Human factors review of each product’s HMI</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Regulatory acceptance of provisional price in contract</td>
<td>R</td>
<td>—</td>
</tr>
</tbody>
</table>

### 8. Performance: inspection and testing

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not ‘real time’</th>
<th>‘real time’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Test plans formally produced and verified</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Software module tests</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>System integration tests</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Black box, white box tests</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Functional testing, dynamic inputs</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Function testing through application range</td>
<td>HR</td>
<td>R</td>
</tr>
<tr>
<td>Timing analysis and test, loading, performance testing</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Human ‘eyeball’ review of all documents and code</td>
<td>R</td>
<td>—</td>
</tr>
</tbody>
</table>
### 8. Performance: inspection and testing (cont.)

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not 'real time'</th>
<th>'real time'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Statistically random test trajectories</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>FAT document and report</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SAT document and report</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>User acceptance test document and report</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Independence of FAT and SAT or user test definitions from designers and implementers</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

### 9. Performance: inspection and testing

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not 'real time'</th>
<th>'real time'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Trace of requirements to test</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>Independence of validation team from implementation</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Independent organization for validation</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Code analysis tools to show complete, correct code</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Statement test cover, path test cover (minimum recommended values)</td>
<td>—</td>
<td>100%, AHARP</td>
</tr>
<tr>
<td>Source code matches target code, or validated compiler and loading tools</td>
<td>100%, 20%</td>
<td>50% min.</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>—</td>
</tr>
</tbody>
</table>

79
## 10. Assessment: management self-assessment

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not 'real time'</th>
<th>'real time'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Self-assessment of management performance</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Mission of organization</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Expectations of organization</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Opportunities for improvement</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

## 11. Assessment: independent assessment time

<table>
<thead>
<tr>
<th>Activity description</th>
<th>not 'real time'</th>
<th>'real time'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety analysis</td>
<td>Performance</td>
</tr>
<tr>
<td>Independent external group assessment of adherence to software quality plan</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>Independent internal group assessment of adherence to software quality plan</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Internal group assessment of software technical qualities to internal standards</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>Resolution of anomalies by formal report</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Appendix II

PROPOSED PROCEDURE FOR INTERFACE CONTROL

SCOPE

This procedure provides an overview of the controls that must be established in order to manage the interfaces between an organization's internal groups, between the organization and its suppliers, and between the organization and its clients.

PURPOSE

This procedure specifies how the organization defines and manages internal and external interfaces and identifies other relevant procedures that help manage particular interfaces to address the basic requirements defined in its overall quality assurance manual.

TYPES OF INTERFACE

During the execution of any work the responsible manager shall consider interfaces with:

(a) Groups within the organization;
(b) Organizations involved with other phases of the project (i.e. construction, commissioning and operating organizations);
(c) Any supplier or design consultant;
(d) The client.

INTERFACE CONTROL WITHIN THE ORGANIZATION

Assignment of responsibilities

The responsible manager shall direct that:

(1) All the units in the organization required to participate in the work be identified, as appropriate, in project plans, generic or project specific quality assurance manuals, and work scope documentation;
(2) The roles of all participating units be identified, as appropriate, in project plans, or generic or project specific quality assurance manuals;
(3) Specific work responsibilities be documented in appropriate work scope documents.

Control of documents crossing organizational boundaries

The responsible manager shall direct that:

(1) Each work scope document identify the technical documents required to be supplied by others and these requirements be communicated in writing to the interfacing organizations;
(2) Each work scope document identify the technical documents to be supplied to others and these requirements be communicated to the interfacing organizations;
(3) Appropriate distribution lists be developed and issued to the records management staff for all technical and procurement documents to be issued by the organization;
(4) Project plans and quality assurance manuals, as appropriate, specify the procedures controlling the release and distribution of technical and procurement documents;
(5) Project plans and quality assurance manuals, as appropriate, specify the procedures governing the preparation, review, approval and revision of technical and procurement documents;
(6) Design verification plans, quality engineering plans and schedules specify the required reviews and the organizations participating in the reviews.

Communication

The responsible manager shall direct that:

(1) Staff be briefed on the interfaces affecting them;
(2) Communication sessions be held and documented to inform staff of project requirements, and of the content of applicable project plans, quality assurance manuals and procedures;
(3) All interfacing organizations hold and document regular meetings to review work progress and identify all difficulties encountered.
CONTROL OF INTERFACES WITH ORGANIZATIONS INVOLVED WITH OTHER PHASES OF A PROJECT

Assignment of responsibilities

The responsible manager shall direct that:

(1) The external organizations be identified, as appropriate, in project plans and applicable quality assurance manuals;
(2) The organization's units expected to interact with the external organization be identified, as appropriate, in project plans and applicable quality assurance manuals;
(3) Specific work responsibilities of the organization's units be documented in appropriate work scope documents.

Control of documents crossing organizational boundaries

The responsible manager shall direct that:

(1) Each work scope document identify the technical documents to be supplied by the external organization and this requirement be communicated in writing to that organization;
(2) Each work scope document identify the technical documents to be supplied to the external organization and these requirements be communicated to that organization;
(3) Appropriate distribution lists be developed and issued to the records management staff for all technical documents to be issued to the external organization;
(4) The external organization issue its documents to the organization according to distribution lists acceptable to the responsible manager;
(5) Project plans and applicable quality assurance manuals, as appropriate, specify the procedures controlling the release and distribution of technical and procurement documents to the external organization;
(6) Project plans and quality assurance manuals, as appropriate, specify the procedures governing the review, acceptance or approval, as appropriate, of technical documents prepared by the external organization;
(7) Design verification plans, quality engineering plans, test plans and schedules specify the required reviews and the organizations participating in the reviews.
Communication

The responsible manager shall direct that:

1. Appropriate correspondence procedures be prepared and issued to all staff interacting with the external organization;
2. Affected staff be briefed on the interface with the external organization;
3. Positions corresponding to the formal contacts with the external organization in the organization be identified in project plans and quality assurance manuals, as appropriate;
4. Communications sessions be held with the external organization and documented to review the scope and technical requirements of the assignment to ensure a full understanding of all requirements by all parties;
5. Regular meetings be held with the external organization and documented to review work progress and identify all difficulties encountered.

CONTROL OF INTERFACES WITH SUPPLIERS AND DESIGN CONSULTANTS

Assignment of responsibilities

The responsible manager shall direct that:

1. The units in the organization responsible for procurement be identified, as appropriate, in project plans and applicable quality assurance manuals and work scope documents;
2. The specific roles of the procurement organizations be identified, as appropriate, in project plans and applicable quality assurance manuals;
3. Specific work responsibilities be documented in appropriate work scope documents;
4. Suppliers' responsibilities be specified in procurement documents.

Control of documents crossing organizational boundaries

The preparation, approval and issue of procurement related documents are controlled through applicable procedures. The submission of suppliers' documents for review, acceptance or approval is governed by the applicable procurement and contract documents. Applicable records management requirements are defined in the applicable procedures.
Communication

The responsible manager shall direct that appropriate meetings and discussions take place with the supplier at the start of the contract to ensure that the supplier fully understands all of the organization’s requirements.

Continuing communication between the procurement function and suppliers is controlled through applicable procedures.

Communication between the design and development functions and suppliers involving technical issues shall be controlled through applicable procedures (e.g. procedures for bid evaluation, deviation disposition requests, design change instructions). The responsible manager shall also direct that the design and development functions maintain continuing communication with the supplier on technical issues not associated with change notices and concessions.

CONTROL OF INTERFACES WITH THE CLIENT

The responsible manager shall direct that:

(1) The contract with the client be reviewed to determine the communications and document transmittal requirements specified therein;
(2) Appropriate project specific procedures be prepared and implemented to control continuing communication and the management of documents crossing the interface (such procedures shall reflect the requirements of the contract and applicable quality assurance standards).
Appendix III
CONSIDERATIONS BEFORE ACQUISITION
OF COMPUTERIZED TOOLS

A number of factors should be taken into account before computerized tools are acquired:

— Capability: The tool should be capable of performing the intended function. In this context it is important to recognize that a certain technique or methods should be shown to work manually without a computerized tool before an automating tool is introduced.

— Features: Once it is determined that a computerized tool can improve the performance of a certain function, the required features of such a tool should be established. The importance of each feature should be rated. A specification should be prepared to document the desired features. The selection of an appropriate tool should be based on the degree of compliance with this specification.

— Cost and benefits: The benefits of the tool should be evaluated in terms of its effect on quality and quality assurance of the organization’s principal activities, productivity, responsiveness, cost reduction and the degree to which the use of the tool will restrict the variety of work practices.

— Platform: The specific hardware and software environments within which the tool has to operate should be considered.

— Programming language: The tool should be compatible with the programming languages in which the software on which the tool is expected to operate is written.

— Ease of use: Tools should have a user-friendly interface. It is desirable that any new tool be similar to other tools in use in the organization in the way it interfaces with the users.

— Openness of architecture: Where it is expected that several tools have to be used in concert in order to achieve a set of desired functions, it is important to select tools with open architectures to obtain the necessary extensibility and flexibility.

— Stability of the tool vendor: The likelihood of the vendor being in business for the foreseeable future is an important consideration in order to be assured of continued product support.

— Organizational culture: Organizations develop certain unique ways of working and doing things. To ensure easy acceptance of a tool, it is essential to take such culture and work patterns into consideration.
Appendix IV

FUNCTIONS OF COMPUTER PROGRAM UNDERSTANDING AND REVERSE ENGINEERING TOOLS

Several tools have been produced and are widely available to analyse and highlight code structure, and track and control software execution. Among them are tools such as the program slicer, static analyser, dynamic analyser, data flow analyser, cross-referencer, dependency analyser and graphic transformation tools.

The program slicer identifies to a programmer only those parts of a computer program that are affected by a proposed change. A program slicer marks all sections of a program that may influence the value of a variable at a given point in the program. Program slicers also display data links and related characteristics so that the programmer can track the effect of changes.

A static analyser provides information about a computer program such as modules, procedures, variables, data elements and programme structure. A static analyser permits general viewing of the program text and generates summaries of contents and usage of selected elements in the program text such as variables and objects.

In order to fully understand the impact of proposed changes to a computer program it is also necessary to control and analyse the program when it is executing. A dynamic analyser allows the programmer to trace the execution path while the program is running. This helps identify the paths that will be affected by a change and those through which a change must be made.

A data flow analyser tracks all possible data flow and control flow paths in the program. This capability is especially important when the effect of a change on other parts of the computer program has to be examined.

The cross-referencer creates an index of the usage of a given program entity. For example, this tool can produce information on the declarations of a variable and all the sections in the program where the variable has been set and used.

A dependency analyser facilitates an understanding of the interrelationships between the various entities in a computer program. The tool typically generates graphical representations of these dependencies where the node in the graph represents a program entity and an arc represents the dependency between entities.

A transformation tool converts computer programs between different forms of representation, usually between text and graphics. Because of the impact of visual representations created by a transformation tool the programmer can view and understand the computer program in a manner that would not be possible just with text.
Appendix V

GENERAL TRAINING GUIDELINES

In general, training programmes should be provided shortly before staff need the information to perform their work. The same rule applies to training in quality assurance topics. There is a basic set of topics that should be covered in any general training on quality assurance topics. These include:

— General topics such as definitions of quality, quality assurance, the seven basic quality tools\(^43\), some examples of management principles to achieve quality, examples of quality maturity indices (e.g. the SEI\(^44\) maturity index), basic concepts about cost of quality, basic concepts of statistical quality control and control of variation.
— The basic elements of any quality assurance programme, using any recognized general quality assurance standard as a basis.
— Review of the elements of the software quality assurance standard chosen by the organization as a basis of its software quality assurance programme.
— Review of the content of the organization's software quality assurance manual to demonstrate how it addresses the requirements of the governing software quality assurance standard.
— Review of the system holding the procedures that implement the software quality assurance programme.
— Review of the main procedures implementing the software quality assurance programme.
— On the job training for familiarity with the execution of work processes, such as preparing, reviewing and approving design documents, coding, testing, error removal, change control, and the operation of configuration and version control systems.

Organizational structures, quality assurance programmes and individual work processes change as organizations, the economic environment, technology and the scope of activities undertaken change. Therefore, there is a need for ongoing sessions on specific quality assurance topics to keep staff aware of latest developments. If staff

\(^43\) The seven basic quality tools for quality control are: checklist (or check sheet), Pareto diagram, histogram, scatter diagram, run chart, control chart, cause-and-effect diagram.

\(^44\) SEI — Software Engineering Institute, which is associated with the Carnegie Mellon University, Pittsburgh, PA, USA.
have received a set of training sessions on basic quality assurance topics, then any additional sessions to address new developments can build on this basic knowledge. Such upgrading sessions can be kept relatively brief.

If organizations engage in any quality training of their staff, such training should not be restricted to technical staff and those working in the quality assurance function. Management staff, and especially senior management staff, should also be exposed to quality training. Management awareness of quality assurance concepts is important because it is a well established principle that a culture of pursuing the achievement and improvement of quality in an organization is possible only if senior management provide the necessary leadership and actively support and participate in the implementation of quality assurance programmes.

Teaching senior managers about quality meets a number of obstacles:

— Senior managers are busy people, who do not have the time or inclination for more training.
— It is difficult to make senior managers aware that they need to know more about quality. They must reach this conclusion themselves.
— Senior managers often delegate responsibility for quality to subordinates. Such delegation is inappropriate to the extent that quality requires leadership and leadership cannot be delegated.
— Quality improvements are often achieved only over the long term. They take time and patience. Senior managers are often under pressure to achieve short term results.
— Senior managers need to acknowledge and use the expertise of their own employees who are working in the quality assurance function.

Certain techniques can facilitate training of senior managers. They include:

— Provide environments where senior managers can learn on their own terms by doing and observing the consequences of their decisions rather than attending training presentations.
— Involve an external consultant who can work well within the organization’s culture since senior managers are reluctant to accept subordinates as teachers.
— Make available, especially during the initial stages of learning, environments that are removed from the distractions of the office. More can be achieved in two or three days of concentrated learning than by spending the same effort spread over a longer time period under circumstances where the managers are distracted by the demands of their daily work.
— Establish tangible outcomes for the learning experience that are directly applicable to their organization, such as completing a quality assessment or developing an improvement strategy for the organization.
— Structure the learning experience so that there are frequent visible accomplishments, such as specific conclusions, decisions or action plans that are directly applicable at work.
— Introduce senior managers to new and different viewpoints through suitable speakers who are perceived as being at their level or higher, such as executives from other organizations who have demonstrated quality achievements.
— Focus the discussions on ways to identify and implement improvements in the organization’s management systems using a terminology that is familiar to the organization. Avoid structuring the discussions around a generally promoted approach to quality improvement, such as total quality management, since such a ‘standard’ technique may not be appropriate for the organization.

There are no universally proven methods to help leaders gain the understanding and commitment to lead a quality improvement effort. The approach must take into consideration the organization’s history and culture, and the personalities involved. The following four steps can be used as a framework from which to develop a programme where senior managers learn by doing while developing leadership skills:

(1) Assess the organization’s quality position in order to develop the creative tension needed to stimulate improvement. Use available assessment models such quality maturity models to help perform such an assessment.
(2) Develop a shared vision and philosophy about what the organization’s desired position on the chosen maturity model should be. Identify specific areas where improvement is required.
(3) Select the criteria for a suitable quality management system. Usually, these criteria are embodied in one or more quality assurance and technical standards or in one of the national excellence award programmes.
(4) Develop the goals, strategy and plan for implementing and improving the quality management system. Goals should be derived from the gaps between the state of the organization’s current quality management system (step 1 above) and the desired state (step 3 above). To close these gaps requires a structured improvement strategy. Such a strategy must include senior management’s continuing efforts to communicate the organization’s goals to the workforce, to put in place and maintain systems and resources to execute the improvement plan, implementing changes to work practices and systems, providing appropriate training to the workforce, and evaluating the results of the effort to determine results achieved and to identify weaknesses in the implementation plan.
Appendix VI

PROPOSED OUTLINES FOR TRAINING PROGRAMMES ON QUALITY ASSURANCE

The proposed training outlines are generic. Specific training programmes should be tailored to the organization's specific activities. The fundamental quality assurance concepts that are common to quality assurance programmes applicable to any type of activity should be understood. In addition, organizations should provide training that addresses specifically the software related elements in their quality assurance programmes.

INTRODUCTION TO QUALITY ASSURANCE FOR MANAGERS AND QUALITY ASSURANCE STAFF

Objectives

— Recognize the quality assurance programme as a management system.
— Demonstrate senior management's commitment to implement an effective quality assurance programme.
— Achieve a common understanding of quality assurance principles.
— Achieve a common understanding of the organization's quality assurance programme.
— Educate managers and quality assurance staff in quality assurance principles to ensure that the exercise of leadership and direction is consistent with the organization's quality assurance programme.
— Clarify management's roles and responsibilities in implementing quality assurance programmes.
— Achieve more efficient, effective and economical operations.
— Promote continuous improvement of work processes, products and services.
— Facilitate the transfer of information from trained managers to their staff.
— Translate general quality and quality assurance concepts into the organization's specific quality assurance programmes.
— Transmit to managers and quality assurance staff an overview of the content of applicable quality assurance standards.
— Develop and maintain expertise to teach about the organization's quality assurance programmes.
— Provide an auditable record of training in quality assurance as required by applicable quality assurance standards.
Participants’ requirements and expectations

— Confirm participants’ actual requirements and expectations.
— Ensure a basic understanding of quality assurance principles.
— Ensure improved organization and management of work.
— Enhance the ability to give clear directions to staff that are consistent with the organization’s quality assurance programme.
— Reduce surprises such as audit observations, budget overruns, design review actions, non-conformance reports.
— Arrange a training course of modest duration, say no more than a few days.
— Make sure that the information applies to actual work situations.
— Provide practice in what the course teaches in workshops, for example.

Requirements for effective trainers

— Good presentation skills.
— Knowledgeable in the subject of quality assurance.
— Seen to know about quality assurance.
— Recognized as leaders in the organization.
— Available to prepare and deliver the course and to answer questions during the course and afterwards.
— Members of the organization for at least 10 years.
— Some knowledge of education principles.

Course content

(a) What does ‘quality’ mean? Discuss the meaning of ‘quality’ as the organization uses the term.
(b) What is a quality assurance programme and what is it not? Discuss what a quality assurance programme can and cannot do for an organization and its members.
(c) The five main elements of any quality assurance programme:
   — Define the organization, roles and responsibilities;
   — Define work practices;
   — Define verification, inspection and testing practices;
   — Define audit and monitoring practices;
   — Define processes for improving the quality assurance programme.
   Illustrate the value and importance of these elements using examples that are available in the published literature.
(d) How do applicable standards address the five main quality assurance programme elements? Illustrate by example and direct correspondence how applicable standards specify requirements that match the five elements.
(c) How does the organization address quality assurance requirements in its quality assurance programme?

(f) Current strengths and weaknesses of the organization's quality assurance programme.

(g) Review of case studies, preferably based on the organization's past experiences.

(h) The documentation of a quality assurance programme: quality assurance manual, procedures and work instructions.

(i) The role of the quality assurance function.

(j) The role of quality assurance staff.

(k) The role of the line manager in implementing a quality assurance programme.

(l) Who is responsible for "quality"?

(m) The quality assurance programme as a management tool.

(n) Review of the general processes for software design, development, testing, integration and use.

(o) Review of the general procurement process.

(p) Essential features of effective procedures.

Handouts

— Distribute to attendees a week or so before the course starts.
— Ensure that handouts consist of text and copies of all illustrations used.
— Keep handouts consistent with the applicable standards and with the organization's quality assurance programme.
— Have all material suitably bound.
— Establish a table of contents.
— Have a glossary of terms.
— Have a keyword index.
— Have a bibliography for further reading.
— Include a message from an executive manager.
— Include an up to date index of all procedures and work instructions.
— Have a list of references at the end of each course section.
— Maintain all training material in electronic form for easy updating.
— Subject handout materials to an independent review process.
— Test market the handouts with a sample of potential attendees.
— Provide space in the handouts for attendees to add notes.

Classroom facilities

— Use a suitable training room.
— Provide appropriate audiovisual equipment.
— Provide refreshments as necessary.
— Configure seating arrangements so that trainers can interact with the class.
— Insulate attendees from interruptions.
— Provide pointer, flip charts and markers for trainers.
— Provide name cards for attendees.

INTRODUCTION TO QUALITY ASSURANCE FOR EXECUTIVES

(a) Principal elements of an effective quality management system based on the five main elements of any quality assurance programme and other suitable principles, such as those for national quality awards.

(b) Risks in the absence of an effective management system based on:
— case studies;
— the organization’s past experiences;
— client audits;
— any relevant other external assessments;
— relevant financial data.

(c) The executives’ part in any effective management system:
— leadership in defining effective work practices;
— definition of fundamental policies;
— implementation of the quality management system;
— assessing operational feedback;
— directing improvement/modification of the quality management system;
— the purpose of quality assurance manuals, procedures and work instructions.

(d) Workshops to examine current issues such as:
— current organization chart;
— managing computers and software;
— maintaining quality assurance manuals, procedures and work instructions;
— financial reporting to identify problems and measures of success;
— preservation of technical know-how;
— defining, documenting and implementing effective work process;
— cost estimating processes.

45 The five main elements of any quality assurance programme are:
(i) Define the organization, roles and responsibilities;
(ii) Define work practices;
(iii) Define verification, inspection and testing practices;
(iv) Define audit and monitoring practices;
(v) Define processes for improving the quality assurance programme.
(e) What's in it for the executives:
   — manageable personal workload;
   — improved staff morale;
   — improved financial performance;
   — fewer crises to manage;
   — few surprises;
   — orderly problem solving;
   — confidence to approve documents containing management directives;
   — improved performance evaluations;
   — recognition in the market place.

INTRODUCTION TO QUALITY ASSURANCE FOR STAFF

(a) Principal elements of an effective quality management system based on the five elements described above and other suitable principles, such as those for national quality awards.

(b) Overview of the organization's quality assurance programme based on the organization's quality assurance manual.

(c) Review of standard work practices and procedures in their area.

(d) Learning about the organization's technology using appropriate proprietary guides and research reports.

(e) What's in it for staff:
   — manageable personal workload;
   — getting it right the first time;
   — pride in work;
   — ready identification of relevant technical data;
   — high morale;
   — recognition of technical competence;
   — improved performance evaluations.
Appendix VII

CHARACTERISTICS OF DEFECT PREVENTION PROCESS

The defect prevention process consists of four key elements:

1. *Causal analysis meetings.* Brainstorming sessions conducted by technical teams at the end of each stage in a development process. Defects found during the stage are analysed, root causes are identified, and possible actions are suggested to prevent similar errors from recurring. Methods for removing similar defects in a current product are also discussed. Team members discuss any overall defect trends that may emerge from their analysis of this stage, particularly what went wrong and what went right, and examine suggestions for improvements. After the meeting, the causal analysis leader records the data (defects, causes and suggested actions) in an action database for subsequent reporting and tracking. To allow participants at this meeting to more freely express their thoughts and feelings on why defects occurred without jeopardizing their career or next appraisal, management is not present during this meeting.

2. *Action team.* The action team is responsible for screening, prioritizing and implementing suggested actions from causal analysis meetings. Each member has a percentage of his/her time allotted for this task. Each action team has a co-ordinator and a management representative (the action team manager). The team uses reports from the action database to guide their meetings. The action team is the engine for the process. Other than action implementation, the team is also involved in feedback to the organization, reports to management on the status of its activities, publishing success stories and taking the lead in various aspects of the process. The action team relieves the programmers from having to implement their own suggestions, especially actions that have a broad scope of influence and require substantial resources. Of course, the existence of the action team does not preclude action implemented by others. In fact, technical teams are encouraged to take improvement actions, especially those that pertain to their specific areas.

3. *Stage kick-off meetings.* The technical teams conduct these meetings at the beginning of each development stage. The emphasis is on the technical aspect of the development process and on quality. What is the right process? How do we do things more effectively? What are the tools and methods that can help? What are common errors to avoid? What improvements and actions have been implemented? The meetings thus serve two main purposes: as a primary
feedback mechanism of the defect prevention process and as a preventive
measure.

4. Action tracking and data collection. To prevent suggestions from being lost
over time, to aid action implementation and to enhance communications among
groups, an action database tool is needed to track action status.

The process is illustrated schematically in Fig. 6.

**FIG. 6. Defect prevention process.**
THE WATERFALL DEVELOPMENT PROCESS

The so-called waterfall model represents the sequence of activities needed to develop a new piece of software as a linear series of steps beginning with the definition of customer requirements. At each stage an intermediate deliverable is produced that feeds into the next stage. Typical representations of the waterfall model are shown in Figs 7 and 8.

The output from each step is subject to various verification and testing activities to ensure that it conforms to the requirements defined in the previous step. This "entry-task-verification/validation-exit" paradigm is a key characteristic of the waterfall model.

Advantages of the waterfall model include:

— Provides a conceptually simple model;
— Encourages the development team to establish first what the software is supposed to do (define system requirements);

**FIG. 7.** Software life-cycle phases: simplified diagram for monitoring, control and safety software.
— Breaks the development effort into several logical steps such as design, coding and testing with corresponding intermediate deliverables that lead to the final software product;
— Applies various verification and validation techniques to ensure that each step is executed properly with good quality deliverables;
— Tracks project progress accurately and uncovers possible slippages early;
— Encourages a structured and disciplined approach by the development organization;
— Stipulates the preparation of key documents that can be used later to test and maintain the software;
— Makes complex software projects more manageable and reduces the risk of cost overruns.

Disadvantages of the waterfall model include:

— Represents a complex set of processes with many iterations and feedback processes during various verification and testing stages where corrections are fed back to the output from the previous stage;
— Accommodates significant changes to requirements during the progress of software design and development;
— Requires an understanding in depth of actual software development practices to ensure that an appropriate number of iterative and feedback processes are part of the development life-cycle.

FIG. 8. Software life-cycle phases: simplified diagram for engineering and analysis software.
PROTOTYPING APPROACH

Sometimes requirements change significantly between the time when the specification of the software system is finalized and when the development of the software product is complete. Sometimes the requirements are not well understood at the start of the development activity. The waterfall model is not well equipped to deal with such circumstances since it relies heavily on the complete definition of requirements early in the development process.

Under such circumstances a prototyping approach can be adopted. A prototype is a partial implementation of the software product with as many features as are known or anticipated at the start of the development process. Potential customers use the prototype and provide feedback to the development team before full scale development begins. With this approach the customers and the development team can clarify both requirements and their interpretations.

As shown in Fig. 9, the prototyping approach usually involves the following steps:

1. Requirements gathering and analysis.
2. Quick design.
5. Refining design and prototype.
6. If customers are not satisfied with the prototype, return to step 2.
7. If customers are satisfied, begin with full scale development.

The key to success in using the prototyping approach is quick turnaround in designing and building the prototypes. This can be accomplished using reusable software parts. Formal specification languages (Z notation, Input/Output Requirements Language can be used to generate executable code.

Advantages of the prototyping approach include:

— Quick implementation of a software component;
— Evolving identification of requirements;
— Suitability for small development tasks such as at the subsystem level.

Disadvantages of the prototyping approach include:

— Lack of formal specification languages most useful for prototyping;
— Requirement of continuing user feedback to complete requirements identification;
— Difficulty in applying prototyping to complete and complex software projects;
— Difficulty in identifying when the specification of requirements is complete and therefore when to stop iterating.

SPIRAL MODEL

The spiral model of software development is a refinement of the waterfall model. It relies on prototyping and risk management and is more flexible than the waterfall model. The model is illustrated in Fig. 10.

The basic concept of the model is that for each portion of the software product and for each of its levels of elaboration, the same sequence of steps is involved. The radial dimension in Fig. 10 represents the cumulative cost incurred in accomplishing the steps. The angular dimension represents progress made in completing each cycle of the spiral. As indicated by the quadrants in the figure, the first step of each cycle is to identify the objectives of that portion of the product being elaborated, the alternative means of implementation of this portion of the product and the constraints imposed on the application of the alternatives. Each cycle is completed by a review of the output by key members of the development team.
Some of the advantages of the spiral model are:

— It focuses attention on options involving the reuse of existing software because the identification and evaluation of alternatives is one of the key steps in each cycle.
— It accommodates preparation for life-cycle evolution, growth and changes of the software product.
— It provides a mechanism for incorporating software quality objectives into software product development.
— It helps identify and eliminate errors and unattractive alternatives early.
— It does not involve separate approaches for software development and software enhancement.
— It provides a framework for integrating hardware–software system development.

Some disadvantages are:

— Software developed under contract relies heavily on control, checkpoint and intermediate deliverables. Compared to the waterfall model, the challenge is to preserve the spiral model's greater flexibility without losing accountability and control for contract software.
— It requires more experienced development staff with appropriate risk management expertise.
— For less experienced staff and for large scale projects, the steps in the spiral must be further elaborated and more specifically defined to achieve consistency, tracking and control.

ITERATIVE DEVELOPMENT PROCESS MODEL

The iterative development process model begins with a subset of the requirements and leads to a subset of the software product that meets the essential needs of the users, provides a means for analysis and training of customers, and provides a learning experience for the developers. In a series of iterations, the design and the requirements are then modified to provide a system to the users that meets evolving customer needs as identified from feedback and learning. The iterative development process model combines prototyping with the classical waterfall model. The nature of the iterative development process model is shown in Fig. 11.

The iterative development process model illustrated in Fig. 11 typically contains eight major steps:

(1) Domain analysis;
(2) Requirements definition;
(3) Software architecture;
(4) Risk assessment;
(5) Prototype;
(6) Test suite and environment development;
(7) Integration with previous iterations;
(8) Iteration release.
The iteration process involves the last five steps. During these iteration steps the following activities are carried out:

- Analyse/review the system requirements;
- Design/revise a solution that meets the known requirements;
- Identify and assign priorities to the highest risk areas for the project; use prototyping to address the highest priority risks; leave lower risks for later iterations;
- Define and plan the next few iterations;
- Develop the iteration test suite and test environment;
- Implement the minimum amount of design needed to meet the requirements for the current iteration;
- Integrate the software in the test environment and perform regression testing;
— Revise supporting documentation for release with the iteration;
— Release the software at the current iteration and its supporting documents.

The advantages and disadvantages of the iterative development process model are similar to those of the prototyping approach. The nature of the iterative development process is similar to that of the prototyping approach.

CLEANROOM METHODOLOGY

Cleanroom software engineering approaches software development as an engineering process with mathematical foundations rather than a trial and error
programming process. Cleanroom management is based on incremental development and certification of a series of user function increments that accumulate into a final product. Cleanroom operations are carried out by small, independent development and testing teams. Figure 12 illustrates the process.

Advantages of the cleanroom methodology include:

— The bases of the process are proof of correctness of design and code and formal quality certification via statistical testing.
— All testing is based on anticipated customer usage. Therefore, errors that are likely to cause frequent failures to the users are likely to be found first.
— Software quality is certified in terms of mean time to failure.

Disadvantages of the cleanroom methodology include:

— Its adoption is so far mostly confined to small projects.
— Its ability to be scaled up to large projects and the mathematical training required remain unanswered questions.
Appendix IX

RECOMMENDATIONS FOR DESIGN INPUT DOCUMENTATION FOR MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

This appendix provides recommendations for the content and format of design input documentation (DID) to facilitate the software engineering process for 'real time' software. The DID should be complete, consistent and unambiguous. In particular, it should:

(a) Divide the system so that:
- the software subsystem important to safety is isolated from other subsystems;
- the software functionality which does not directly support meeting the minimum performance specification for the software important to safety is kept to a minimum.

(b) Define all functional, performance, safety, reliability and maintainability requirements of the subsystem, clearly identifying the safety requirements.

(c) Identify each computer within the computer system and include a description or reference to the computers’ characteristics, such as memory capacity, instruction sets, speeds and input/output registers.

(d) Define all details of the interfaces with external inputs and outputs.

(e) Define all accuracy requirements and tolerances.

(f) Define all failure modes (identified by hazards analysis of the system), and the appropriate response to them, including any degraded operation modes required.

(g) Define any constraints placed on the design options.

(h) Take into account any relevant experience from previously developed systems.

(i) Limit and localize the use of complex calculations upon which safety critical decisions depend.

(j) Avoid any requirements that are in direct conflict with each other.

(k) Provide a clear definition of terms.

(l) Identify foreseeable changes and enhancements to the system.

(m) Define each requirement uniquely and completely in one location to prevent inconsistent updates and to facilitate easy referencing by other documents and by verification processes.

(n) Include or reference a revision history for all documents making up the DID.
Appendix X

RECOMMENDATIONS FOR SOFTWARE DEVELOPMENT PLANS APPLICABLE TO MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

The software development plan (SDP) should provide information about the management approach to the software development project.

The SDP typically contains information on:

— the identity of applicable input documents to the software development project;
— the development and programming practices to be followed;
— the organization of the development team;
— interfaces within the development team and between the team and others who will have to work with the software to be developed;
— schedule and resource needs;
— project milestones;
— project personnel loading, including skills and expertise requirements;
— any special hardware and environmental needs, such as computer time, special test facilities, compilers, linkers and other systems;
— provision for involvement in testing of those responsible for interfacing systems, such as reactor monitoring and control systems.

Particularly for substantial development efforts, the SDP can be subdivided into a number of major sections such as:

— Development, verification and support plan
— Documentation plan
— Configuration management plan
— Training plan.

The development, verification and support plan identifies the design input documentation and describes the scope of the software engineering effort, the organization and responsibilities, the expected resource loading and the software engineering life-cycle model. It identifies key milestones and dates reflecting budget and schedule documentation used to monitor the progress of the work. It describes the approach to the development and the key design and implementation issues, and identifies the support tools and facilities.
The documentation plan provides an index and summary descriptions of all project documentation. It also identifies responsibilities for producing, reviewing, approving and issuing all project documents.

The configuration management plan identifies the configuration management activities, the organizational structure and independence requirements between developers, verifiers and validators. It also identifies configuration management support tools and facilities.

The training plan identifies the training requirements, the training organization and training tools, resources and documents.
Appendix XI

RECOMMENDATIONS FOR STANDARDS AND PROCEDURES HANDBOOKS APPLICABLE TO MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

Standards and procedures handbooks can be structured into four main subheadings:

— Development, verification and support standards and procedures;
— Documentation standards and procedures;
— Configuration management standards and procedures;
— Training standards and procedures.

The development, verification and support standards and procedures typically:

(a) Identify standards for the preparation, review, approval, distribution and revision of the SDP and the standards and procedures handbook (SPH);
(b) Identify procedures for using tools and facilities during the development, verification and support processes;
(c) Identify the standards and procedures to ensure that all development, documentation, testing and commissioning processes and outputs meet established requirements;
(d) Specify criteria and guidelines for the generation of test cases to achieve a known and consistent level of coverage;
(e) Define checklists and guidelines for requirements, design and code review to achieve a consistent and known level of review;
(f) Define the guidelines and methodologies for the performance of each verification process;
(g) Identify all verification tools, techniques, checklists, procedures and databases, their prior use and current versions;
(h) Require procedures for ensuring that the part of the software not affected by change will still work correctly following a change implementation;
(i) Define procedures for the formal design review of the software requirements specification (SRS);
(j) Define procedures for the formal design review of the software design description (SDD);
(k) For systematic design verification, define transformation rules which convert either the SRS or the SDD to a notation which is directly comparable with the other;
For systematic code verification, define transformation rules which convert either the SDD or the code to a notation which is directly comparable with the other.

The documentation standards and procedures typically:

(a) Define standards for all documents defined in the documentation plan portion of the SDP. These standards should ensure that documentation produced by different individuals has a uniform format and structure. The format should be chosen to make the documents readily understandable, reviewable and

FIG. 13. Example of a personal qualification record form.
maintainable. The standards and procedures should provide guidance on the appropriate format for revision history record, document table of contents, labelling of figures and tables, page identification, section identification, cross-referencing between related documents, glossary, index, references, date, signatures and file numbers.

(b) Define a comprehensive glossary to ensure a common understanding and consistent usage of project specific and industry accepted terminology. The use of unique terminology and definitions should be minimized.

The configuration management standards and procedures typically:

(a) Specify guidelines to handle and document exception cases or change requests and their resolution arising from errors or exceptions which are detected during software development and verification (i.e. any non-conformances to the SDP or the SPH, errors found during all verification activities, changes arising during installation and operation, tests deferred to a later testing phase, etc.).

(b) Identify a procedure for ensuring that all personnel are working from the most current version of the documentation.

(c) Identify procedures for unique identification of software changes, software subsystems, release, version and update designations, software subsystem component names, documentation and media.

(d) Define procedures for issuing, analysing, estimating the impact of, classifying, approving, distributing, scheduling implementation of and tracking software change requests to completion. The procedures should require that the activities necessary to verify the change be identified and justified.

(e) Define procedures for identifying all implemented software change requests and all configuration items and revision numbers for all software releases.

(f) Identify procedures for support library operations (i.e. security, disaster recovery, archive maintenance, protection, access control, retention period, change history, withdrawal, submittal, issue and approval).

(g) Define standards and procedures for change request documentation, error analysis summary reports and configuration management status reports in order to identify the most current configuration and to provide a traceable history of software changes.

(h) Define procedures for approving and issuing software releases to the user and for confirming and documenting that the software delivered is identical to that approved for release.

An example of a personal qualification record form is given in Fig. 13.
Appendix XII

RECOMMENDATIONS ON THE CONTENT OF SOFTWARE REQUIREMENTS SPECIFICATIONS FOR MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

The software requirements specification (SRS) should meet the following criteria to ensure that the resulting software satisfies safety, functionality, reliability, reviewability and maintainability acceptance criteria.

To achieve completeness, the SRS should:

— Contain or refer to all relevant requirements from the design input documentation (DID), including functional, performance, safety, reliability and maintainability requirements;
— Specify any additional requirements and software implementation constraints;
— Identify those variables of the physical environment that the software must monitor and/or control and represent them by mathematical variables;
— Define the behaviour of the entire set of controlled variables in terms of monitored variables using mathematical functions;
— Specify the timing tolerances and the accuracy requirements by specifying the allowable deviation from the specified behaviour of the controlled variables;
— Identify the software interfaces to the hardware and/or other software in terms of input and output mathematical variables;
— Describe the characteristics of the input/output variables addressing issues such as variable types, formats, units of measurement, valid ranges and method of access by the software;
— Describe the relationship between the monitored and input variables and the relationship between the controlled and output variables;
— Define the required response to all expected types of errors and failure modes identified in hazard analyses described in the DID; indicate the cases for which error recovery must be attempted or for which fail-safe action must be taken;
— Define software reliability requirements consistent with the reliability requirements of the hardware system that the software is designed to control;
— Define all those requirements for which future changes are anticipated.

To meet correctness expectations, the SRS should:

— Conform to the specified format for such documents;
— Address the requirements and design constraints in the applicable DIDs;
— Provide justifications for any requirements in addition to those specified in the DIDs.
To be self-consistent, the SRS should:

— Not contain any requirements that are in conflict with each other;
— Use standard terminology and definitions throughout the document.

To be readily verifiable, the SRS should describe any mathematical functions using a notation which has a precisely defined syntax and semantics.

The SRS should define each unique requirement only once in order to prevent inconsistent or conflicting requirements and revisions to them.

The requirements in the SRS should be traceable to information in the DID. Therefore, the SRS should:

— Use a coverage matrix, cross-reference table or other suitable representation to demonstrate explicitly the mapping and complete coverage of all requirements and design constraints recorded in the DIDs;
— Identify each requirement uniquely so that the software design description can refer to it in a clear and unambiguous way;
— Include references to any notes which document design decisions relevant to the software requirements.

The SRS should be written in a notation which the intended audience, including those responsible for the DIDs, the developers, verifiers, users and maintainers, can interpret consistently and unambiguously.

To ensure the robustness of the monitoring or control system, the SRS should define requirements for fault tolerance, fault recovery and graceful degradation.
Appendix XIII

RECOMMENDATIONS ON SOFTWARE DESIGN DESCRIPTIONS FOR MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

The software design description (SDD) should meet the following criteria to ensure that the resulting software satisfies safety, functionality, reliability, reviewability and maintainability acceptance criteria.

To achieve completeness, the SDD should:

(a) Describe a design which meets all functional, performance, safety, reliability and maintainability requirements, and design constraints as described in the software requirements specification (SRS).

(b) Define the types of errors and the software’s response to them.

(c) Provide a level of detail in the description of the design that requires no further refinement of the structure of the software during the coding phase.

(d) Identify all modules, data structures and databases which make up the design, describing their purpose, the programs contained in each module and the hierarchical structure of all programs making up the design.

(e) Provide the following information about each program:
   — unique identifying name;
   — program type (e.g. main program, subroutine, function, macro instruction);
   — the purpose of the program and what requirement it satisfies;
   — the function of the program;
   — the subordinate components comprising the program to assist in tracing requirements;
   — relationships with other programs, such as interactions, data flow, timing, order of execution, data sharing;
   — interfaces with other programs and hardware systems, such as methods of invocation or interruption, communication, database access, message passing, data formats, ranges of validity;
   — resources needed to run the program, such as hardware devices, software libraries, operating system services, processing resources;
   — local data definitions including data type, initial values, valid ranges;
   — the programming language to be used.

(f) Consider relevant experience from previously developed similar systems.

To ensure correctness, the SDD should:

— Conform to an agreed documentation standard;
— Limit the design to the requirements specified in the SRS;
— Define the function of each program for the full range of each program input.

In order to promote computer programs whose behaviour is predictable, the SDD should:

— Describe the design of programs which provide the required response to all identified conditions;
— Define a method of scheduling computer resources which is mostly deterministic and predictable;
— Limit the use of interrupts and event driven software;
— Describe design features that allow checking on the execution of programs to uncover errors associated with the frequency and/or order of program execution.

To achieve a robust computer program, the SDD should address all requirements contained in the SRS related to fault tolerance and graceful degradation.

The SDD should use standard terminology and definitions throughout the document.

The SDD should describe a design that follows a logical hierarchical structure. Control should pass from the highest to successively more detailed levels. Control should always return to the calling program except for error/exception handlers.

In order to facilitate verification, the SDD should:

— Describe the function of each program using a notation that has a precisely defined syntax and semantics;
— Describe a design that limits reliance on a program’s ‘long term memory’ and calls for periodic re-initialization of variables to make random testing feasible.

To facilitate modifications of the computer program, the SDD should describe a design where:

(a) The software is organized into modules so that changes in requirements can be implemented by requiring changes to only one or a limited number of modules;
(b) Those functions and data structures likely to change are designed with interfaces that are insensitive to changes in individual functions;
(c) Data structure access, database access and input/output access are separated from the application software using access programs (but where globally accessible data are avoided);
(d) Each program has a single function minimizing program coupling.
The statements in the SDD should be traceable to information in related design and development documents. Therefore, the SDD should:

— Demonstrate the mapping and complete coverage of all requirements and design constraints in the SRS using a coverage matrix, cross-reference table or other suitable document format;
— Identify each function in the design so that it can be uniquely referenced in the coding;
— Contain or reference a revision history which identifies all modifications to the design and the rationale for these changes;
— Reference any design notes which document design decisions relevant to the software design.

The SDD should promote a modular computer program design. Therefore, the SDD should describe a design:

— Which is structured so that it comprises relatively small, hierarchically related programs, each performing a particular unique function;
— Which uses specific criteria to limit the program size.

The SDD should be written in language that is easy to understand. Therefore, the SDD should:

— Avoid unnecessarily complex designs and design representations;
— Be written using design description techniques which suit the various aspects of the design to allow unambiguous interpretation.
Appendix XIV

RECOMMENDATIONS ON DESIGN AND DEVELOPMENT DOCUMENTS
FOR DESIGN, ENGINEERING AND ANALYSIS SOFTWARE

Problem definition

The problem to be solved by the computer program should be documented. This description should include as much information as possible on:

— the physical phenomena to be analysed;
— the geometry of the reactor systems;
— identification and properties of interacting materials;
— events or states to be analysed;
— conditions to be analysed;
— theories most closely describing the physical phenomena.

Software development plan

The elements of the software development plan (SDP) should be documented. The plan should include all the development phases and milestones required to carry out the development of the computer program. It should include information on the model development, design, coding, testing, verification and validation tasks to be performed during the development of the program. The SDP should be amended, as necessary, during the progress of the work in order to keep it current at all times.

SDPs should typically contain the following elements:

— the breakdown of computer program development into manageable tasks and the assignment of related responsibilities, including review and approval authority;
— the sequence and timing of activities to be performed, key milestones and outputs;
— the development tools, techniques and methodologies to be used;
— the review, testing, verification and validation activities to be performed, the methods to be used and the rationale for their selection;
— the means to achieve independence between those performing and those verifying activities;
— the means to resolve non-conformances;
— the computer program components to be developed by subcontractors and the applicable quality assurance programmes;
— the methods to control interfaces between contributors to the computer program development, including customers;
— the identification of the documents to be produced as part of the development process, a description of their purpose and content, and the identification of responsibility for producing, reviewing and approving documents;
— the identification of the configuration management methods;
— if an existing computer program is being changed, a specification of the version to be modified and a proposed identification of the new version.

Software requirements specification

The software requirements specification (SRS) gives an unambiguous list of essential, verifiable requirements for the computer program development. The SRS should identify the essential verifiable requirements as a minimum.

The SRS should provide the following information:

— the name of the computer program;
— the functions of the computer program;
— hardware, software and computer program user interface requirements;
— operating system requirements;
— computational speed requirements;
— portability requirements;
— file size and type requirements;
— input and output requirements;
— data structure and data flow requirements;
— programming language;
— imposed physical or mathematical models or numerical algorithms;
— error detection and handling requirements;
— accuracy targets;
— requirements on programming practices.

Design description

The software design description (SDD) describes how the specified requirements are to be met. The SDD should provide information how the specified requirements are met.

In addition, the SDD should provide the following information:

— identification of the algorithms;
— computer program structure, such as data structures (to the appropriate level of detail), program flow, database design, object descriptions and packaging of code;
— description of modules and module interfaces;
— library functions.
Verification reports

Verification reports should be prepared to document verification activities identified in the software development plan as well as program verification cases. One or more verification reports should be produced for the SRS, the SDD and the actual coding, including test cases. It may be convenient to combine these into a single document that may be called a verification manual.

Verification reports should include:

— identification of the verification plan (which gives requirements against which items are verified);
— identification of the methods and verification criteria;
— a description of the verification exercises;
— an assessment of the results of the verification exercises with respect to the criteria and disposition of anomalies.

A verification report for test cases documents the verification test suite and the results of its application. The verification cases confirm that a new computer program or new version of a computer program produces results consistent with the SRS and the SDD for a new program, or consistent with those of successful previous runs of an existing program to within pre-defined acceptance criteria.

The following should be included in this type of verification report:

(a) Verification criteria. The verification criteria make up a set of comparison conditions which, if satisfied, imply that the new version functions properly. The verification criteria which are described might indicate, for example, that only the following differences are acceptable:
— differences in date and time;
— differences which can clearly be attributed to, and explained by, changes made to the coding;
— differences which can easily be attributed to changes in the compiler/interpreter, computer architecture, operating system, run-time libraries and/or other system software components related to the computer system.

(b) Test suite. This consists of the input files and the batch processing file which executes the computer program. The runs should be listed and briefly described. Further details of the verification set should be given in appendices. This may include hard copies of the main user input files and the batch processing files.

(c) Verification results. A brief description of the results of the comparisons of the verification outputs should be given. The reasons for any significant differences should also be given. Further details should be given in an appendix, including hard copies of electronic comparisons. All differences should be reported unless
they are repetitive. In this case, a single typical or abridged output of the electronic comparison may be sufficient.

It may be convenient to prepare a verification manual to provide an overview of the verification activities if a number of verification reports have been produced in the verification of a computer program version. A verification manual may include:

— a list of the verification activities and verification reports;
— the associated verification criteria;
— an assessment of the results of each exercise with respect to its verification criteria.

**Programmer's manual**

The programmer's manual contains information useful to the developer for understanding and modifying the computer program. The following information should be included in the programmer's manual:

— *Program flow and structure*. Information about program flow and structure is usually prepared in the programmer's manual after coding is complete. Information about program flow and structure is usually also provided in the design description before any coding is prepared. This information may appear in the two documents at different levels of detail. However, the information should be consistent. Wherever possible, the information can be contained in one document and referred to in the other.

— *Translation of theory into coding*. The programmer's manual includes information on how numerical techniques and programming techniques are implemented. This and other considerations in translating the theory from the theory manual into a viable computer program should be described.

— *How to modify and maintain the program*. Information useful for future modifications and maintenance of the computer program should be given. This includes descriptions of any relevant programming and debugging tools. It may be useful to discuss error and warning messages in this context.

— *Programming conventions*. Conventions on programming practices such as variable naming and array structure should be included. Programming philosophy should also be mentioned, in particular with regard to computer program commentary.

There may be technical notes or marked-up hard copies that form an essential part of the programming process. Such notes or the mark-ups can be attached to either the SDD or the programmer's manual.
Appendix XV

RECOMMENDATIONS ON APPLICATION DOCUMENTS FOR DESIGN, ENGINEERING AND ANALYSIS SOFTWARE

Computer program abstract

The computer program abstract provides a concise summary of the purpose, capabilities, operating environment and limitations of the computer program. The computer program abstract is a primary reference for the user community. It should include the following information:

— the computer program name;
— the version identification;
— a brief description of the problem solved;
— the applicable configuration components (e.g. hardware and system requirements);
— method of solution and range of application;
— capabilities and features;
— limitations and restrictions;
— references and list of available manuals (including revision numbers).

Theory manual

The theory manual describes the theoretical and mathematical foundations of the problem being solved by the computer program. It should also provide an overview of the program and the needs it meets in the nuclear industry.

The theory manual should include the following information:

— the theory and mathematical equations;
— assumptions and constraints;
— solution techniques, and the rationale for selecting these techniques, such as accuracy requirements and other limitations;
— any empirical correlations, their range of application and associated uncertainties;
— applicable existing references.

The theory manual should be written for a wide audience, including regulators, present and potential customers, users and developers. It may also be useful to include a nomenclature and a keyword index to facilitate navigation around the document.
User's manual

The user's manual contains the information necessary for the user to effectively execute the computer program. It should be sufficiently detailed to permit qualified personnel to prepare input, to run the computer program, and to understand the output. It assumes that the user understands the theoretical basis of the computer program. The user's manual should clearly state the computer program name and version identifier. It should also include the following information:

— instructions on installing and running the computer program;
— description of features, capabilities and options;
— description of input and output;
— description of error and warning messages, their interpretation and recommended corrective action;
— identification of embedded and default values;
— description of limitations and restrictions;
— sample cases that illustrate the use of components and modules;
— a list of references and other related documentation;
— the use of auxiliary input preparation and output routines supplied with the program;
— a discussion of the sources of computational error.

If there are forms for problem reports or enhancement requests, they may also be given here.

Validation report

The validation report provides a measure of the accuracy of the computer program for the intended application. It documents the results of the validation activities performed. Validation reports should provide the following information:

— a statement of the application for which the computer program is being validated;
— a description of the methods used;
— identification of data against which validation was performed;
— computer program input and output;
— validation results;
— assessment of validation results with respect to program accuracy and uncertainty allowances.
It may be convenient to prepare a validation manual to summarize the results documented in the validation reports. A typical validation manual should include the following information:

— introduction, giving the context of the compendium and a brief description of the computer program;
— a summary of the validation documents available;
— a summary of the validation results (this may be by phenomenon);
— sensitivities and uncertainties;
— general conclusions from the validation results.

**Version tracking record**

The version tracking record summarizes the differences between configuration versions. It should contain sufficient information to allow the retrieval and execution of any computer program version archived in the software library.

For each configuration version, the following information should be included in the version tracking record:

— identification of the version that was modified and of the new version;
— reasons for the change;
— significance of the change and the basis for the categorization of the change;
— identity of those who made the change;
— release date of the new version;
— modified or added computer program components;
— description of changes and reference to documents containing more detailed information, where appropriate;
— methods used to verify and, as appropriate, validate the new version;
— location where the computer program is archived;
— list of all computer program documentation associated with the new version;
— list of operating system, compiler, library functions, object modules and other relevant details necessary to obtain identical program execution.

For each change to a version, a reference to the following information should also be included:

— the version of the programming language;
— the medium on which the archived computer program files are stored.
Appendix XVI

SUGGESTED GOOD CODING PRACTICES FOR DESIGN, ENGINEERING AND ANALYSIS SOFTWARE

The following describes one possible set of principles on which to base good programming practices:

(a) Program organization
   (i) Clear and easy-to-follow overall program structure;
   (ii) Overall program control centralized in a single module;
   (iii) Orderly progression of calculation flow;
   (iv) Single, well defined function for each module;
   (v) Uniform module layout;
   (vi) Input data acquisition centralized in a single program unit (or in a single program unit per module in a multi-module computer program);
   (vii) Display and saving of major calculation results.

(b) Programming language
   (i) Adoption of a standard programming language;
   (ii) Adherence to an applicable programming standard;
   (iii) Limited use and thorough documentation of extensions to the standard programming language;
   (iv) Use of assembly language only where absolutely necessary.

(c) Data transfer
   (i) Use of a uniform data transfer technique throughout, unless a data structure requires special handling;
   (ii) Documentation of data transfer technique(s).

(d) Input/output
   (i) Input data identification, preparation, organization;
   (ii) Input checking;
   (iii) Echoing of input data for visual inspection;
   (iv) Echoing of computer program version identification in output;
   (v) Reports on calculational path, intermediate results and calculation progress;
   (vi) Issuing of warning and error messages in the event of abnormal conditions;
   (vii) Adoption of user friendly input and output formats and user’s options;
   (viii) Use of a single module to read or write a data file;
   (ix) Documentation of data file content and structure.

(e) Source statements and variable names
   (i) Specification statements for variables;
   (ii) Adoption of naming conventions and/or meaningful variable names;
   (iii) Liberal use of comment statements;
(iv) Single purpose use of variables;  
(v) Checking of parameter types.

(f) Hardware and operating system dependencies  
(i) Independence from unique hardware and operating system features;  
(ii) Documentation of dynamic memory management techniques.

(g) Other recommended programming techniques  
(i) Initialization of arrays, variables and default parameters;  
(ii) Clear identification of range, beginning and end of program loops;  
(iii) Checking of array bounds;  
(iv) Unambiguous branching decisions in the computer program;  
(v) Standards on the use of subscripted variables;  
(vi) Avoidance of mixed mode operations;  
(vii) Standards for structuring arithmetic and logical expressions for clarity;  
(viii) Testing of intermediate and final results for validity;  
(ix) Standards on the use of nested constructs;  
(x) Manageable module sizes;  
(xi) Limitation of module parameters to the minimum number required;  
(xii) Standards for module and loop entry and exit points;  
(xiii) Standards on computer program readability;  
(xiv) Standards on efficient and reliable programming techniques.

Code generation may be by means of an automatic tool which operates to produce the code from a graphical representation of the requirements on a video display screen. When this is the case, suitable assurance is necessary that the modules used to encode each potential graphical element have been validated. The method and code used to assemble the modules into a complete integrated code body should also be shown to be validated. This may be by a generic demonstration made and recorded by the supplier or preferably by specific validation tests of the code produced. The method used should allow the application information to be recorded and held in a suitable configuration management system, to record and control issue and authority for use.
Appendix XVII

RECOMMENDATIONS ON PROGRAMMING OF MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

The computer programs should meet the following criteria to ensure that the resulting software satisfies safety, functionality, reliability, reviewability and maintainability acceptance criteria.

The computer program should reflect the design completely. Therefore, the computer program should:

— implement the design precisely as documented in the software design description (SDD);
— create all required databases complete with initial data;
— be integrated and debugged to satisfy the design as specified in the SDD;
— contain no unreferenced or undefined variables, constants or data types.

To ensure correctness, the computer program should:

— conform to applicable programming standards;
— meet the accuracy requirements for all variables for the chosen data types and algorithms;
— contain meaningful and accurate comments;
— have data declarations that are consistent with their use;
— invoke all programs with the correct number of parameters.

The computer program’s responses should be predictable. Therefore, it should:

— be written in a programming language which has a well defined syntax and semantics;
— contain no self-modifying code;
— be written following guidelines which promote static control logic;
— not rely on defaults provided by the programming language used;
— not contain unintended infinite loops;
— not contain recursions.

The computer program should provide protection against detectable run-time errors such as out-of-range array index values, division by zero, out-of-range variable values and stack overflow.

The computer program should use the same format, invocation convention and structure as defined in the specified programming standard and practices.
The program structure should be clear. Therefore, the computer program should be written so that:

— each function of the program is a recognizable block of code;
— loops have only one entrance.

To facilitate verification activities, implementation practices and techniques which are difficult to test should be avoided.

To facilitate future modifications with a minimum risk of error insertions, the computer program should:

— refer to constants symbolically to facilitate change;
— include cross-references or data dictionaries showing variable and constant access by program;
— consist of programs which have only one entry point and one exit point (except for fatal error handling);
— refer to labels or other symbolic constants rather than addresses.

In order to make it easy to relate the computer program to its design documentation, the computer program should:

— identify each program module uniquely;
— provide a cross-reference framework through which the computer program can be traced directly to the SDD;
— contain or reference a revision history of all computer program modifications and the reasons for them;
— reference the design notes which document design decisions relevant to the computer program implementation.

The computer programming should be easy to understand. Therefore, the computer program should:

— contain ample comments using clear language phrases to describe the physical process;
— be formatted to enhance readability using techniques such as consistent indentation to show nesting of structures and using adequate “white space” to make the structures stand out clearly;
— use mathematical equations which correspond to the mathematical models as described/derived in the SDD;
— provide clear and detailed comments on non-standard or complex programming logic;
— use comment blocks to highlight the overall flow of program logic;
— define identifier names based on naming conventions defined in applicable programming standards and practices;
— identify the valid range of each variable.
Appendix XVIII

DISCUSSION OF VERIFICATION AND VALIDATION METHODS

Overview

Verification methods should be applied to:

— the software requirements specification;
— the documents produced during the computer system design;
— the software design;
— the implementation of the design during the coding phase of the computer program.

A common form of verification and validation involves manual reviews and inspections of documents and computer program coding. Other forms of verification and validation should also be applied and may be essential to achieve the required assurance of quality. For instance, some kinds of software designs can be mechanically checked for conformity to design rules, and static analysis (e.g. dataflow analysis) may be applied to source code, as well as static-semantic checking (e.g. type-checking) in the course of coding and testing the executable computer program.

An analysis of features added during the various design stages should be carried out to confirm that the desired functionality is achieved by injection of appropriate test signals at the system boundary. For example, such an analysis can be applied to confirm correct fail-safe actions or appropriate responses to out-of-range input signals.

A traceability analysis should be performed to demonstrate that the validation requirements (i.e. for test or evaluation) are complete with respect to the computer system requirements specification.

The system hardware subjected to validation testing should be fully representative of the final computer based system configuration as installed at the site. Similarly, the software being tested should be identical to that installed at the site. The system should be subjected to a wide range of static input and dynamic input tests. It is important to exercise all systems using realistic scenarios involving a full range of inputs (dynamic testing). The dynamic tests should be based on an analysis of the plant transients induced by postulated initiating events.

However, because it is not reasonably practicable, nor is it safe, to test the behaviour of a safety system using real accident scenarios on the plant, the system should be tested using a simulation of the accident scenarios. A test harness should be used to record the results.
The test profiles should be representative of the expected plant parameter variations to which the computer system is expected to respond. The number of tests executed should be sufficient to provide confidence in the system dependability.

Consideration should also be given to the possibility of subjecting the computer system to statistical testing. The goal is to produce an estimate of the computer system reliability by subjecting it to a series of statistically valid tests. These tests should be randomly selected from the system's operational input space.

Pure random testing, which implies the use of inputs selected from all possible combinations of inputs, can also be used since it provides a convenient means of generating a large number of test cases which might reveal unexpected computer system responses.

When the set of inputs is not included in the operational input space of the computer system, corresponding tests are defined as stress tests. Such tests are performed to assess the response of the computer system to unexpected or out-of-range conditions that are not envisioned in the software requirements specification. The nature of the response may have important safety implications.

In both the above cases, number and range of tests applied should be related to the computer system reliability requirements. Evidence should be provided demonstrating that the method of error detection is adequate.

Tests should be performed to confirm set point accuracies and hysteresis.

The system operation and maintenance manuals should also be validated during this phase.

The need for a probationary period of computer based system operation prior to commissioning should be considered. During this period the system should be subjected to routine test and maintenance activities.

A log should be kept of any revealed faults and appropriate corrective action taken.

**Reviews, walkthroughs, inspections, audits**

Manual verification methods such as reviews, walkthroughs, inspections or audits should be applied to the verification in all life-cycle phases. They may be the only applicable techniques for verification of output from those phases prior to source code generation. Since these are manual methods it is important that the means are considered by which the results of such reviews are to be recorded.

These verification techniques are applicable to both in-house verifiers and to third party assessors.

Since these techniques rely on manual methods it is important that the format by which the results of such reviews are recorded be considered. Checklists can be
used. However, care should be exercised in the construction of checklists, e.g. the required response to a checklist item should be clear and not open to interpretation. The means by which verifiers are to record the results of their reviews should be stated in the verification plan together with a justification of the chosen method.

An inspection of the software design specifications should be performed before the design of the test cases is undertaken for the software test procedures. In this way the design of the test cases can take into account the results of the inspection process.

Particularly where software is introduced from external organizations, a receiving inspection should be performed and documented to confirm that:

- the computer programs and associated documentation represent the versions and document revisions specified;
- the computer programs reside on the specified electronic media;
- computer program versions and supporting document editions are related;
- the integrity of the computer programs is intact;
- the supporting documentation is complete.

Recommendations on audits are given in Ref. [4].

Manual inspection methods should not be used as the sole means of source code verification (e.g. the structural test coverage targets must be met). Any exceptional cases should be fully documented and justified. The reviews should also check that the design and coding standards have been adhered to in the production of the software.

Records of the types and numbers of any anomalies identified during manual verification activities should be maintained. These records should be reviewed to determine whether any lessons can be learned. Appropriate process improvements should be implemented.

More detailed recommendations on manual verification methods are given in Refs [2, 28].

**Static analysis**

Static analysis techniques should be used to build confidence in the correctness of the source code. The following forms of static code analysis should be considered:

- design and coding standards/constraints checking;
- control flow analysis;
- data usage and information flow analysis;
- symbolic execution;
- formal code verification.
If static analyses have not already been employed by the developer in the course of code production, then such techniques should be applied to all software by the verification team. Persons carrying out third party assessments might also wish to make use of static analysis. As part of the system integration verification process, the final version of the software should be subjected to appropriate types of static analysis.

The scope and depth of static analyses applied to the software should be justified in the verification plan. In particular, decisions not to apply any of the above analyses should be justified and documented.

**Formal code verification**

If software requirements have been formally specified then formal code verification can be performed. However, the generation of formal proofs generally requires considerable expertise and therefore the need to have access to competent analysts.

Further guidance on formal verification and program proofs can be found in Ref. [2].

**Testing**

Testing involves an examination of the software while its object code is being executed (typically on a host processor or simulator). As a result the correctness of the software, translation tools and hardware elements may all be challenged during testing.

Challenges to the software under examination can be constructed by designing suitable test cases. In addition a test strategy should be designed which allows an incremental buildup of the full software (e.g. bottom-up or top-down). A typical test programme might involve initial testing of the lowest level module part in the software hierarchy followed by testing the module parts at progressively higher levels.

In this way the lower level module parts which have already been tested can be integrated into the test environment so that they can be accessed by the higher level module parts. In designing test cases for the individual test units (e.g. module parts) and the progressive integration of module parts the topics in the following sections of this appendix should be addressed.

**Test conditions**

As stated in Ref. [4], Section 8, the objectives of the testing activities in the software development life-cycle are to expose latent defects and to provide
assurance that the software performs according to its technical and operational requirements, as specified in the software requirements specification and according to the design documentation. The test activities should therefore be aimed at ensuring that these objectives are achieved in an orderly, cohesive, clear and controlled fashion. Both the test activities and the test results should be formally recorded.

The tools employed in producing software belong to two broad categories:

— Software development tools, whose output (possibly transformed) becomes part of the computer program implementation and which can therefore introduce errors. Code generators, compilers and linkers are examples.

— Software verification tools, which cannot introduce errors (but may fail to detect them). Static analysis tools and test coverage monitors are examples.

Test conditions and acceptance testing should address these circumstances. They should be based on applicable design and safety analysis documents.

It is difficult to fully establish the operating profile for a new system. Also, the operating profile usually changes over time, often as a result of introducing the new system into the environment. Despite these facts, the test environment should be representative of all expected operating conditions.

Acceptance testing is discussed in more detail in Ref. [4].

The test environment should be as representative as possible of the actual operating conditions. However, it is difficult to specify completely the operating profile for a new system. In addition, the operating profile usually changes over time, often as a result of introducing new systems into the environment.

Before either a software development tool or a software verification tool is applied, it is essential to have a precise definition of the tool's functionality:

— For a software development tool, the domain of applicability should be precisely known.

— For a software verification tool, the analyses or checks it performs should be well defined.

In all cases, a tool should have sufficient safety dependability to ensure it does not jeopardize the safety of the end-product. Therefore, a software development tool whose output is used without further review should be of the highest dependability level, although this requirement may be reduced if its output is subjected to additional verification. For a software verification tool the requirements may also be reduced somewhat, on the grounds that its output will be closely scrutinized.

The means by which the fitness for use of the design tools is to be demonstrated should be defined in the verification plan.
Qualification, responsibilities and controls

The teams specifying and undertaking the software verification and validation tests should be independent of the designers/developers. However, those performing the tests should be competent in software development methods as well as testing methods.

Detailed knowledge of the design of the software is required in order to assess the actual test runs against the expected behaviour of the software. This familiarity with a computer program is taken advantage of during the common practice where programmers debug their own programs during development. The programmer has the detailed knowledge about how the software is intended to work, and can quickly scan the changing values of internal variables to see if they match expectations. If discrepancies are found, the source of the problem can usually be localized and identified quickly. Note that while debugging is a common practice to remove programming errors, it cannot be considered an independent testing method.

Project related communication between the testing team and the designers should be recorded. All changes to approved test procedures should be recorded and should be subject to formal approval.

Reverse engineering

In some circumstances it is possible to use reverse engineering techniques to provide confidence in the proper translation, for instance, from a design description to a source code, or from a source code to a machine code. This reverse engineering involves applying in reverse (or inverting) the normal software development process. Its feasibility depends on the original software development process being well defined and traceable, rather straightforward and uniquely invertable. Such reverse engineering has been performed satisfactorily, and even mechanized, for output of so-called code generators from information describing the computer program design. The technique has also been used successfully in reconstructing source code from machine code where the source has been fairly low level. The technique cannot be rigorously applied to reconstruct high level language programs, for which the mapping to executable code is extremely complex.

Operational feedback

Feedback from the in-service behaviour of the software may provide additional confidence in its ability to perform its desired functions. If such information is to be used then an analysis of the relevance of the past usage, especially its compatibility with any proposed new environment, and thus with the proposed new application, should be provided.
Hazards analysis

The objective of hazards analysis is to:

— verify that the software required to handle failure modes identified by subsystems hazards analysis does so effectively;
— identify any failure modes that can lead to an unsafe state and make recommendations for changes;
— determine the sequence of inputs which could lead to the software causing an unsafe state and make recommendations for changes.

Any computer based monitoring and control system and its interfaces to the plant should be evaluated at various phases of the development for potential contributions to hazards at the plant level. (Possible techniques are outlined in Ref. [2].) When such potential critical behaviours are identified, they should be traced through the computer system design, the software design and the code in order to identify parts of the software design and of the code that require special design features.

The monitoring and control system software design and coding should be verified to determine whether the plant safety criteria have been satisfied in the software design and whether the software implementation has impaired safety or introduced new hazards. Because of the difficulty of these verifications and the impact they may have on the design in terms of proposed changes or additional features, it is recommended that they be carried out as the design proceeds rather than waiting until the implementation is complete.

The results of such hazards analyses should be documented, and the completeness of their coverage should be demonstrated.

Possible techniques for performing hazards analyses are outlined in Refs [2, 3].

Commissioning tests

Computer programs used in monitoring and control systems subject to commissioning tests should be identified. The scope of the tests should be sufficient to demonstrate the required performance over the range of operation of the controlled function or process. The scope of the tests should take into consideration any differences between the validation environment at the developer's facilities and the actual application environment. The risks associated with these differences should be identified and factored into the scope of the tests. The scope of the tests for each computer program should be documented and should be sufficient to provide confidence that the control or monitoring system responds to the computer program signals as required.
Commissioning results should be documented. The documentation should include, or reference information on:

- the computer program tested;
- the computer hardware used;
- test equipment and test tools used;
- the scope of the tests;
- test results.

A documented demonstration of test coverage adequacy, including tracing of tests to source requirements (such as safety system requirements), should be provided by the commissioning team.

**Issues involving testing**

*Test process*

Testing is a process of observing a system's behaviour under a selected set of circumstances and comparing actual behaviour with required behaviour with the aim of finding errors. The test procedures should present the rationale for each test case and provide for tracing of the test cases to the relevant source documents.

The results available from previous tests that have been carried out during the testing and commissioning phases and the relevant test procedures can be used for a direct comparison to check whether the software still correctly performs the originally specified tasks.

*Test planning*

The basis for the testing program is a testing plan which should reference all applicable test procedures. Effective test planning should start with the requirements specification and address the totality of the testing to be performed throughout the development and applicable phases of the software life-cycle, including the operation and maintenance phases where changes can be made to the software.

The test plan should contain information to demonstrate that the suite of tests provides means by which full coverage of the software functions is achieved. The test plan should contain information that makes it possible to trace each of the test cases to one of the requirements or design documents used to derive the test cases.

The test plan should be comprehensive and as complete as is reasonably practicable. The test procedures should present the rationale for each test case and
provide for tracing of the test cases to the relevant source documents. The expected test results should be stated (with their method of derivation) in the test documentation prior to execution of the tests. Typical topics discussed in the test plan include:

**Performance testing.** Performance testing shows how the performance requirements are met. The means by which this is demonstrated should be documented. This documentation should cover all timing requirements (for monitoring and control system software), relating to speed of response to inputs, time to detect and recover from faults, and capability to accept high input rates.

**Structural testing.** Structural testing includes statement, branch and path testing of the computer program. Targets for the structural coverage metrics (e.g. 100% statements and branch coverage) should be stated and justified in the test plan. Any deviation from the targets stated in the plan should be justified and documented. Any deletions should be noted.

**Interface testing.** The test programme should pay particular attention to the testing of module/module, software/hardware or internal/external interfaces. The tests should confirm that all data passing mechanisms and the interface protocols are performing satisfactorily.

**Mode coverage.** All modes of system operation should be considered during the design of the test cases.

The means by which the testing of exceptional conditions (e.g. divide by zero, variable out of range) is to be performed should be identified. This might require the use of special test tools (e.g. in-circuit emulators).

Further guidance on the generation of test cases can be found in Ref. [2].

An important element of the planning process is to define a testing organization that is independent from personnel participating in the various software design and development processes to ensure that testing is carried out objectively and effectively.

The testing processes should be defined, including testing standards and conventions, practices to be used and the identification of test status.

Each testing process includes preparing procedures to specify the environment in which the code is tested, detailing the steps to be followed by the tester, and specifying the set of inputs, execution conditions and expected results.

Subsystem test procedures should describe the expected results of each test case so that a pass/fail determination can be made as part of the outcome of each test. The expected results should be based on the information contained in the software requirements specification.

Subsystem test procedures, validation test procedures, reliability test procedures, as well as the unit test procedures should identify all equipment (and its required calibration), tools and support software required to perform the tests and provide adequate set-up and test execution instructions so that the test can be repeated
by personnel who did not perform the original test. Detailed information on the structure of the code under test should be provided.

The installation and commissioning plan should cover the commissioning test cases and sequence and the corresponding plant states needed to confirm proper functioning of the system in the plant environment.

Validation test procedures for control system software should define test cases, using dynamic simulation of input signals, to cover normal operation, anticipated operational occurrences, abnormal incidents and accident conditions requiring subsystem action as described in the software design documentation.

More detailed recommendations on test planning are contained in Ref. [4].

Interface control

The interfaces should be identified between organizational entities within the design and development organization and between the design organization, its client and other organizations involved in the software system development process. Controls should be established to monitor the interaction between the various organizations and should include the assignment of responsibilities and the controlled issue of documents across organizational interfaces. A clear division of responsibilities between interacting organizations is important to avoid duplication of effort or the omission of important tasks. The controlled transfer of documentation is important to ensure that current information is available to those who need it.

Test performance

Recommendations on the performance of tests are given in Ref. [4].

Test repeatability

The equipment used to conduct the tests should be uniquely identified and recorded in the test procedures to ensure that tests are repeatable. Such information also assists in the investigation of anomalies. During a testing process, verifiers should also record sufficient information about the process used so that specific tests could be repeated with confidence of achieving the same results.

If any testing tools or computer program performance measurement tools are used, they should be completely identified (including version and configuration) and any input or set-up parameters used should be recorded. Such tools should be qualified for their intended use.
**Test records**

Test results should be documented in appropriate reports. These test reports should capture:

- the test methods used;
- outputs from the tests;
- an assessment of the test results against expected results;
- evidence that the tests were executed and analysed as specified in the test procedures.

Test reports should record all deviations from the expected results, with their justifications. The test plans, procedures and expected results, and reports on the tests, should be maintained and be available for quality assurance audits and third party assessments.

Test reports should be prepared in a standard format.

Additional recommendations on the preparation of test records are given in Ref. [4].

**Test reviews**

Test results should be reviewed to assure that the test requirements have been satisfied and that the results confirm that the computer program performance is as specified in the software design documents.

Specific recommendations on the performance of test reviews are given in Ref. [4].

**Non-conformance control**

Recommendations on non-conformance control are given in Ref. [4].

**Corrective action**

Recommendations on the performance of corrective actions are given in Ref. [4].

**Special concerns (such as security and safety)**

For applications where safety, security or financial risk are major concerns, there is a growing tendency for software projects to include independent, formal and
semiautomatic verification and validation methods. Such methods typically include subsystem test procedures with test cases which attempt to subvert any existing safety or security mechanisms.

The tests should be designed to demonstrate that effective measures have been taken to protect the computer based system during its whole lifetime against physical attacks, intrusions, intentional and non-intentional, frauds, viruses, etc.

Some security requirements applying to the whole computer system are addressed through requirements on the software. Examples are validity checks on inputs and stored data, or even on the computer program itself.

**Recommendations on test procedures**

Procedures should be in place to record and analyse test results. The expected outputs should be stated in the test procedures before the tests are undertaken. All outputs from the system under test should be monitored. Any deviation from the expected results should be investigated. The results of such investigations should be documented.

Where it is determined that errors exist in either the hardware or software then any necessary changes should be made under the control of agreed change control procedures. Errors should be analysed to determine their root cause and why early detection did not occur during the development process. Appropriate changes to the development processes and infrastructure systems (such as configuration management systems, records management systems) should be made to prevent the recurrence of similar errors during future software development projects.

**Training requirements**

The objective of training is to ensure that the testing personnel have the necessary skills to perform their tasks, and are completely conversant with all required procedures and standards which affect their work.

Personnel planning and conducting the tests should be given appropriate training in the use of test tools, procedures and techniques.

**Anomalies**

Both data and control flow analysis can only provide information about the properties of the design or code. Analysis should also uncover anomalies which, when investigated, reveal more subtle errors. For example, if a variable is identified but never used, it may indicate that the programmer anticipated a need for the variable early in the development, but later forgot to deal with the anticipated need.
Any test performance anomalies should be reviewed and any necessary changes to the test procedures should be implemented according to the appropriate change control procedure.

Records of all anomalies should be maintained, including evidence of the means by which they were resolved.

Possible techniques for inspecting software for anomalies are outlined in Ref. [3].
Appendix XIX

RECOMMENDATIONS ON VERIFICATION REPORTS AND ACTIVITIES FOR MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

Requirements review report

Requirements review reports should provide evidence that the review has covered:

— all requirements in the design input documents (DID) and in the software requirements specification (SRS);
— all requirements and design constraints appearing in the SRS which are not derived from the DID;
— all applicable standard practices and procedures.

Design review report

Design review reports should provide evidence that the review has covered:

— all requirements in the SRS and all programs, data structures and databases in the software design description (SDD);
— applicable standard practices and procedures.

Design verification report

Design verification reports should provide evidence that the review has covered:

— all applicable requirements in the SRS and all programs, data structures and databases in the SDD;
— all justifications for inclusion in the SDD of any functionality outside the requirements defined in the SRS.

Code review report

Code review reports should provide evidence that:

— the review has covered all programs, data structures and databases in the code;
— the code has been reviewed for compliance with applicable standard work practices and procedures.
Code verification report

Code verification reports should provide objective evidence that the verification has covered all programs, data structures and databases in the SDD and all of the code.

Hazards analysis report

The hazards analysis report should comply with applicable standard work practices and procedures and identify:

— the failure modes related to random access memory variables whose corruption could lead to an unsafe subsystem failure;
— the failure modes related to read-only-memory constants whose corruption could lead to an unsafe subsystem failure;
— the failure modes related to instructions in the code which could lead to an unsafe subsystem failure;
— the input conditions which could lead to the software causing an unsafe state;
— all self-checking software contained in the code and its ability to eliminate the identified failure modes or reduce their likelihood of occurring (and recommend code modifications which would eliminate the identified failure modes or reduce their likelihood of occurring);
— the revisions of the documents analysed;
— the analysis procedures applied and the methods and tools used;
— the analysis participants.

Unit test procedures

Unit test procedures should:

(a) Define the tests performed on the target processor but possibly in a simulated environment.
(b) Define the test cases, derived from the analysis of the SDD, to ensure that the executable code for each program behaves as specified in the SDD. The number of test cases should be considered sufficient when they include:
— all possible decision outcomes;
— all possible conditions for each decision;
— tests on each boundary and values on each side of each boundary for each input;
— tests based on postulated coding implementation errors.
(c) Define a sufficient number of test cases, derived from the analysis of the code, to ensure that the executable code for each program behaves as specified in the
SDD. The number of test cases should be considered sufficient when they cause to be executed at least once:

- every program statement;
- all possible decision outcomes;
- all possible conditions for each decision;
- each loop with minimum, maximum and at least one intermediate number of repetitions;
- a read and write to every memory location used for variable data;
- a read of every memory location used for constant data.

(d) Define a sufficient number of tests to cause each interface to be exercised.
(e) Describe the expected results of each test case so that a pass/fail determination can be made as to the outcome of the test; the expected results should be based on the information contained in the SDD.
(f) Identify all equipment (and its required calibration), tools and support software required to perform the test and provide adequate set-up and test execution instructions so that the test can be repeated by personnel who did not perform the original test.
(g) Identify the programs being tested.
(h) Identify the SDD name, document identity and revision number.
(i) Provide a cross-reference between sections of the SDD and the test procedures to document the test coverage.
(j) Comply with applicable standard practices and work instructions.

Subsystem test procedures

Subsystem test procedures should:

(a) Define the tests to be carried out with target hardware and any pre-developed software;
(b) Define test cases to test each functional requirement in the SRS;
(c) Define test cases to test the performance requirements as described in the SRS;
(d) Identify all resources used by the subsystem software and define test cases which test the subsystem under conditions that attempt to overload these resources in order to determine if the functional and performance requirements defined in the SRS are met;
(e) Define tests to exercise any interfaces between the software and the target hardware, and the software and any pre-developed software;
(f) Define tests to show that the subsystem meets its requirements under each hardware configuration and operational option;
(g) Define tests to test the ability of the subsystem to respond as indicated in the SRS to software, hardware and data errors;
(h) Define test cases which attempt to subvert any existing safety or security mechanisms;

(i) Define a reasonable number of test cases where the input values are randomly selected from the set of possible values such that the set of values for all tests for each input has a uniform probability distribution across the domain of the input;

(j) Describe expected results of each test case so that a pass/fail determination can be made as to the outcome of the test. The expected results should be based on the information contained in the SRS;

(k) Identify all equipment (and its required calibration), tools and support software required to perform the test and provide adequate setup and test execution instructions so that the test can be repeated by personnel who did not perform the original test. Detailed procedures on how the code under test is built should be provided;

(l) Identify the SRS name, document identity and revision number;

(m) Provide a cross-reference between the sections of the SRS and the test procedures to document the test coverage;

(n) Comply with applicable standard practices and work instructions.

Validation test procedures

Validation test procedures should:

— Require all tests to be done with target hardware and pre-developed software;
— Define test cases to test each functional requirement in the DID that is applicable to the subsystem;
— Define test cases to test each performance requirement in the DID that is applicable to the subsystem;
— Define test cases, using dynamic simulation of input signals, to cover normal operation, anticipated operational occurrences, abnormal incidents and accident conditions requiring subsystem action as indicated in the DID;
— Describe expected results of each test case so that a pass/fail determination can be made as to the outcome of the test from information provided in the DID;
— Identify all equipment (and its required calibration), tools and support software required to perform the test and provide adequate set-up and test execution instructions so that the test can be repeated by personnel who did not perform the original test;
— Identify the DID components and revisions;
— Provide a cross-reference between the sections of the DID components and the test procedures to document the test coverage;
— Comply with applicable standard practices and work instructions.
Reliability test procedures

Reliability test procedures should:

(a) Require all tests to be done with the target hardware and pre-developed software;
(b) Define a sufficient number of tests representative of the software's usage profile, to provide statistically valid evidence showing that the probability of failure of the software is small enough that the computer system will meet its reliability requirements as described in the SRS;
(c) Define each test in terms of:
   — the initial value of each input;
   — the final value of each input;
   — the length of time of the test or period;
   — the time related function describing how each input will vary over the period of the test;
(d) Randomly select the final input values from a distribution that is representative of the values seen when the system need not take some action;
(e) Randomly select the final input values from a distribution that is representative of the values seen by the system when it is required to take some action;
(f) For each input, randomly select the time related function from a representative distribution that represents intermediate values assumed by the input as it progresses from the initial value to the final value; the function will include the effects of instrument response times, signal noise and any other characteristics that are known about the input;
(g) Randomly select the time period of the test and ensure that all retained memory is initialized prior to running each test, so that it is sufficient to assume that the effects of the retained memory of the software do not invalidate the independence requirement between tests to ensure statistical validity. Justification should be provided for any exceptions made to this rule (e.g. excessively long or infinite time periods not covered);
(h) Ensure that the set of values for each input for all tests has a representative distribution across the domain of values which the input may assume;
(i) Define the expected behaviour of the outputs of the system during the tests. The expected behaviour must be based on the specified behaviour in the SRS;
(j) Provide justification supporting the chosen distributions showing that they are representative of the software's usage profile;
(k) Define the acceptance criteria to be applied when comparing actual behaviour with expected behaviour;
(l) Identify all equipment (and its required calibration), tools and support software required to produce the input and the expected results data for the test cases;
(m) Identify all equipment (and its required calibration), tools and support software required to perform the test and to provide adequate setup and test execution instructions so that the test can be repeated by personnel who did not perform the original test;

(n) Identify the SRS and DID name, document identities and revision numbers;

(o) Comply with applicable standard practices and work instructions.

Test reports

Test reports should document the results of each test activity, such as unit testing, subsystem testing, reliability testing and validation testing. For each test activity the test reports should:

(a) Identify the test performed;
(b) Identify the test procedures used;
(c) Include the comparison of actual and expected test results as defined in the applicable test procedures;
(d) Summarize any discrepancies;
(e) Summarize all positive findings;
(f) Describe the conclusions and recommendations;
(g) Identify the date and time of the performance of each test;
(h) Identify the program or subsystem and the versions being tested;
(i) Identify the testers involved;
(j) Reference the detailed test results;
(k) Comply with applicable standard work practices and procedures.
Appendix XX

RECOMMENDATIONS ON COMMISSIONING MONITORING, CONTROL AND SAFETY SYSTEM SOFTWARE

Testing control

Test requirements and acceptance criteria should be defined. They should be based on applicable design and safety analysis documents.

Test plans should be documented. The plans should identify, or reference procedures which identify, the following:

— test objectives;
— the scope of the tests;
— types of tests to be performed;
— sequence of tests and expected results;
— special concerns such as security and safety;
— test environment, tools and test tools, including any associated qualification and controls;
— personnel required and associated training requirements, including training material;
— the responsibilities for specification and performance of the tests;
— acceptance criteria;
— the method of recording results;
— disposition of non-conformances, including suspension criteria and resumption requirements;
— the repeatability of the tests.

The processes should be defined for inspection and test activities, including testing standards and conventions, practices to be used and the identification of test status.

Test results should be reviewed to assure that the test requirements have been satisfied and that the results confirm the monitoring, control and safety system performance as specified in design and safety analysis documents.

The computer programs subject to commissioning tests should be identified. The scope of the tests for each computer program should be documented and should be sufficient to provide confidence that the control system responds to the program signals as required. The scope of the tests should be sufficient to demonstrate the required performance over the range of operation of the controlled function or process. The scope of the tests should take into consideration any differences between the validation environment at the developer's facilities and the actual application.
environment. The risks associated with these differences should be identified and factored into the scope of the tests.

Inspections should be performed and documented to confirm that:

(a) The computer programs and associated documentation represent the versions and document revisions specified;
(b) The programs reside on the specified electronic media;
(c) Computer program versions and document editions are related;
(d) The integrity of the computer programs is intact;
(e) The supporting documentation is complete.

Testing tools, system performance data and computer program performance measurement tools should be identified and qualified for their intended use. The set of testing tools should be placed under configuration management.

Commissioning results should be documented. The documentation should include, or reference information on:

— the computer program tested;
— the computer hardware used;
— test equipment and test tools used;
— the scope of the tests;
— test results.

Configuration control

Procedures should be established to ensure that each monitoring, control and safety system computer program version is uniquely identified together with the related documentation editions.

A system should be established for tracking and reporting on the status of the monitoring, control and safety system computer program components.

Proposed modifications to computer programs and related documentation should be reviewed before implementation by a group of reviewers who have access to pertinent background information and who have an adequate understanding of the requirements and the intent of the design.

A configuration management system should be established with the following features and capabilities:

(a) A configuration plan identifying when configuration control should be applied and the components to be controlled;
(b) A system for uniquely defining and controlling the components of the control system computer programs and the associated documentation and any changes
to these components, for maintaining the integrity and traceability of the configuration;

c) Control of requests for changes subject to appropriate change control processes;

d) A computer program component library with controls for adding to and withdrawal from that library;

e) Maintenance of a master copy of the current version of all computer program components;

(f) Maintenance of sufficient information about previous versions to permit recovery of any issued computer program version.

Measures should be established to ensure that:

(a) Control computer programs and related documents, including changes and revisions, are:
   — reviewed for adequacy and approved for release by authorized personnel;
   — distributed for use at the location where the prescribed activity is performed or where the document is required for reference.

(b) The latest authorized computer program versions and related documentation are used.

(c) Information that permits the identification of any changes and revisions to all such computer program versions and related documentation is maintained.

Measures should be established to ensure that obsolete computer program versions and related documentation are identified as such and are promptly removed from use.

**Non-conformance control**

Non-conformances identified during commissioning and their eventual resolution should be documented consistent with the requirements of the configuration management system.

The persons with responsibility and authority for the disposition of all non-conformances should be identified.

The methods for reporting, tracking the progress of and making the final disposition of all non-conformances should be defined.
GLOSSARY

architecture. The organizational structure of a system or component [29].

assemble. To translate a computer program expressed in an assembly language into a machine language [30].

availability. The fraction of time that a system is actually capable of performing its mission.

code. In software engineering, computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler or other translator [29].

coding. Translating a computer program design into a programming language [30].

compile. To translate a computer program expressed in a high level language into its machine language equivalent [29].

complexity. The degree to which a system or component has a design or implementation that is difficult to understand and verify [29].

computer program. A sequence of instructions suitable for processing by a computer [30].

configuration management. The process of identifying software components, controlling changes, and maintaining the integrity and traceability of the configuration [31].

database. A collection of interrelated data stored together in one or more computerized files [29].

data structure. A representation of the logical relationships among individual data elements [30].

debugging. The process of locating, analysing and removing errors [30].

defect. See definition for 'error'.

error. The difference between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition [29].

executable code or file. A computer program in a language that can be directly executed by a computer [30].
failure. The inability of a system or component to perform its required functions within specified performance requirements [29].

hardware. Physical equipment used in the execution of computer programs and the storage or transfer of data [30].

heuristics. Rules for developing good systems.

integration. The process of combining software components, hardware components, or both into an overall system [29].

interpret. To translate and to execute each source language statement of a computer program before translating and executing the next statement [30].

maintainability. The ease with which a computer program or component thereof can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [29].

modularity. The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components [29].

module. A discrete and identifiable part of a computer program that can be separately compiled and assembled [30].

redundancy. Provision of alternative (identical or diverse) elements or systems, so that any one can perform the required function regardless of the state or failure of any other.

reliability. The probability that a device, system or facility will perform its intended function satisfactorily for a specified time under stated operating conditions.

safety related system. System important to safety which is not included in safety systems [8].

safety system. System important to safety, provided to ensure the safe shutdown of the reactor or the residual heat removal from the core, or to limit the consequences of anticipated operational occurrences and design basis accident conditions.

simplicity. The degree to which a system or component has a design and implementation that are straightforward and easy to understand [29].

software. Computer programs, procedures, rules and associated data and documentation pertaining to the operation of a computer program [32, p.13].
**software life-cycle.** The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life-cycle typically includes a concepts phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase and, sometimes, retirement phase [29].

**software quality assurance.** A planned and systematic pattern of actions designed to provide adequate confidence that a computer program will be of the required quality.

**source code.** Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler or other translator [29].

**source program.** A computer program that must be compiled, assembled or interpreted before being executed by a computer [29].

**testability.** (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met [29].

**testing.** The process of exercising a computer program or portion thereof to verify that it satisfies specified requirements [30].

**traceability.** Characteristic showing there is a high probability of finding the source of information in the products of the preceding life-cycle phase.

**unit.** A separately testable element specified in the design of a computer program component [29].

**usability.** The ease with which the user can learn to operate, prepare inputs for, and interpret outputs from a computer program [29].

**validation.** Testing conducted on a completed, integrated computer system (hardware and software) to ensure compliance with the requirements specification.

**variable.** A quantity that can assume any of a given set of values [30].

**verification.** The process of ensuring that a phase in the software development process meets the requirements imposed on it by the previous phase.
walkthrough. A review process in which a designer or programmer leads one or more members of a review team through a computer program or its design, or portions thereof [30].
REFERENCES


158
BIBLIOGRAPHY

TEXTBOOKS


JOURNAL ARTICLES

COCHEU, T., Building a leadership foundation through improvement, Quality Progress, 41–44 (April, 1995).


STANDARDS


CONTRIBUTORS TO DRAFTING AND REVIEW

Abel, R. Atomic Energy of Canada Ltd, Canada
Bienvenu, H. ANDRA, France
Brijesh, A. Department of Energy, Battelle Memorial Institute, USA
Calderon, T. INVAP S.E., Argentina
De John, M. Atomic Energy of Canada Ltd, Canada
Duong, M. International Atomic Energy Agency
Elde, M. Atomic Energy of Canada Ltd, Canada
Ficheux-Vapné, F. Electricité de France, France
Kulig, M. International Atomic Energy Agency
Moldaschl, H. Siemens AG, Germany
Pachner, J. International Atomic Energy Agency
Pieroni, N. International Atomic Energy Agency
Reznik, V. International Atomic Energy Agency
Stuller, J. State Office of Nuclear Safety, Czech Republic
Turi, T. Paks Nuclear Power Plant, Hungary
Van der Geld, F. FORATOM
Welbourne, D. United Kingdom
Yates, B. Health and Safety Executive, United Kingdom

Advisory Group Meeting
Vienna, Austria: 2–4 February 1998

Consultancy Meeting
Vienna, Austria: 24–26 March 1997
WHERE TO ORDER IAEA PUBLICATIONS

In the following countries IAEA publications may be purchased from the sources listed below, or from major local booksellers. Payment may be made in local currency or with UNESCO coupons.

AUSTRALIA
Hunter Publications, 58A Gipps Street, Collingwood, Victoria 3066
Telephone: +61 3 9417 5361 • E-mail: jpdavies@ozemail.com.au

BELGIUM
Jean de Lannoy, avenue du Roi 202, B-1060 Brussels • Telephone: +32 2 538 43 08 • Fax: +32 2 538 08 41
E-mail: jean.de.lannoy@infoboard.be • Web site: http://www.jean-de-lannoy.be

BRUNEI
Parry’s Book Center Sdn. Bhd., 60 Jalan Negara, Taman Melawati, 53100 Kuala Lumpur, Malaysia
Telephone: +60 3 4079 176, 4079179, 4087235, 4087528 • Fax: +60 3 407 9180
E-mail: haja@pstd@janng.my • Web site: http://www.mol.net.my/~parrybook/parrys.htm

CANADA
Renouf Publishing Company Ltd., 1-5369 Canotek Rd., Ottawa, Ontario, K1J 9J3
Telephone: +613 745 2665 • Fax: +613 745 7660
E-mail: order.dept@renoufbooks.com • Web site: http://www.renoufbooks.com

CHINA
IAEA Publications in Chinese: China Nuclear Energy Industry Corporation, Translation Section, P.O. Box 2103, Beijing

DENMARK
Munksgaard Direct, Postbox 173, DK-1000 København K
Telephone: +45 77 33 33 33 • Fax: +45 77 33 33 77
E-mail: direct@munksgaarddirect.dk • Web site: http://www.munksgaarddirect.dk

FRANCE
Nucleon, Immeuble Platon, Parc les Alphettes, F-91194 Gil-sur-Yvette, Cedex
Telephone: +33 1 69 35 36 36 • Fax: +33 1 69 35 09 99 • E-mail: nucleon@wanadoo.fr

GERMANY
UNO-Verlag, Verlags- und Verlags GmbH, Am Hofgarten 10, D-53113 Bonn
Telephone: +49 228 94 90 20 • Fax: +49 228 94 90 222
E-mail: unoverlag@aol.com • Web site: http://www.uno-verlag.de

HUNGARY
Librófia Ltd., Book Import, P.O. Box 126, H-1656 Budapest
Telephone: +36 1 257 7777 • Fax: +36 1 257 7472 • E-mail: books@librotrade.hu

INDIA
Allied Publishers Limited, 1-13/14, Asaf Ali Road, New Delhi 110002
Telephone: +91 11 3233002, 004 • Fax: +91 11 3235967
E-mail: sblue@del2.vsnl.net.in • Web site: http://www.alliedpublishers.com

ISRAEL
YOZMOT Ltd., 3 Yohanan Hasandlar St., P.O. Box 56055, IL-61560 Tel Aviv
Telephone: +972 3 5284851 • Fax: +972 3 528 5397

ITALY
Libriera Scientifica Dott. Lucio di Biasio “AEIOU”, Via Coronelli 6, I-20146 Milan
Telephone: +39 2 48 95 45 52 or 48 95 45 62 • Fax: +39 2 48 95 45 48

JAPAN
Manzenn Company, Ltd., P.O. Box 5000, 100-3191 Tokyo International
Telephone: +81 3 744 8673 • Fax: +81 3 744 8676

MALAYSIA
Parry’s Book Center Sdn. Bhd., 60 Jalan Negara, Taman Melawati, 53100 Kuala Lumpur, Malaysia
Telephone: +60 3 4079 176, 4079179, 4087235, 4087528 • Fax: +60 3 407 9180
E-mail: haja@pstd@janng.my • Web site: http://www.mol.net.my/~parrybook/parrys.htm

NETHERLANDS
Mokusui Nijhoff International, P.O. Box 269, NL-2501 AX The Hague
Telephone: +31 79 124 76 00 • Fax: +31 79 132 36 97
E-mail: sblue@del2.vsnl.net.in • Web site: http://www.mokusui-nijhoff.nl

POLAND
Ars Polona, Book Department/Import, P.O. Box 1001, PL-00-950 Warsaw
Telephone: +48 22 826 1201 ext. 147, 151, 159, 167 • Fax: +48 22 826 4763
E-mail: ksiazki@arspolona.com.pl • books119@arspolona.com.pl • Web site: http://www.arspolona.com.pl

SINGAPORE
Parry’s Book Center Pte. Ltd., 528 A MacPherson Rd., Singapore 1336
Telephone: +65 744 8673 • Fax: +65 744 8676
E-mail: haja@pstd@janng.my • Web site: http://www.mol.net.my/~parrybook/parrys.htm

SLOVAKIA
Alfa Press, s.r.o, Račianska 20, SK-832 10 Bratislava • Telephone/Fax: +421 7 566 0489

SPAIN
Diaz de Santos, S.A., C. Juan Bravo, 3A, E-28006 Madrid
Telephone: +34 91 791 94 80 • Fax: +34 91 575 55 03
E-mail: compras@diazasentas.es • carmela@diazasentas.es • julio@diazasentas.es • Web site: http://www.diazasentas.es

UNITED KINGDOM
The Stationery Office Ltd, International Sales Agency, 51 Nine Elms Lane, London SW8 5DR
Telephone: +44 171 873 9090 • Fax: +44 171 873 8463
Web site: http://www.the-stationery-office.co.uk

UNITED STATES OF AMERICA
Bernan Associates, 4611-F Assembly Drive, Lanham, MD 20706-4391, USA
Telephone: 1-800-274-4447 (toll-free) • Fax: (301) 459-0056 / 1-800-865-3450 (toll-free)
E-mail: orders@bernan.com • Web site: http://www.bernan.com

Renouf Publishing Company Ltd., 812 Proctor Ave., Ogdensburg, New York, 13669
Telephone: +888 551 7470 (toll-free) • Fax: +888 508 8546 (toll-free)
E-mail: order.dept@renoufbooks.com • Web site: http://www.renoufbooks.com
SAFETY OF NUCLEAR POWER PLANTS: DESIGN: Safety Requirements
Safety Standards Series No. NS-R-1

This publication establishes design requirements for safety functions and associated structures, systems and components important to the safe operation of a nuclear power plant and establishes requirements for a comprehensive safety assessment to identify the potential hazards that may arise in the operation of a plant.
STI/PUB/1099 (67 pp.; 2000)
ISBN 92-0-101900-9
Price: 200 Austrian schillings (€14.53)

QUALITY ASSURANCE FOR SAFETY IN NUCLEAR POWER PLANTS AND OTHER NUCLEAR INSTALLATIONS: Code and Safety Guides Q1-Q14
Safety Series No. 50-C/SG-Q

The revised Safety Code and its corresponding Safety Guides present new simplified basic requirements and implementation methods for quality assurance in a nuclear context.
Available in English, French and Russian.
STI/PUB/1016 (364 pp., 6 figures; 1996)
ISBN 92-0-103696-5
Price: 1040 Austrian schillings (€75.58)

VERIFICATION AND VALIDATION OF SOFTWARE RELATED TO NUCLEAR POWER PLANT INSTRUMENTATION AND CONTROL
Technical Reports Series No. 384

This report provides practical guidance on the methods available for verification of the software and validation of computer based systems in nuclear power plants.
STI/DOC/010/384 (126 pp., 7 figures; 1999)
ISBN 92-0-100799-X
Price: 480 Austrian schillings (€34.88)

SOFTWARE IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS
Technical Reports Series No. 367

This report provides guidance in dealing with the important issues of software engineering facing nuclear power plant system designers, software producers and regulators.
STI/DOC/010/367 (169 pp., 1 figure; 1994)
ISBN 92-0-101594-1
Price: 560 Austrian schillings (€40.70)

http://www.iaea.org/worldatom/Books