Software Important to Safety in Nuclear Power Plants
SOFTWARE IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS
The following States are Members of the International Atomic Energy Agency:

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

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, 1994

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

Printed by the IAEA in Austria
June 1994
STI/DOC/010/367
SOFTWARE IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS
VIC Library Cataloguing in Publication Data

   p. ; 24 cm. — (Technical reports series, ISSN 0074-1914 ; 367)
   STI/DOC/010/367
   ISBN 92-0-101594-1
   Includes bibliographical references.

1. Nuclear engineering—Safety measures—Computer programs.
   II. Series: Technical reports series (International Atomic Energy Agency) ; 367.

VICL 94-00084
FOREWORD

The use of software based equipment and systems in both new and older nuclear power plants is rapidly increasing as analogue technology is being phased out by the control and instrumentation industry in favour of digital, software based devices. These applications include not only safety related applications such as process control and monitoring, but also safety critical applications for safety functions such as reactor protection or other safety feature actuation. Software is therefore of increasing importance to safety in nuclear power plants and the dependability of this software must be ensured.

This requires a concerted effort to deal with the inherent possibility of failure of software by means of the systematic production, use and licensing of software important to safety in nuclear power plants. Although a number of national and international standards dealing with the production of and quality assurance for software important to safety exist or are being prepared, internationally agreed criteria for the safety assessment of such software are not generally available.

In 1991 the IAEA initiated activities on software important to safety aimed at helping Member States to ensure that software based systems in nuclear power plants are safe and properly licensed. The Advisory Group, which included distinguished experts such as Professors N.G. Leveson of the USA and D.L. Parnas of Canada, provided recommendations for these activities, including detailed guidance for the preparation of this report.

The report provides guidance on current practices, documenting their strengths and weaknesses for dealing with the important issues of software engineering that nuclear power plant system designers, software producers and regulators are facing. The focus of the report is on safety critical applications of general purpose processors controlled by custom developed software; however, it should also have application in safety related applications and for other types of computers. In addition to system designers, software producers and regulators, the intended readership of this report includes users of software based systems, who should be aware of the relevant issues in specifying and obtaining software for systems important to safety.

The report is a result of a two year effort by a large international team which the IAEA would like to acknowledge. It was prepared for publication by P.-J. Courtois of Belgium and J. Pachner of the IAEA using contributions of the following technical specialists who drafted individual sections of the report: R.P. Taylor (Sections 3, 8, 9 and 14), R. Hohendorf (Sections 2, 4 and 15), G.H. Archinoff (Section 2), N. Ichiyen (Section 4), P. Joannou (Section 5), A. Wassyg (Section 5), R. Crane (Section 11) and M. Viola (Section 11) of Canada;
B. Carré (Sections 6, 7, 12 and 13), J. Baldwin (Sections 4 and 11), D.N. Wall (Section 10) and R. Bloomfield (Section 1 and Annex I) of the United Kingdom; W. Ehrenberger (Section 10) of Germany; P.-J. Courtois (Sections 6 and 10) of Belgium; and J. Pachner (Section 1 and Annexes I and II) of the IAEA. Comments on the first draft received from representatives of 13 Member States at a Technical Committee Meeting in June 1992 were taken into account in the preparation of the final report.

EDITORIAL NOTE

Although great care has been taken to maintain the accuracy of information contained in this publication, neither the IAEA nor its Member States assume any responsibility for consequences which may arise from its use.

The use of particular designations of countries or territories does not imply any judgement by the publisher, the IAEA, as to the legal status of such countries or territories, of their authorities and institutions or of the delimitation of their boundaries.

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.

Reference to standards of other organizations is not to be construed as an endorsement on the part of the IAEA.
## CONTENTS

1. INTRODUCTION ................................................................. 1
   1.1. Purpose and objectives ................................................. 1
   1.2. Scope .............................................................................. 2
   1.3. Background ...................................................................... 2
   1.4. Structure .......................................................................... 3
   References ............................................................................... 4

2. APPLICABILITY OF EXISTING SAFETY PRINCIPLES TO SOFTWARE IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS ....................................................... 6
   2.1. Issues ............................................................................... 6
   2.2. Experience ........................................................................ 6
   2.3. Current practices ............................................................ 7
   References ............................................................................... 15

3. THE ENGINEERING CONTEXT ............................................. 16
   3.1. Issues ............................................................................... 16
   3.2. Experience ........................................................................ 16
   3.3. Current practices ............................................................ 18
   3.4. Summary and follow-up .................................................... 27
   References ............................................................................... 28

4. MANAGEMENT ........................................................................... 30
   4.1. Issues ............................................................................... 30
   4.2. Experience ........................................................................ 33
   4.3. Current practices ............................................................ 37
   4.4. Summary and follow-up .................................................... 40
   References ............................................................................... 42
1. INTRODUCTION

1.1. PURPOSE AND OBJECTIVES

Software is of increasing importance to safety in nuclear power plants (NPPs), and the dependability\(^1\) of software must be ensured.

The purpose of this report is to assist Member States in ensuring that computer systems "important to safety\(^2\) in NPPs are safe and properly licensed. The licensing approach should be based on principles and practices similar to those adopted for other plant systems and components important to safety, and should provide similar confidence.

A number of national and international standards dealing with the production of software for safety critical or safety related\(^3\) applications exist or are being prepared (see Annex I and Ref. [1.3]). However, there are a number of unresolved issues relating to the formulation of requirements and the software engineering methods for meeting those requirements which make it very difficult to establish fitness for purpose of software. These are areas of active technical development of which Member States should be aware when undertaking the development or acquisition of software important to safety. Perceptions of these problems are becoming clearer, and ways of solving at least some of them are gradually emerging. In this context, the objectives of this report are:

1. to document the situation with regard to these issues in specification, design, development, validation and licensing for safety critical software;
2. to provide a technical basis for a future IAEA Safety Guide on software important to safety.

---

\(^1\) 'Software dependability' means the capability of the software to deliver specified service; reliability, availability, maintainability, safety and security are attributes of dependability [1.1].

\(^2\) 'Items important to safety' are defined in Ref. [1.2] as: "The items which comprise: (1) those structures, systems, and components whose malfunction or failure could lead to undue radiation exposure of the Site Personnel or members of the public (this includes successive barriers set up against the release of radioactivity from nuclear facilities); (2) those structures, systems and components which prevent Anticipated Operational Occurrences from leading to Accident Conditions; (3) those features which are provided to mitigate the consequences of malfunction or failure of structures, systems or components."

\(^3\) 'Safety critical applications' in NPPs are applications for implementing safety functions of emergency reactor shutdown (also referred to as reactor trip, reactor protection or scram), emergency core cooling and containment. 'Safety related applications' in NPPs are those applications important to safety which are not safety critical.
This report is intended to be a source of guidance on current practices, documenting their strengths and weaknesses in ways of dealing with some of the software engineering problems that confront NPP system designers, software producers and regulators. It will also be of interest to software users, to alert them to the relevant issues in specifying or obtaining software for systems important to safety.

The reader should ideally have some knowledge of software standards and software engineering as applied to real-time control systems. However, readers with little or no previous experience in this area will also find the report useful.

1.2. SCOPE

Computers are used in a wide variety of applications in NPPs. These applications include the following functions:

(1) safety functions of emergency shutdown, core cooling and containment;
(2) continuous control functions such as reactivity control or primary heat transport system control;
(3) data acquisition, monitoring and display and information processing functions, and design of nuclear plant systems.

'Computers' can refer to:

(1) general purpose processors controlled by purchased or custom developed software;
(2) special purpose processors controlled by standardized logic blocks, configured or linked for a particular application;
(3) programmable logic controllers.

Within this array of possibilities, this report focuses on safety critical applications of general purpose processors controlled by custom developed software, since this is the area in which there is recent experience in some Member States that should be documented.

There are general principles and techniques in the use of safety critical software that are expected to have a broader application, both in other applications important to safety and in other types of computers. The present report does not discuss all aspects of NPP software in detail, but concentrates on the key areas listed in Section 1.4. If a topic is not discussed, but is nevertheless considered to be important, this is indicated, and references to appropriate material are provided. Software security is recognized as an important issue that is outside the scope of this report.

1.3. BACKGROUND

The nuclear industry deals with systems that must be safe and reliable in order to protect the public from the risks of potential accidents. Although the industry has
accumulated many years of experience with safety related software applications (such as in control systems), it has, in general, been cautious in the application of software in systems important to safety. However, over the past decade software controlled systems important to safety have been installed in a variety of NPPs. There is expectation that this trend will continue, both in new plants and in the refurbishment of older stations when controllers, monitors, display systems and data acquisition and interpretation systems are replaced with software controlled devices. A review of international experience is provided in Refs [1.4 and 1.5] and recent experience is detailed in Annex II.

It is important to ensure that software based systems have the required reliability and that they contribute to, rather than compromise, plant safety. It is also important to recognize that software based systems in NPPs, because of the defence in depth approach used, need to have only adequate reliability and do not always require reliability targets as high as for some applications in the transport sector.

There are a number of potential advantages in using software based systems rather than hardware systems. These include: the ability to perform more complex functions and to keep the operator better informed; higher numerical precision and stability; higher availability due to self-testing; less risk of maintenance induced hardware failures; and overall equipment cost savings. These advantages are elaborated in Section 3, which discusses the engineering context. The advantages of software based systems are balanced by a number of disadvantages relating to the degree of trust that can be placed in computer systems and the possibility of high project and licensing costs. These issues are discussed in detail in this report.

1.4. STRUCTURE

The report is organized into sections discussing the following key areas of software engineering:

Section 2: Applicability of existing safety principles to software important to safety in nuclear power plants: Extension of general safety principles important for engineering software for NPP systems.

Section 3: The engineering context: How nuclear safety principles and general system engineering principles and techniques apply to software.

Section 4: Management: Issues of importance in the management of software important to safety.

Section 5: Modelling for requirement analysis and design: Extension and management of the application of standard engineering modelling and analysis to software engineering.

Section 6: Documentation: The contents and forms of documentation that should be required for NPP software.
Section 7: Programming: Principles of programming and programming languages that should be applied in NPP systems.

Section 8: Verification and validation: How software can be shown to be in conformance with requirements.

Section 9: Testing: Special issues related to testing of computer software.

Section 10: Figures of merit: Measurement and assessment techniques for software.

Section 11: Maintenance: Appropriate approaches for improving and adapting software throughout its lifetime.

Section 12: Tool support: The qualification and appropriate use of tools for software development.

Section 13: Use of existing software: Potential problem areas in using software that has been constructed independently from the NPP system being developed.

Section 14: Personnel qualification and training: Requirements in terms of education and experience for personnel developing NPP software.

Section 15: Balance: Making trade-offs when undertaking a software project so as to satisfy conflicting requirements for system implementation and safety.

It is intended to strike a balance in the report between providing a readable and self-standing narrative and not reproducing information which is readily available elsewhere. Accordingly, the sections of the report do not provide a complete description of the aforementioned key areas, but refer as appropriate to other documents for extraneous detail or background information. The topics which are discussed in greater detail are those in which there is currently the most debate; the selection does not indicate relative importance.

REFERENCES


2. APPLICABILITY OF EXISTING SAFETY PRINCIPLES TO SOFTWARE IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS

In Section 2, principles that have been developed in the areas of both nuclear safety and software engineering are summarized, and their applicability in the development and maintenance of software used in systems important to safety in NPPs is considered.

The nuclear safety principles discussed were presented primarily in IAEA reports [2.1-2.3] and in international standards and the national standards of some Member States [2.4]. Each principle is discussed in terms of a system that uses software to perform a safety function, with particular attention to the applicability of the principle and how it would need to be modified in practice to achieve the same level of safety as originally intended in its application to a hardware system.

Researchers and designers, producers, regulators and users of safety critical software in non-nuclear fields have developed principles that can be applied to help achieve safe software in NPPs [2.5-2.7]. Such principles are also considered here.

2.1. ISSUES

General principles that can be applied to specific practices to achieve a high level of safety in the design and operation of NPPs have been developed over the years. These principles were developed primarily on the basis of experience of Member States with hardware based systems, in particular instrumentation and control systems performing important plant safety functions. With the trend towards the use of computer based instrumentation and control systems in safety applications, these general principles, appropriately adapted, may also be applicable to the use of software. However, to achieve real benefits, it is necessary, in view of the differences between hardware and software, carefully to consider the validity of principles developed on the basis of experience with hardware before they are applied to software.

2.2. EXPERIENCE

The two most basic principles in safety system design are to keep the system simple and to use proven components to assist in achieving an adequate level of confidence in hardware systems. Once the basic design is established, high reliability is achieved through the use of redundant and diverse components and periodic in-service testing. Redundancy confers tolerance to random failures; diversity mitigates the effect of common cause failures; and periodic in-service testing facilitates the timely detection of component failures (see also Section 3.3.3).
Experience in other industries has shown that the traditional methods of achieving confidence in a design may not be sufficient for application to software. The basic principle of simplicity still applies to software, but often the primary reason for using software is to implement functionality that is too complex to be implemented via hardware. It must also be recognized that the use of computers brings with it a minimum level of complexity that exceeds the minimum level associated with hardware. A basic understanding of the nature of computers must be acquired before software based designs of the requisite dependability can be produced.

Another method used to achieve confidence in hardware systems, namely, the use of designs and components that have been proven previously, is generally not practical for software, which is usually customized for individual applications.

Since software errors are design errors, the principle of redundant components does not apply, as all copies of the same software would contain the same design errors. N-version programming is one method of tackling this problem, but there are concerns over its effectiveness, as discussed in Section 3.3.3. Finally, in-service testing is not useful for detecting software errors that have failed to be detected during the development phase, because in-service tests are typically basic tests of functionality aimed at detecting random and wear-out hardware failures. Such tests do not capture the unusual combinations of inputs and input histories that typically manifest software errors that result in significant consequences. Moreover, in-service tests consist of essentially the same test case repeated at regular intervals. Once the software passes the test the first time, it can be expected always to pass.

2.3. CURRENT PRACTICES

Safety principles in three categories are considered here, namely: safety philosophy; design and development activities (including maintenance); and management and quality assurance. In line with the approach adopted in Ref. [2.1], principles are stated in the present indicative tense, to reflect the state of affairs in an organization in which there is a high level of safety consciousness and a good understanding of issues associated with software. Within each category discussed, a safety principle is stated in bold type. In some cases this is then elaborated in medium roman type. Finally, a discussion follows in italic type of the possible implications if the principle were to be extended to safety critical software in NPPs. Some of the topics introduced here are discussed in greater detail in subsequent sections.

2.3.1. Safety philosophy for software

(a) Systems important to safety are kept simple

The functionality of safety systems encompasses only what is necessary to perform the safety functions; non-essential functions are separated out into other
systems. Where a choice exists, the simpler alternative is chosen to implement a function. Features that enhance reliability only marginally are not included if they add complexity without significant benefit. Design decisions are made deliberately to permit simple operation and maintenance.

Problems of software reliability and software safety are in part problems of complexity. Software enables the implementation of more complex functionality, with increasingly subtle failure modes. Only the functionality necessary to achieve the safety function is included in the computer system that is a direct part of the safety system. Stepwise refinement is an important concept for overcoming complexity problems because it not only allows the developer to tackle several smaller, more manageable problems instead of one large one, but also enables a more effective review by the verifier. The design of interfaces is quite difficult and is a source of errors due to miscommunication between software developers. These errors can be reduced and a program can be made more modifiable and extendable by applying 'information hiding' (see Section 3.3.4). Careful attention is paid to timing interactions, which can be a significant source of failures and are difficult to detect by testing. Simple algorithms are chosen over complex ones; simplicity is not sacrificed to achieve performance that is not required. The computer hardware used in safety systems is specified with sufficient capacity and performance that software does not have to become complex to accommodate unavoidable functional changes.

(b) An established safety culture governs the actions and interactions of all individuals and organizations engaged in activities related to nuclear power.

An environment of safety consciousness is established. Sound procedures are developed and strict adherence is demanded (this refers primarily to operational practices but the underlying motivation can be extended to design). A questioning attitude is encouraged, so as to prevent complacency and foster excellence and accountability with respect to safety.

All personnel engaged in a safety critical software project understand the safety functions of the system, and how their jobs relate to achieving that function, so that they make informed judgements in their jobs with respect to safety. Personnel are encouraged to question activities or decisions that may compromise the safety of the system. The personnel in a safety critical software project include application specialists and computer software and hardware specialists. This combination of expertise helps to ensure that the requirements for the safety system, which are generally quite well known owing to the maturity of the industry, are effectively communicated to the computer specialists.
(c) The acceptable level of safety is a balance between risks, benefits and costs.

The design constraints needed to implement the safety function may conflict with other required system characteristics. Assessing the trade-offs is a complex process that requires quantitative evaluation of the risks as well as human value judgements.

This balance leads to a need for different safety criticality levels for software, with each level having associated with it appropriate processes to warrant the level of confidence in the software that is necessary. A top down design and development process is needed to assess tradeoffs consistently. The basic decisions in identifying and specifying safety requirements and establishing the relative priorities needed for evaluating trade-offs are made in the early design and development phases. This is discussed further in Section 15.

(d) The concept of defence in depth is applied. It involves several levels of protection, including successive physical barriers. It also includes protection of the barriers by averting damage to the plant and to the barriers themselves.

No single human or mechanical failure should lead to injury to the public, and even combinations of failures that are only remotely possible would lead to little or no injury. The reliability of the physical barriers is enhanced by applying the concept of defence in depth to them in turn.

Various defences are applied to enhance the reliability of the computer system, some at the hardware level (such as watchdog timers) and some in the software (such as baton checks). Reliance on only a single type of defence is avoided.

(e) Where practicable, the fail-safe principle is incorporated into the design of systems and components important to safety. Safety is built into the system, not added on.

Fail-safe features improve the safety reliability of protection systems. However, fail-safe features are application dependent. Thus, it should be recognized that a feature designed to fail safely in one application may fail unsafely in another.

Fail-safe features should not result in unnecessary protection operations and should avoid needless stresses on the nuclear plant.

Software hazards are identified and eliminated so that the software will operate in a safe manner under all postulated failure modes that could lead to unsafe action (see Section 3.3.1). System fail-safe design features are built into the software, but since achieving a fail-safe state upon detection of a fault requires additions to
software, a balance with software complexity is maintained. If the resultant additional complexity cannot be justified by the global increase in system reliability, use of a hardware design to achieve fail-safe system states is preferred. Where feasible, a device is not used to check itself, particularly if its ability to respond correctly upon detecting a fault is in doubt. External devices, such as watchdog timers, are more appropriate.

(f) Design strives for high functional reliability and periodic testability commensurate with the safety functions to be performed. No single failure results in loss of the protection function.

Initiation and operation of the engineered safety features are highly reliable, and reliability targets are set for safety systems or functions. Safety reliability is typically measured by the probability that the system is capable of performing its safety function, per demand or over a fixed period of time. Safety reliability is achieved by the use of fail-safe design, protection against common cause failures, independence between safety and process control systems, testability, including self-checking where necessary, redundancy, diversity and simplicity. The plant design includes the capability to test automatic safety systems periodically when the reactor is in operation throughout the plant's life, with automatic self-tests where possible.

The safety reliability and testability of the computer system would need to meet the criteria for the safety category of the system of which the computer software is a part.

However, periodic system testing does not demonstrate software reliability, which refers to the level of assurance, or probability, that the software consistently performs its specified function. Such testing is used to detect random hardware failures, not software design errors. Because the software put in service may encounter a combination or sequence of inputs never tested for and for which it may fail to satisfy requirements, statistically valid random testing, based on expected operational profiles, may be an effective means of assessing software reliability (see Section 9 for further discussion of reliability testing). Self-checks within software are included commensurate with reliability targets being met and with due recognition of the efficacy of routine testing as a means of detecting hardware failures.

The single failure criterion is more difficult to meet in software than in hardware. This issue is discussed in Section 3.3.3.

(g) The protection system performs in all modes of normal operation the protective tasks that may be necessary and performs them when subjected to consequences resulting from postulated initiating events.

The design of a protection system includes consideration of all possible operating modes and conditions in the operating modes under which the protective action
may be necessary. The protective system is designed to withstand environmental or other consequential effects of the postulated initiating event, including the full range of input changes that may occur with the event.

The requirements for the software system cover the full range of operating modes and specify the protective action that must be taken when potentially unsafe conditions arise. The computer system performs reliably under all conditions created as a result of the postulated initiating event. The software performs effectively for the full range of input changes that may occur with the event.

(h) The possibility for human error in plant operation is taken into account in the design.

In the consideration of the system level design aspects, requirements for operator intervention to perform the safety critical functions are minimized. However, the design ensures that the data in the control room are sufficient for the operator to diagnose any faults that may develop and to assess the effects of actions taken. The design minimizes the likelihood that operator actions could defeat the effectiveness of the protection system. Design facilitates correct decisions by operators and inhibits wrong decisions, and provides means for detecting and correcting or compensating for error. To optimize operator performance, systematic consideration of human factors and the human–machine interface are included in the design process at an early stage of design development and continue throughout the entire process.

Clear and simple displays of the information and user interfaces needed to operate the system are provided, together with sufficient information to diagnose problems with the system and to assess the effects of corrective actions taken. Overloading the operator with information is avoided. Any on-line interactions by the operator that are associated with safety critical functions are checked for validity to provide an additional defence against impairment of the safety function via inadvertent operator error.

(i) Designs are subject to third party assessment and licensing review. Design dependability is not only achieved, it is also demonstrated.

Design and development practices are followed that are acceptable to regulatory authorities, and high quality documentation is provided to facilitate third party review by qualified personnel. Traceability is provided through each step of the implementation process.

Agreement is reached with the regulator on acceptable practices. Design decisions encapsulate functions and isolate them from one another. Portions that are likely to change on their own are segregated. Program access to data is restricted on a need to have basis. Mathematically based methods are used so that a precise, unambiguous syntax and semantics can be defined to describe the required behaviour.
of the software in all system modes and for the full ranges of inputs and so that direct comparison of the requirements with the source code can be made. Read only memories (ROMs) are used wherever possible to minimize instability, thereby helping to ensure that source code analysis remains valid. Software documentation is viewed as a software design medium to express and record design decisions and, as such, must be open to scrutiny by reviewers, including those who have expertise in areas other than software engineering. For safety critical software, this means that the design is fully analysable, so that all possible behaviours can be understood. The design documentation is layered to provide different views. Tools are used to eliminate duplication and inconsistency.

(j) Safety related systems are the subject of regular inspection and servicing to ensure the continuity of the required functional capabilities.

The system is designed to facilitate maintenance. Inspection and maintenance activities are carried out in accordance with written procedures supported by quality assurance measures. When maintenance requires disabling a particular safety system, this system should be placed in a safe state. If systems or components important to safety cannot be designed to be able to be inspected or monitored to the extent desirable, adequate safety precautions are taken to compensate for potential undiscovered failures. Engineering and technical support, competent in all disciplines important for safety, is available throughout the life of the plant.

The computer system is designed to facilitate maintenance, with recognition that the system may have to be disabled for maintenance to be performed. (When this is necessary in a multichannel voting configuration, the channel being maintained must be placed in a safe, that is, tripped, state.) Hardware and software technology that will be available and supported in the long term are used. The operating organization is prepared to provide or acquire technical support throughout the life of the product. Designers include operational maintenance issues as key design issues. Aids which facilitate and thereby encourage correct maintenance and associated testing are provided.

2.3.2. Safety principles for design and development activities

(a) Technology is used that is based on engineering practices that are proven to be effective by testing and experience. Significant new design features are introduced only after thorough research and prototype testing.

Systems and components are conservatively designed, constructed and tested to quality standards commensurate with the safety objectives. There is a balance between technological innovation and established engineering practices. The design is in accordance with applicable national or international standards, approved codes and other appropriately documented statements.
Proven software engineering methods are used; methods that are still at the research stage are not used. Proven software that can be reused is sought.

(b) Measures such as redundancy linked with diversity are used to defend against common cause failure.

Appropriate measures are taken as far as is practicable in the design to minimize common cause failures (that is, failures of a number of devices or components to perform their functions as a result of a single specific event or cause, including a design deficiency). The causes of such potential failures are examined to determine where redundancy and diversity could be used effectively to enhance reliability.

Where the instrumentation and control system is a redundant system, independent computers are used in each redundant part. Diverse software, on its own, provides some improved protection against common cause software errors, but does not guarantee independence of errors. In particular, if multiple versions of software are developed from a common specification, errors in the specification may translate into errors in all versions of the software. The best protection against common cause software errors is to ensure that the requirements are correct and complete and fully understood by the software development and testing team, and that a rigorous software development process is followed.

(c) For system and component design, acceptance criteria are used in the form of engineering rules.

A well defined software development process comprising specific methods and acceptance criteria for each phase of the process is applied to give the necessary assurance of confidence in the software product. Computer programs are written in language formats that are easy to understand, and they are well documented; an example of a clear format is one divided into readily recognizable sections (modules) with distinct purposes.

(d) Appropriate tools and techniques are used to keep the development of digital systems under intellectual control and to manage complexity.

These include formal specification methods, structuring methods and test coverage analysers. The use of high level languages can eliminate from consideration a whole detailed level of complexity that was never inherent in the design at all. High level languages also provide inherent self-checking and rigour inducing characteristics that may be used.
2.3.3. Safety principles for management and quality assurance

(a) Management establishes the functions and qualifications required of personnel involved in design, production, operation and maintenance programmes, and ensures that only qualified personnel perform these functions.

Personnel qualifications are established for each task within the software development and maintenance process, including the execution of the quality assurance programme (for example, the design and development, verification and audit functions). Only qualified staff participate in the process. Designers' handbooks containing recommended practices are used to help ensure that the required level of quality in all phases of production and maintenance is achieved.

(b) To achieve the required quality, well proven and established techniques and procedures supported by quality assurance practices are used in manufacturing and construction.

Manufacturing and construction are guided by detailed specifications for processes and products and for methods of testing and inspection. It is common for suppliers of important safety related equipment to have their competence checked and certified by third parties. The manufacturer establishes procedures for process and document control, identification and control of materials and components, inspection and test schedules, maintenance of records, hold points and corrective procedures for deviations: all of these being subject to a hierarchy of quality assurance practices. Although the manufacturer has immediate responsibility for quality, the operating organization fulfils its general responsibility for safety by reviewing and auditing the practices and documentation of the manufacturers and contractors. Preferred equipment is equipment of a proven design wherever possible, with a predictable failure mode, designed to facilitate test, maintenance and repair, and which has an operating life equal to or longer than the scheduled life of the plant.

Specific procedures are prepared for the various steps of the software engineering process. A detailed plan is prepared for the production of software. The operating organization is sufficiently knowledgeable to judge the adequacy of the software developer’s practices and products. Audits are performed to check the adequacy of the practices and product. No unproven software engineering practices are accepted.

(c) Quality assurance is applied throughout all activities to ensure with high confidence that all items delivered and services and tasks meet specified requirements.

The responsibility for achieving quality rests with the individual performing the task; others verify that the task has been properly performed, and yet others audit the entire process.
The organization’s quality assurance programme extends into the software
development process, and also covers configuration management and change control
after the software is delivered. The software engineering rules for independent verifica-
tion and validation and testing, with independent supporting audits, are applied.

(d) The assignment and subdivision of responsibility for safety are kept well
defined throughout the design and development phase and during any
subsequent modification.

Interfaces are established between organizational entities within the design
organization and between the design organization and its client and other design
organizations. Controls pertaining to design interfacing are established and include
the assignment of responsibilities and the issue of documents to and from interfacing
organizations.

A safety role is defined and assigned at the beginning of the project and
pervades all aspects of the project to ensure that all decisions are consistent with the
safety requirements. Interfaces between the software system and hardware compo-
nents within the system, and with the operator, are established, documented and
controlled.

REFERENCES

[2.1] INTERNATIONAL NUCLEAR SAFETY ADVISORY GROUP, Basic Safety Princi-
[2.2] INTERNATIONAL ATOMIC ENERGY AGENCY, Code on the Safety of Nuclear
[2.3] INTERNATIONAL ATOMIC ENERGY AGENCY, Safety-Related Instrumentation
and Control Systems for Nuclear Power Plants, Safety Series No. 50-SG-D8, IAEA,
[2.4] CANADIAN STANDARDS ASSOCIATION, Requirements for the Shutdown
Systems of CANDU Nuclear Power Plants, CSA CAN3-N290.1-80, Rexdale, ON
(1980).
[2.5] BLOOMFIELD, R.E., BRAZENDALE, J., SafeIT: A Framework for Safety Stan-
dards, Interdepartmental Committee on Software Engineering, Department of Trade
3. THE ENGINEERING CONTEXT

3.1. ISSUES

System engineering for NPPs has developed some basic principles and methods to ensure safety. The principles of simplicity, defence in depth, fail-safe design and the single failure criterion and others are discussed in Section 2. It is important that the use of software does not violate these basic principles. The methods that have been used to apply these principles to hardware systems should be re-evaluated for systems that use software to ensure that the principles are still being upheld.

Section 3 discusses the following system safety engineering methods and indicates how they may be applied to software:

- hazard analysis
- mathematical analysis
- design diversity and redundancy
- modularization
- reliability estimation
- quality assurance.

The foundation of Section 3 is the idea that safety is a system issue and must be evaluated in this context. Software safety should not be separated from system safety [3.1]. System and hardware design decisions which are based on basic safety principles must not be compromised by the use of software. Ultimately, the concern is the system as a whole, defined as [3.2-3.4]:

"A bounded physical entity which achieves in its environment a defined purpose through the interaction of its parts."

Software alone cannot be a system by this definition because it is not a physical entity. When software is executed on a computer, the combination is a system, just as a hardware design, when realized in physical components, is a system. Any separate analysis of the software should be related back to the safety of the system as a whole.

3.2. EXPERIENCE

3.2.1. Advantages and disadvantages of software

The use of software in NPPs, particularly in safety related systems and safety critical systems, has been an area of concern for a number of years. In some
countries, such as Canada, software based systems have been an integral part of nuclear plant control systems for decades, but generally with some backup safety systems implemented without software. In other countries, such as France, safety systems were implemented with software before the introduction of software into reactor control systems. In still other countries, such as Germany, software based systems have been regarded as insufficiently trustworthy for reactor control and safety applications. However, in all countries with NPPs, there is now increasing pressure to use software based systems simply because the older analogue technology is being phased out by the control and instrumentation industry in favour of digital, software based devices.

There are significant advantages in the use of software based systems over analogue systems:

- it is possible to implement more complex functions;
- it is possible to achieve greater precision;
- a larger amount and variety of information can be collected and used;
- the user interface can be made more flexible;
- it is easier for the system to detect and handle anticipated internal faults;
- functional changes can be made without physical changes or even physical access;
- standard processors of known reliability can be used in many applications.

However, the disadvantage that has prevented software based systems from being used in the nuclear industry as widely as they have in the other industries is the element of trust necessary. Because of the complicated and discontinuous nature of software, software faults are more difficult to detect and protect against than hardware faults. Failures which occur because of software faults are difficult to predict and the resulting system reliability is difficult to determine.

3.2.2. Life-cycle of a software based system

The development of software must be put into the context of the development of a system. The life-cycle of a software based system is the period of time that starts when the system is conceived and ends when it is no longer available for use. The life-cycle typically includes a requirement stage, design stage, implementation stage, test stage, installation/integration stage, validation and commissioning stage, operation and a maintenance stage [3.5, 3.6].

Each stage uses information developed in earlier stages and provides information for later stages. When a system is conceived from nuclear safety requirements, the subsequent stages may be to develop protective system requirements, protective system design, computer system requirements, computer system design, software requirements and software design. Each stage is characterized by its product, the
specifications, documents and programs that should be produced during that stage.
See Appendix F of Ref. [3.5] for detailed guidelines in this area.

In practice, it is usually not possible to complete the stages one by one in the
sequence which is specified. As the design progresses, faults and omissions made
in the earlier stages become visible and require iterations on these earlier stages.

It is important that a structure of stages be identified before the development
starts, and that none of the identified stages be omitted. Some of the most important
benefits are:

(1) Specifications produced during the early stages enable the system designers
and the software engineers to reason about the functions of the system and to
make predictions about its behaviour before it is built. Misunderstandings
about requirements (a major cause of software failures) can thus be resolved
earlier in the development process.

(2) Every product can be systematically reviewed and checked after each stage to
verify that it is consistent with the specifications produced in the earlier stages;
faults and omissions can thus be discovered at early stages and repaired at less
cost (that is, with fewer iterations on the previous stages) than if their effects
propagate and affect later stages.

(3) Specifications of functional requirements have a chance to be reviewed and
become mature before decisions concerning the system and the software
implementations are made. These implementations can be more efficient and
less likely to impose unnecessary limitations on the system capabilities.

(4) Quality assurance plans can be established at the start of a project on a stage
by stage basis with personnel responsibilities and qualifications, hold points,
acceptance criteria, audits and reviews defined for each stage.

Clear identification of the stages of the iterative software development process thus
offers many advantages. Various parts of this report make reference to the division
into stages suggested here, especially in the sections on Modelling (Section 5),
Documentation (Section 6), Verification and Validation (Section 8), Testing
(Section 9), and Personnel Qualification and Training (Section 14).

3.3. CURRENT PRACTICES

3.3.1. Hazard analysis

System hazards are conditions or system states that can lead to accidents caus­
ing death or injury or damage to the environment [3.1, 3.3, 3.4, 3.7, 3.8]. Hazard
analysis refers to the examination of a system from the point of view of safety; that
is, identifying and assessing potential hazards with the objective of using this infor­
mation to eliminate hazards from the design. If it is not possible to eliminate hazards,
then the likelihood of their occurrence should be reduced and mitigation of their effects included in the system design.

One use of hazard analysis is to categorize systems and subsystems according to their level of 'criticality' [3.3, 3.4]. The system level hazard analysis should identify whether the system or subsystems can prevent unacceptable accidents or reduce their risk. Various schemes have been proposed for relating criticality levels to the severity and probability of the accident that the system is intended to prevent or mitigate. Much work is being done in this area and many categorization schemes have been proposed [3.3, 3.4, 3.9–3.11].

In the nuclear industry, safety systems have been identified as those systems which ensure the safe shutdown of the reactor or residual heat removal from the core, or which limit the consequences of anticipated operational occurrences and accident conditions. The requirement for safety systems comes about from a high level hazard analysis (safety analysis) of the entire plant. This definition corresponds in many ways to the definitions of safety critical systems in other industries (and so the term safety critical will be used to refer to safety systems in this report). However, it should be noted that owing to the multiple layers of protection that are built into NPPs and the existence of a single safe state (shut down), the reliability targets (when used) for individual safety systems in NPPs are sometimes less stringent than those in other industries. For example, aircraft control systems cannot have as many layers of protection, and an aircraft in the air must remain under control and must go through a complicated control process (landing) in order to reach a safe state.

Hazards can be analysed for accident severity and likelihood and a subset selected for further analysis. That analysis may include use of a technique such as fault tree analysis to trace a hazard back to its possible causes. Alternatively, event trees can be used to trace a primary event forward to determine consequences. If a system contains software, the fault tree can be extended into the software itself in order to find paths through the software that can lead to the undesired (hazardous) conditions. Software fault tree analysis can also provide confidence that such paths do not exist and guide in the use of software design techniques to make the software fail-safe [3.12, 3.13]. The practicality of this process depends on the size and complexity of the software [3.7].

The result of a hazard analysis can be:

1. the identification of design faults that could lead to a specific hazard;
2. the identification of weak points where a single, possible failure could lead to a serious hazard;
3. the identification of areas where the design could be made more fault tolerant; and/or
4. the identification of areas where the design can be made safer; for example, where changes in the code can make it fail-safe.
The basic technique of fault tree analysis is described in Ref. [3.14] (for electromechanical systems). Reference [3.12] describes how the technique can be applied to software with examples in Ada. Reference [3.8] is a practical example of the use of fault tree analysis.

The main advantage of hazard analysis is that it provides an independent path to an improved design. Normal safety system design and development focuses on the question 'How can the system be made to provide this function?' Hazard analysis and other system safety techniques ask the question 'How can the system be made safer?' The techniques answer this question by determining how the system can fail or interact with other systems leading to an unsafe state, so that the corresponding hazards can be removed.

However, hazards analysis alone cannot ensure complete safety. It is difficult to be sure that the analysis is complete and correct for entirely new systems. For systems that are similar to existing systems, it is important to make use of historical information and experience concerning previous failures, accidents and near accidents as well as previous hazard analyses.

### 3.3.2. Mathematical analysis

Mathematical analysis is the foundation of most engineering practices. The extension of mathematical analysis to software is generally called formal methods. Formal methods for developing software are mathematically based techniques for specifying software and module properties and for verifying each level of specification against a higher level. Formal methods can be used to specify requirements, modules and interfaces and to describe actual system properties. The mathematical basis gives a means to precisely define quality attributes such as consistency, completeness and correctness. A formal method typically includes a specification language with mathematically based syntax and semantics. See Ref. [3.15] for a survey of formal methods.

The use of a formal method generally begins with a system or subsystem specification written in a formal language. Such a specification can be checked mechanically for consistency and domain coverage, and various properties of the system being specified can be determined directly from the specification. Such analyses can reveal problems much earlier than they might otherwise be discovered. A formal specification can also form a basis for subsequent testing and acceptance of the final product; a detailed test plan can be developed directly from the specification, independently of the other development work.

As the design and development proceeds, formal methods can be used to specify the interfaces and the properties of each component of the system; static analysis

---

4 In contrast to dynamic analysis, static analysis is performed without exercising the system or executing the program. It is used to determine system and program properties that are universally true for all possible execution conditions.
can determine the well-formedness of a component with respect to its control and information flow, and data usage. For a software based product, the final step is the translation of the component requirements into a source code language having a formal definition.

Formal documentation is discussed further in Section 6 and formal verification in Section 8. For examples see Ref. [3.16].

The advantage of the use of formal methods is that precise, mathematical techniques can be used to analyse and verify the specifications, design and code. A formal, unambiguous specification can be an important tool to communicate requirements to developers as well as facilitating discussions amongst the development team. Although the mathematical techniques themselves may be subject to human error (and so should not be used alone), they are complementary to the technique of verification by testing. The precision and mathematical foundation of formal methods gives reviewers and regulators more confidence in the quality of the product.

There are a number of difficulties with the use of formal methods, which are gradually being overcome:

(1) The most difficult step is to translate vague, incomplete, ambiguous or unavailable user requirements into a formal specification. It is difficult to be sure that the formal specification contains all of the user’s requirements and that they have been correctly specified. However, formal specification generally reveals ambiguities and incompleteness in the informal requirements which are easier and cheaper to fix early in the development process rather than later.

(2) Formal methods have only been used successfully on fairly small systems. Cost and effort rise rapidly with increasing system size and complexity. Costs may be reduced as better tools become available.

(3) Formal languages are often at least as difficult to learn, write and understand as a source code language. Specially trained and skilled personnel are required to work with these languages.

(4) From a project management and budget point of view, it is difficult to sell the idea of spending a substantial fraction of the project funds on the initial specification. This is changing now that managers have seen some examples of the expense and difficulties in debugging non-formally specified software.

(5) There is no consensus in the software industry concerning the quantitative value of formal methods or which formal method is best to use. However, recent standards for safety critical software [3.5, 3.17] are beginning to recommend or require the use of formal methods.

3.3.3. Redundancy and diversity

High safety system reliability has been achieved traditionally by using high quality components in a configuration of redundant instrumentation channels, with
a voting arrangement so that the system still operates effectively despite the failure of one or more channels. This redundancy/coincidence arrangement has been a hallmark of the design approach in the nuclear industry. The approach facilitates the achievement of systems with high reliability. It also facilitates demonstration that target reliabilities are achieved in practice, because each channel in turn can be tested on-line, without impairing the overall system or spuriously initiating it. Thus, the redundancy/coincidence principle permits high reliability to be achieved in the design stage and demonstrated during the operational stage.

Extending this concept to software, redundant versions (copies) of software can be implemented in computers associated with different channels of a system. However, unlike hardware, software does not physically fail or break, so any software faults that could lead to impairment of the safety system will be design faults which will exist in all channelized versions. Such faults could result in a common failure of all channels, resulting in impairment of the system. This issue is sometimes dealt with by introducing diversity into the design, so as to reduce the likelihood of common design errors. The theory is that if diverse designs are used, the probability that the same design error exists in both designs is substantially reduced.

The increase in reliability of hardware systems achieved through diversity is generally not quantified. Rules of thumb have been developed to achieve hardware diversity, such as: the use of different design teams working independently; the use of equipment purchased from different manufacturers; the use of different principles of operation. It is accepted practice that, if these guidelines are followed, the resulting degree of diversity is sufficient to justify the assumption that diverse designs will fail independently.

The principle of diversity is sometimes extended to software in order to try to avoid common design errors. The use of different programming teams, and different programming languages and hardware platforms, is thought to provide greater protection against software design errors than would be achieved by redundancy alone. A number of methods have been proposed for incorporating design diversity into software based systems. N-version programming is a method whereby a number of separate people or teams design and code software from the same requirement specification. These separate versions of the software are run in parallel (on redundant or diverse hardware) and the system outputs are determined by a vote amongst the separate outputs. The use of recovery blocks also requires a number of independently developed modules of software, but instead of running in parallel, there is a strict order of execution with an acceptance test to determine whether the outputs are valid. If the outputs of one module are determined to be invalid, then the starting conditions are regenerated and another diverse module is executed. See Ref. [3.18] for a discussion and comparison of these two methods.

Many examples of N-version programming experiments have been published, the most widely referenced being Ref. [3.19], which shows that experimental evidence does not support the assumption of independence of errors in independently
written software. However, there are benefits to be gained by the use of software diversity. Reference [3.20] summarizes the results of a number of experiments and concludes that the ratio between total errors and common errors is of the order of 10:1. Reference [3.21] describes an experiment which is particularly relevant to the nuclear industry since the sample software was for a reactor trip system.

The goals of software diversity are to increase the fault tolerance of the system and to reduce the possibility that a single human error will cause an unsafe failure. To this extent, some success has been achieved, but it has proved controversial to quantify the increase in reliability and safety. The difficulties are that having to develop multiple versions of the software multiplies the software costs and there is no guarantee that more than one designer will not make the same error. There are indications that diversity leads to an improvement in overall reliability, but not necessarily by the amount that would be implied by assuming that diverse versions contain totally independent errors. Forcing the use of diverse techniques seems to be better than mandating independence and leaving diversity to chance. Using diverse (independent) teams for development and verification reduces the chance that a design error will be missed during verification.

It should be noted that common failures due to software design errors are just one type of common error that plant designers and operators have to consider. There can also be common design errors associated with hardware components that are used in all redundant channels. Alternatively, an operational event (such as exposure to a harsh post-accident environment) may simultaneously impair all channels. The latter issue is often dealt with by physically separating and environmentally qualifying the instrument channels, so that a single event cannot impair multiple channels.

### 3.3.4. Information hiding and reusable modules

'Information hiding' is a design technique which has changed with the evolution of software development methods and has been referred to by many names. The term information hiding applies to the division of a system or a body of software into smaller components (or modules) so that the interactions between the components are minimally dependent on the internal operation or internal state of each component. Each component 'hides' the information about its internal state and operation behind a concise, stable interface specification (and thus confines it in scope and visibility to the unit bodies). The internal operation and data storage of a component are inaccessible to other components. Information hiding is essential to the development of reusable software modules, and to the feasibility of reviewing and verifying large pieces of software. The technique is based on the engineering principles of separation of concerns and design simplicity.

Similar techniques have been used by hardware developers with great success. For example, integrated circuit chips are available that have precise specifications of their external interface, yet their internal design may vary and the chips may be
supplied by a variety of manufacturers. Such parts may be removed and replaced independently, and the specification of the interface assists system designers in coming up with new configurations to meet changing requirements. The availability of standard parts that can be reused in many different applications is a key factor in developing inexpensive, reliable systems.

Reference [3.22] contains a detailed explanation of how information hiding can be done in software development and includes an example of how the technique was applied to a multiprocessor data acquisition and transaction system.

The potential benefits of information hiding include:

1. reducing the likelihood of errors and of error propagation owing to decreased connectivity between modules;
2. allowing separate, parallel development of modules;
3. allowing the system to be built incrementally from separately compilable modules;
4. making the design easier to review and verify;
5. making the system easier to maintain and change; and
6. allowing reuse of modules between systems.

The main drawbacks are that:

1. designing information hiding modules is not easy: no prescriptive techniques exist for achieving a good information hiding design;
2. criteria for what constitutes a ‘good’ information hiding design are not available;
3. the link with improved system safety and reliability is indirect;
4. when a designer is focusing on a particular application, it is difficult to conceive of modules that hide application specific information and that will have more general applicability;
5. it can be more expensive, in the short term, to develop information hiding modules, and difficult to justify the expense on the basis of reduced maintenance costs and reduced cost of future system development.

Even when the expense of developing reusable modules is accepted, there is still a problem with getting software developers to use them [3.23]. This may improve with the growing use of object oriented techniques and with developments in modular programming languages (see Section 7).

### 3.3.5. Reliability and safety

In some countries, safety systems in NPPs must meet defined reliability targets. In general, reliability refers to the probability that a system will correctly perform the functions stated in a requirements specification. In the past, the major concern regarding the ability of safety systems in NPPs to meet requirements has
been random failures, due to wear or ageing, of hardware components. Reliability predictions take into account the probability that individual components may fail at any time and the effect of such failures on system performance. Such predictions assume that if all components are working properly, the system will perform its required functions correctly; that is, there is no systematic design error. More recently, the impact of systematic design errors has been recognized, and in some applications limits on reliability claims are used to reflect these uncertainties (such as in the United Kingdom Design Safety Guidelines).

It is recognized that, in general, a system may be reliable but unsafe. If the design intent of the system excludes safety considerations, then the system may operate in conformance with its requirements in a reliable manner, but it may prove to be unsafe. An example would be an elevator/lift that closes its doors reliably upon request, but which also crushes anyone caught in the path of the doors when they close.

In the context of a safety system in an NPP, dependability takes two forms. The first may be referred to as safety reliability, which is the probability that the safety system will perform its safety functions correctly. The second is referred to as production reliability, which is the probability that the safety system will not actuate spuriously (that is, when not required). Thus, a reliable plant safety system should be both safe and not detrimental to the production of electricity. In this report, discussion of safety system reliability refers only to safety reliability, because the production aspects are of secondary concern in this context. However, it is recognized that spurious shutdown of a reactor is undesirable.

The derivation of safety reliability is usually done in the context of a probabilistic approach to risk assessment. Within this framework, the concerns are: how the safety reliability can be measured; what the target reliability levels should be; how to build a system to achieve these target levels; and how to evaluate whether a design meets a specified reliability target.

In the nuclear industry the safety case usually relies on both probabilistic and deterministic arguments. In the elevator/lift example, it might be possible to show that the system closing the doors on a passenger could not exert enough force to cause severe injuries. This case would then be excluded on the basis of a deterministic argument. However, movement of the elevator while a passenger is trapped in the door could also cause severe injury, and consequently this case would have to be analysed to determine the probability of its occurrence.

Deterministic arguments rely on a model of the system and of its environment. However, there is always the need to establish the validity of the model used and this might require a mixture of deterministic and probabilistic reasoning.

In some circumstances it may be easier to show that an event is impossible than to show it is very unlikely (such as $10^{-9}$ failures per demand) because of the possibility of analytical evidence to support the claim of impossibility. Consider a small piece of software performing some propositional logic. The correctness of this
software with respect to an engineering specification containing look up tables or
decision tables could be readily established either by analysis or by exhaustive test-
ing. The claim could then be reasonably made that a failure is impossible. However,
as this example is scaled up, doubts emerge and gradually become more significant:
it becomes necessary to consider errors in the proof, the dependability of the proof
process, the problem of tool support, and so on. All these doubts can be addressed
by either deterministic or probabilistic arguments.

The use of statistical arguments to verify complex systems by drawing conclu-
sions directly from system behaviour is generally only of use for systems with
moderate reliability requirements. For other systems, it is necessary to rely on deter-
ministic arguments backed up by a consideration of the residual doubts concerning
the validity of the model and of the analysis. Therefore it is possible to ensure higher
levels of reliability only for those systems for which sufficiently accurate models
exist that are subject to logical analysis. This is one of the motivations for using
formal methods to ensure the reliability of digital systems (see Section 5).

As far as software for NPP safety is concerned, probability safety assessment
(PSA) is intended to provide appropriate probabilistic measures and targets for soft-
ware reliability. However, no software development method gives enough confi-
dence yet that it is able to deliver a product of a given required reliability, and no
consensus yet exists on methods for evaluation and verification that a required reli-
ability has been achieved.

In the development and manufacture of hardware systems, reliability and
quality control are important engineering matters that have a direct impact on
product cost and success. Development methods, manufacturing techniques and test-
ing procedures have been developed to ensure high levels of reliability and quality
in both commercial and custom products. Software engineering has not yet reached
the same point; there is still disagreement about what reliability and quality mean
with respect to software. Methods for measuring and achieving software reliability
and quality are not mature [3.24]. Some workers claim to have brought their soft-
ware quality control methods to a level comparable with those for hardware quality
control [3.25], but these methods are not yet in widespread use.

In the area of software reliability evaluation, there appear to be three lines of
research:

(1) investigating ways to provide measures of reliability deterministically (by
looking only at the code and its documentation) or using analytical techniques
to show the absence of certain types of faults;
(2) investigating testing methods and ways of interpreting test results and operat-
ing experience data to derive useful reliability estimates; and
(3) investigating how some measures of software reliability evolve throughout the
software development process (reliability growth).

The first line of research has resulted in qualitative or comparative measures
of reliability, but these measures have resisted quantification and calibration. A large
number of models of software development and reliability growth have been proposed by investigators of the third line of research, but none is generally accepted because of a lack of good data and evidence. Investigators of the second line of research claim that there is not yet a good enough measurement of reliability on which to base the third line of research.

3.3.6. Quality assurance

One of the major motivations behind this report is the desire to improve the quality of software in NPPs. 'Quality' is a term which defies definition, but many people have listed characteristics of software which they associate with quality. For example, software quality attributes which are important to safety critical applications include: correctness, completeness, reliability, security. Other quality attributes which facilitate the assurance of safety are: simplicity, structuredness, testability, maintainability, and understandability. Annex C of Ref. [3.6] lists and defines these and other quality attributes.

It is important that the desired quality attributes of a particular software be specified at an early stage in the lifetime if they are to be satisfactorily achieved. A quality assurance plan should then be developed which specifies how those attributes will be achieved (see Sections 6 and 7), how the software will be verified (see Sections 8 and 9) and how the quality attributes can possibly be measured (see Section 10). The quality assurance plan should address the full lifetime of the product. Quality can only be built in, not added on at the last stage of the development.

The quality assurance plan should also address the requirement for independent assessment of the software. It is impossible for people to check their own work because they tend to make the same mistakes in the review as they did in the development, and they are not motivated to find errors in their products. The quality assurance plan should ensure that each product and document is independently reviewed. All software should be verified and validated by people independent of the developer and under separate management to ensure proper motivation to find errors and safety concerns.

3.4. SUMMARY AND FOLLOW-UP

This section has evoked a number of system engineering methods that are now being applied in software development. These methods have had some success when they have been applied, but there is still considerable debate over how widely they should be applied, and over their costs and benefits. Although their proponents claim that they can reduce the overall system cost, they have not yet received widespread acceptance. Their use generally requires greater advance expenses while cost
reductions usually occur later as a result of reduced maintenance, and of increased reliability and safety (items whose costs are notoriously underestimated). The methods are clearly beneficial, but it is impossible to quantify their benefits, or to conduct controlled experiments to demonstrate their advantages.

The methods described in this section are not mutually exclusive; instead, they should be used complementarily, even within the same project. The choice of a particular set of methods is far from being independent of the relative importance of safety in the functional specifications, and of the level of the risks associated with the potential failures.

REFERENCES


[3.3] UNITED KINGDOM MINISTRY OF DEFENCE, DIRECTORATE OF STANDARIZATION, Requirements for the Analysis of Safety Critical Hazards, Draft Interim Defence Standard INT DEF STAN 00-56/1 DRAFT, Ministry of Defence, Glasgow (May 1989).


4. MANAGEMENT

Management practices for software development projects are prime factors in determining the success or failure of the resultant software product in fully meeting its requirements. Management decisions touch on every aspect of the software production processes and insufficient management experience or poor management decisions can render otherwise good development teams ineffectual.

With respect to safety, accident investigations often find that the root causes of accidents are management weaknesses [4.1]. The degree of safety achieved in any system is directly related to the priority and emphasis that management assigns to safety relative to other system goals.

Management responsibilities include the following aspects:

1. Establishing safety policy and culture.
2. Formulating strategies for licensing and achieving safety as part of the overall planning process; regulatory control to ensure that these safety goals are achieved.
3. Communicating the concepts of the system to all participants in software production.
4. Integrating software engineering with system engineering.
5. Obtaining and supervising qualified personnel and maintaining their qualification.
6. Providing adequate tool support.

In the following sections, each one of these aspects is briefly discussed, including the nature and significance of the issue, past experience in addressing the issue, present recommended practice and further work needing to be done.

4.1. ISSUES

4.1.1. Safety policy and culture

For safety critical software projects, as for other safety critical projects, management is responsible for establishing safety policy and ensuring its cultural acceptance by personnel, so that all design decisions are made with safety as one of the most important criteria. This should demand personal dedication and accountability of all individuals working on a nuclear plant. A deficient or inconsistent safety policy, or a deficient implementation of the policy, will result in greater probability of decision errors throughout the development, leading directly either to increased rework costs or possibly to an unsafe result.
Safety policy, and the means for its implementation, are defined and communicated to personnel in the form of organizational responsibilities and goals, and in the selection and definition of standards, methods and procedures.

4.1.2. Strategies, planning and licensing

Managers must be capable of the task of assessing trade-offs among a very complex and demanding set of requirements associated with a software project (since the use of software is usually only practicable if functional complexity exceeds a certain threshold level).

Management must determine what constitutes an adequate software product (code and documentation) and how it can be produced, measured and assessed, efficiently, for each category of application risk. Categorization of software, according to the impact of software errors on safety, will allow tailoring of software production methods and quality programs to permit cost effective software engineering.

One approach has been to assume that good products will be produced if a strong software quality program is in place. Owing to the difficult nature of software projects, especially those concerned with safety, the assignment of tasks, responsibilities, authority and accountability must be explicit to ensure that there are no omissions and that project goals are met in a visible, reliable and cost effective way. However, there is no international consensus on what is an acceptable quality program for safety related software engineering. Design of software important to safety requires rigour along with comprehensive analysis and meticulous attention to detail. Careful gradual refinement from high level concept to detailed implementation using a well defined process is mandatory in order to achieve the consistency and coherence necessary.

For any project it is necessary to be able to measure objectively process and product characteristics (a) to determine how well project goals are being met during development, and (b) to allow comparison with other projects. This will provide a feedback mechanism on the effectiveness of project procedures and promote process improvements on future projects.

It is required that design decisions and the associated rationale be adequately documented as part of the project documentation to allow an independent assessment of major design decisions. Formal audits play a part in this programme and are invariably a contractual necessity but the software quality assessment process [4.2] is a more effective tool. The regulator in each country will schedule audits and inspections during the design process to define acceptance criteria for safety critical software and develop an approval strategy.

The usual approach to licensing safety systems is for the licensee to design the system according to methods and standards that are accepted by the engineering community and by the regulatory authorities. This approach is difficult to follow when there are no generally accepted standards for developing safety critical software, but
management must overcome this difficulty if it is to achieve an efficient and risk reduced licensing process.

4.1.3. Integration of software and system engineering

The issue of software engineering and safety is not yet adequately treated within a systems engineering framework [4.3]. System engineering and hardware engineering need to be performed as interrelated processes with software engineering in the context of the overall system safety. This will ensure that the software engineering decisions are consonant with the objectives and approaches of the overall system safety.

An important decision for management to make is how far to exceed pre-defined targets in making a system fail-safe. In theory, it is sufficient to meet the safety targets, which are agreed to in advance by the regulatory authorities, but in practice there is pressure to go the extra mile. However, each incremental amount of safety insurance carries with it increased cost in terms of implementation and ongoing maintenance, plus a possible spurious initiation into the safe state, reducing the plant's availability for its intended purpose.

This issue is particularly relevant to computer based safety systems, because computers have greater capacity for checking the health of other components in the system than do the hardware devices replaced. There is a temptation to build in ever increasing safety measures, but at the possible increase of complexity and cost.

4.1.4. Personnel and supervision

The success of any software project is dependent on the ability to acquire the necessary resources to carry it out properly.

Management is responsible for the provision of sufficient personnel, for ensuring that the qualifications and skills of personnel are appropriate for the tasks, and for maintaining this qualification. Management must also instil personal responsibility.

The quality of an individual's work, or the product of a work group, must be reviewed upon completion. Feedback, in the form of review comments and change actions, must be provided to close the change loop, resolve problems and confirm design changes.

To some extent, safety critical software projects involve the acquisition of knowledge and skills which some personnel feel are not portable or transferable. This can have a demotivating effect. Failure to address adequately issues of personnel selection, management and motivation will result in a shortage of qualified personnel, which in turn will result in:

— longer and repeated learning curves with attendant higher cost;
— a greater error and inconsistency rate in the delivered design, with resulting increase in rework costs or possible unsafe results.

4.1.5. Provision of tool support

Development, testing and maintenance support facilities are essential elements for producing a quality software system at reasonable cost and maintaining it throughout its entire life-cycle. However, because of this role, it is necessary to qualify these facilities to ensure that they are worthy of the confidence placed in them and would not impair safety.

Care should be taken not to depend on tools to undertake tasks beyond their capability. For example, automated tools can perform checks, sorts, comparisons and other simple, tedious or routine tasks better than humans, but they cannot substitute for humans where judgement is involved. In many cases where a tool is not proved, diverse tools may be required to provide confidence in the process.

The exacting nature of software development work in order to meet the stringent requirements of a software quality plan place heavy demands upon management. In order for management to exercise effective control, they must be provided with the appropriate tools for determination of progress, scheduling and budget control.

4.2. EXPERIENCE

4.2.1. Safety policy and climate

For projects developing highly dependable software, the volume of documentation on standards, guidelines and procedures can be high. This leads to difficulty for personnel to assimilate the material and consequently difficulty in maintaining a consistent approach across the full set of safety policy documents.

Software projects governed by a rigorous software engineering process are labour intensive for some aspects of the work, resulting in large peaks in effort at some points in the projects. These requirements of effort can be satisfied by using temporary personnel or by subcontracting the work. Special attention needs to be paid to inculcating safety culture in these individuals and setting the standards by which the work will be done.

Safety critical projects are subject to external review or certification by an organization appointed by the customer or regulatory body. In the past, safety policy and software engineering processes have not always been clearly defined to satisfy the requirements of this external body. The regulatory bodies look at the requirements and policy established by other similarly placed organizations around the world in order to establish their own. Inward looking design organizations have
established programmes based on their own understanding of need and effectiveness. As a result, significant shifts in policy mid-project have been experienced, forced on the project by the regulatory body in line with public expectations. This shift in policy results in costly reworking that may have been avoidable with sufficient planning and regular consultation with the regulatory body.

4.2.2. Strategies, planning and licensing

In the early 1980s, safety systems containing software were licensed successfully in France and Canada. The software did not become an issue that impeded overall licensing of the plant and, in both countries, new plants were subsequently planned with software as an integral part of safety systems.

Subsequently, regulators as well as licensees have become more aware of the issues associated with safety critical software, and the licensing of software has become a more prominent part of the overall licensing process. The objective of applying agreed upon standards exists; however, the absence of such standards has led to significant licensing delays in at least one well publicized instance. Examples of approaches used by regulators (Darlington Nuclear Generating Station in Canada and Sizewell B in the United Kingdom) are given in Refs [4.4, 4.5]. In both of these cases, the licensing process has been made more difficult because of the absence of available, agreed upon standards. Demonstrating or proving software reliability is a challenging problem. As a result, there is a focus on establishing good software production processes on the premise that the quality of a software product is largely governed by the quality of the processes used to create and maintain it. The standard model for software production has been the classic waterfall model. Owing to the limitations of this model, other models have been developed, such as the iterative waterfall model, the evolutionary model, the incremental build model, the preplanned product improvement model or the staged development model.

Specifically, organizations with immature processes are often deficient in one or more of the following areas [4.6]:
— project planning;
— project management;
— configuration management;
— software quality assurance.

These deficiencies can result in budget and time overruns and marginal quality products.

The use of metrics for feedback on management productivity has met with some success, but the use of metrics as indicators of software quality has not been widely accepted (see Section 10).

For safety related software, the IEC 880 standard [4.7] makes an attempt at providing measurable criteria for assessing the adequacy of a software engineering process.
4.2.3. Personnel and supervision

For projects involving software, it has generally not been difficult to obtain sufficient personnel, except where project managers have underestimated the need or realized too late in the project that they were underresourced. (Usually it is not possible to recover from this situation by adding an influx of personnel late in the project.) More significant problems have been encountered in obtaining personnel of adequate competence.

Safety critical software projects require specialized expertise, knowledge and skills which are not generally used in other software projects (although these techniques have been used to varying degrees in non-software applications). Such expertise includes:

— application knowledge;
— formalized design techniques;
— test design;
— hazard analysis;
— verification and validation.

Because emerging software engineering technologies (such as formal methods) are being applied to the design of safety critical software, some of the expertise is in short supply.

Personnel from within general design organizations often lack the specialized expertise required for safety critical work. In the past the individual’s quality was measured by his or her track record and insufficient weight was attached to the individual’s fundamental understanding of the issues in software engineering and design for safety (see Section 14).

Many organizations do not consistently track the quality of their software and software processes in order to improve quality. Design decisions are not adequately captured, resulting in the loss of their rationale.

Internal audits (see Section 8), if carried out at appropriate times during the project life-cycle, can provide valuable feedback to management in identifying difficulties in compliance with specified requirements or procedures. Deficiencies may be identified to enable corrective action to be taken without significant cost or delay. However, experience indicates the following problems:

(1) auditors tend to focus on easily verifiable items that are monitored by software production management and that are less significant to the overall product quality;
(2) insufficient time is dedicated to acquiring sufficient knowledge in depth to allow useful observations;
(3) the audit process tends to put audited personnel on the defensive and make them unwilling to identify known or perceived deficiencies unless directly asked;
(4) audits are often carried out too late in the project life-cycle, resulting in high cost or schedule penalties if there are significant findings that need to be addressed.

Assessments are used to ensure product quality in addition to process compliance, and greater emphasis should be placed on assessments and less on audits. The assessment process tends to be much more effective in identifying and correcting problems for the following reasons:

(1) The assessment team is generally perceived to be there to help, focusing as much on identifying root causes of non-compliance as on ensuring compliance with prescribed methods and processes. This engenders co-operation from project personnel and contributes to the effectiveness of the assessment process in contributing to improved methods.

(2) Assessments are commonly initiated by project management early in the project life-cycle to ensure that changes and recommendations can be implemented with minimal impact on costs and the schedule.

(3) The assessment process is carried out with more sustained interaction than an audit, promoting a positive working relationship.

4.2.4. Integration of software and system

In past projects, issues of software safety management involving the combination of expertise from many topic areas have not been carefully considered, nor has allowance been made for the additional effort required for this process, resulting in serious communication problems and potentially increased risk.

In a non-software system implementation, the variety of associated tasks is small, the restrictions on the designers are limited and the dependences between individuals are simple and relatively easy to manage. Software based systems are used because of the need to implement a more complex function than is feasible with other technologies. This complexity drives the need for a top down organization of tasks and a clear and comprehensive assignment of responsibility.

Management must be prepared to devote a great deal of effort to organizing and resourcing a software project both prior to and during its implementation. Paradoxically, they must be prepared to control the scope of functionality while working with a system whose prime strength is the ability to be flexible and to be able to respond to sudden changes in requirements. Reference [4.8] contains one attempt at defining a structure for management of software safety.

4.2.5. Provision of tool support

The development facility provided by management is required for aiding software development. It usually comprises computer aided software engineering
(CASE) tools, a configuration management system and any other systems that can be used to facilitate the development of the target system. Its essential role in quality assurance is to provide automated checking in order to prevent errors by facilitating otherwise tedious and error prone manual operations.

A testing facility is required to assist in detecting errors. Components of this facility may include test data/scenario generators, software and hardware monitors, input/output emulators and environment simulators and a logging facility.

It is beneficial if a configuration management facility is in place from the inception of the project to enable successful identification, controlling and tracking of all items produced in the project and to allow maintenance to be carried out efficiently. Configuration management systems have proven their worth in being able accurately to ascertain the true composition of any software version and able to rebuild old versions and trace the change history, including the rationale for all changes.

Managements have occasionally been reluctant to include the cost of adequate and effective development, testing and maintenance facilities in the total estimate of a computer project lest the additional amount dissuade the decision making authority from approving the project. Even if funding is initially included, some is likely to be diverted at the first sign of budget problems to achieving the functional requirements, which in practice leads to inferior maintenance products, higher lifetime costs and dissatisfaction of end users.

4.3. CURRENT PRACTICES

4.3.1. Safety policy and climate

Safety policy must be established and maintained as a basis for procedures to ensure the implementation of correct practices. Clear lines of communication and responsibility should be established and maintained. The safety policy should be compatible with the policy of the regulatory body prior to software development.

4.3.2. Strategies and planning

Consideration for safety must start at the system design level, and software should be used where it can complement the safety features built into the system architecture. For systems important for safety, the part of the software which is safety critical should be kept as small and simple as possible, and should be isolated, to the extent feasible, from other software which is not important for safety.

These safety needs, together with economic considerations, dictate the need to create high level standards for different categories of software, according to the safety criticality of the software within its application. Such high level standards should contain measurable criteria for every aspect of the process and product.
A specific life-cycle model of software development should be adopted as a focus for planning a project and for improving software production; data must be gathered on errors found throughout the development process so that corrective action can be effectively taken to improve the process or performance of personnel.

Standards that contain measurable criteria for process and product should be used and/or written. A life-cycle model should be used to focus planning and to allow data on errors to be gathered and the information to be fed back into the development process. Software quality assurance needs to be applied rigorously with clearly defined objectives.

Formal methods are increasingly being adopted and are now also being used in the area of both software and hardware design. The main problem is that formal methods cannot easily be applied to systems with a fair degree of complexity. To be cost effective the design must be constrained to be as simple and limited in scope as possible (see Section 6).

The use of information hiding software partitioning principles is intertwined with the use of the other methods because it can aid in making those processes (such as formal verification) easier to perform (see Section 3).

4.3.3. Personnel and supervision

Personnel selected to work on projects for safety critical software need to be competent for the work and the personnel selection process should be more rigorous than in present practice. This is consistent with the increased attention being paid in various quarters to the general issue of accreditation for software engineers [4.9].

It has been observed that, even amongst qualified software producing organizations, training and education are the areas most needing improvement [4.6]. Specialized training programmes need to be established to train personnel for their particular responsibilities (see Section 14).

A quality assurance administrator reporting to the general Project Manager should provide continuity in monitoring and advising on the compliance with the quality assurance programme over the project lifetime.

Assessments should be performed at regular intervals over the product development life-cycle, and more frequently during the early phases, covering representative samples of all software processes and products.

4.3.4. Integration of software and system

The software engineering process must be integrated within the context of the system engineering process. Very few standards tackle system issues. This is one of the strengths of the IEC 880 standard [4.7] and also the focus of its new extension work.

Preparation of responsibility matrices in detail is a method used to describe safety responsibilities clearly and to distinguish them from design and quality
assurance responsibilities, and to identify clearly the extent and nature of everyone’s responsibility for each of these aspects. Safety considerations must include generic safety impact as well as the impact on nuclear safety.

Quality assurance activities should track progress against all quality plans and ensure that non-conformances and issues are resolved effectively and expeditiously.

4.3.5. Provision of tool support

To achieve the quality required for software important to safety, it is the management’s responsibility:

1. To ensure that adequate resources are available for implementing and maintaining the necessary facilities and tailoring them to the particular software production and verification processes required. Investment in tools should be balanced by the expected payback.

2. To recognize that these facilities are integral parts of the quality assurance programme and thus must be planned for over the entire life-cycle of the software system.

3. To ensure that the facilities are sufficiently comprehensive as to be suitable for the intended usage during all the life-cycle phases. This means the required fitness for purpose of a tool needs to be assessed in conjunction with the importance of its role in the overall software production process.

4. To ensure that if necessary diverse tools are used to provide confidence in the resultant product; for example, compilers for code verification.

4.3.6. Managing the licensing process

The most straightforward route to achieving a manageable licensing process is for the licensee and regulator to reach agreement on standards and methods at the outset of a project, and for there to be no substantial escalation of licensing requirements during the project.

Owing to the rapid advances being made in computer engineering, this may be difficult to achieve. The regulatory agencies involved quite correctly expect the best available practice to be applied, consistent with the successful results achieved by other responsible design organizations throughout the world. However, introduction of an insufficiently proven technique, or the imposition of a major process change with only marginal incremental benefits can reduce instead of enhancing safety.

Therefore, it is considered vital that both the licensee and the regulator work towards agreement defining sufficiency, otherwise the financial risk to the licensee will prevent further deployment of software in safety critical systems. The agreement reached should cover both processes and requirements on documentation, so that the
regulator can be provided with the information necessary to perform an assessment of the software.

### 4.4. SUMMARY AND FOLLOW-UP

Software quality assurance in situations of inadequate resources, inadequate task definition, or inadequate management support is generally ineffective in performing its intended role. For low maturity organizations, technical issues are rarely the priority issues. This is not because they are unimportant, but because critical management problems need priority attention [4.6].

#### 4.4.1. Safety policy and climate

Safety policy and its practical and effective means of implementation must be established at the outset and communicated to all personnel.

Work needs to be done to document the heuristics that enable the overall safety framework to be set up effectively and extended in a practical manner into the necessary areas of software implementation. Management analysis techniques such as 'benchmarking' may be useful in providing a systematic means for carrying this out.

#### 4.4.2. Personnel and supervision

For safety critical software projects, personnel must be competent. Specialized education and training must be conducted for personnel performing specialized tasks.

For software important to safety in NPPs, it is critical that the team members working on the application have both a strong knowledge of software engineering and a full and detailed understanding of the process in which the software is being applied. More work is needed to determine how such teams should be organized and how they should communicate for maximum efficiency and effectiveness.

More work is necessary in defining the best means of documenting designs to facilitate reviewability for the purpose of detecting safety defects [4.10]. Further work also needs to be done to identify detailed procedures and the best format to capture design decisions. The additional degree of formality with respect to capturing rationales for design decisions does increase effort, but should, over the lifetime of a system, reduce the risk due to personnel turnover.

Improved processes for performing design reviews and design walkthroughs that are more efficient in detecting errors are needed.

#### 4.4.3. Integration of software and system

Many software project managers need:

- to improve their system for controlling the myriad of software system processes;
— to receive guidance on what indicators to examine when conducting project
reviews;
— to undergo training on methods and procedures for estimating software size
and resource needs and on the production of schedules.

Management must become so confident in their procedures and the capabilities
of their personnel that they are convinced that assiduous adherence to the procedures
will generate a high probability of delivering a quality product. Such a level of confi­
dence would also discourage management from bypassing procedures to meet other
requirements (such as cost or schedule requirements).

There is no widely accepted process for the identification, trade-off and
management of system functional/software requirements. Such a process and clear
procedures for how to use it would be highly desirable, since the requirements defini­
tion phase is the most important phase of the software life-cycle in terms of its effect
on quality of the final product. This is because missed/inappropriate requirements
identified/corrected at a later stage usually require significant redesign and rework
at substantial cost.

4.4.4. Provision of tool support

Data need to be collected on the financial and safety payback which accrues
from the use of suitable development, test and maintenance facilities over the lifetime
of a software based system. Although this payback is intuitively felt to be substantial,
access to hard data would facilitate justifying the expense at the outset of a project.

Work needs to be done to develop efficient strategies for qualifying tools
for use.

Approaches need to be found which will ensure that the substantial invest­
ments in providing a comprehensive tool suite do not prevent better methods
from being used as they are proven to be superior to components of the existing
tool set.

4.4.5. Managing the licensing process

Licensing of software for use in systems important to safety has in some
instances been difficult for both the regulator and the licensee [4.4]. Waiting until
the software is complete and then trying to evaluate its fitness for purpose means that
the regulator risks a false positive result and the licensee risks a false negative result.
For licensing assessment after completion of the software, the effort necessary and
the decision's major implications may preclude an objective and effective licensing
process.
Reference [4.11] identifies a more rational approach for licensing assessment. It is based on a 'tripod' approach consisting of:

1. testing (both deterministic and statistically valid);
2. systematic inspection;
3. confirmation of people and process.

None of these processes would be evaluated at one point in time at the completion of a software project. Rather, the evaluation would be distributed over the duration of the project. This would enable both

1. the detection and correction of deficiencies earlier in the project when corrective action is less disruptive; and
2. the gradual accumulation of confidence in the software product over time.

It is expected that as confidence is gained in the people and processes over several software implementations, the intensity of the systematic inspection would be able to be reduced from a full detailed verification by the regulator of the product against its requirements to an audit of the verification activities carried out by the licensee.

REFERENCES


5. MODELLING FOR REQUIREMENT ANALYSIS AND DESIGN

5.1. ISSUES

Models are used to help us understand the entity being modelled; to predict its behaviour; and to communicate information about it. Software development is no exception. Models aid us in both analysis and communication concerning the software application. This section deals with the use of mathematical and graphical models in the software development life-cycle.

For the purpose of this discussion, we will assume that the software development process for software important to safety in NPPs has, in part, the following distinct stages:

1. nuclear safety requirements;
2. protective system requirements;
3. protective system design;
4. computer system requirements;
5. computer system design;
6. software requirements;
7. software design.

Each stage consists of an analysis phase followed by a specification phase. The specification is the objective of each of these stages.

It is quite possible to develop these specifications without the explicit use of models (as has often been the case in the past). In such situations we are reduced to relying on experience, or on our ability to visualize a 'mental model' of the system which we use to try to foresee how the system should behave in all anticipated input conditions. This serendipitous design process is not acceptable for systems important to safety. It is not even acceptable for commercial systems where the emphasis is squarely on cost effectiveness. Models have the same benefits in software development as they do in any other engineering endeavour. They enable us to argue with more precision about the system behaviour, design and construction before the system is built. Thus we are able to discover errors, easily overlooked situations and other hazards without incurring the dangers or costs which would result if the system were constructed in ignorance of this knowledge.

Section 5.1.1 introduces a basic model of system design. Section 5.1.2 discusses the need for communication at each stage and between stages, and how the needs are addressed by models.

At each stage of system and software development, the needs for analysis differ. Section 5.1.3 discusses how models facilitate analysis at each stage.
5.1.1. Stepwise refinement and levels of abstraction in system engineering

A basic technique in engineering is to subdivide a problem that is too large to handle into smaller problems of manageable size. If done properly (in particular this means paying careful attention to the interfaces between the subproblems and also their common aspects) then the solutions to the subproblems can be integrated together to provide a solution for the original large problem.

This process of stepwise refinement can be represented as an iterative process of:

1. specification of the requirements for the system;
2. specification of the design of the system: the process of design consists of partitioning the system into smaller subsystems, defining the interfaces between subsystems and allocating the system requirements to the subsystems;
3. specification of the design of each subsystem.

If any of the subsystems are too complex for their design to be fully specified, then the specification in (3) is treated as the requirements for the system, as in (1), and steps (2) and (3) are repeated for the subsystem. This process is repeated until the complexity of each subsystem specification is manageable.

An integral part of this stepwise refinement process is that the problem is viewed at a finer level of detail at each layer of subdivision, while still within the context of an integrated whole. The creation of these views of the system using varying levels of abstraction is an important tool for dealing with complexity. The purpose of the different views is to facilitate the understanding and analysis of the system at the different levels of detail. An important function of abstraction is that it allows distinction between things that are fundamental requirements and those that are introduced owing to design decisions made during the design process.

At higher levels of abstraction, continuous time based models of system behaviour are often most appropriate. At lower levels, a discrete time based model better reflects the sequential nature of computers.

5.1.2. Communication

Each of the specifications that are the output of the stages of system development must cover functionality, performance and interfaces.

Each specification must include a description of functional and performance requirements to be met by the next level of specification or design. These requirements form a model of the system being specified. The required behaviour of the system is represented by the behaviour of the model. Any design that produces a system with the same behaviour as the model, within specified tolerances, is deemed to conform to the specification. The tolerances represent the performance requirements.
If the model used has a well defined, formal foundation and is expressed using a notation with well defined syntax and semantics, then the possibility of misinterpretation of the specification is greatly reduced. The formal foundation aids the specifier in producing a specification that is consistent since there are rules against which the specification may be checked for compliance. The specification should be checked to determine if the complete input domain has been covered and this is one test for completeness. Some notations, such as tabular representations of functions, facilitate this checking even as the specification is being produced. This point is further discussed in Section 6.

In order for a model to be effective in facilitating communication, it should:

- provide a means of defining the system boundary;
- provide a means of defining partitions and abstractions;
- assist the analyst to think about and document the problem as well as the solution;
- allow for opposing alternatives but alert the analyst to their existence;
- make it easy to modify the knowledge base;
- be based on a formalism understood by the intended audiences (this is important since (a) the model must be validated by the specifier and must be understood by the users; and (b) models that are understandable by a system designer are not always understandable by a software designer, and vice versa).

Examples of formalisms on which models used for specifications are based are [5.1]:

- finite state machines
- stimulus–response paths
- data flow models
- communicating concurrent processes
- functional composition
- data oriented models.

Finite state machines and stimulus–response path models are most relevant in applications important to NPPs.

5.1.3. Analysis

There are many different types of analyses that must be performed during the software development life-cycle (for instance, analyses for conformance, completeness, consistency, performances and timing). These sometimes differ in their significance depending on the stage of the process.

It is important for these analyses to ensure that the models used capture all the information relevant to the analysis, and exclude most and possibly all the rest.
This section discusses the stages at which model based specifications may be applied, the salient attributes that need to be modelled, the types of analyses that can be performed on models and desirable attributes of a model.

(Models are also used for analysis during testing; but these models are discussed in Section 9.)

5.1.3.1. Specification of nuclear safety requirements

Design requirements for nuclear safety specify protective system characteristics that are necessary to meet nuclear safety requirements for the facility. The nuclear safety requirements are, in part, imposed by regulatory bodies, and they generally specify the acceptable amount and frequency of radioactive releases from the nuclear facility. Models of the systems under consideration are employed to determine the design requirements for nuclear safety. These models are used to simulate the effects of accidents, and to predict the amount of radiation exposure that could potentially result from the postulated failures. They simulate the behaviour of the process in which the failure occurs, the protective system that mitigates the effects of the failure, and the mechanisms that govern the exposure of the environment and population to radiation that may be released as a result of the failure.

The design requirements for nuclear safety for a protective system are based upon the results of model calculations that are used for accident simulations. They are generated by users of the simulation models, and therefore they are often expressed in terms of the model and its characteristics. It is necessary, therefore, that there is a well understood relationship between the models used for deriving the design requirements for nuclear safety, and the final design that incorporates the requirements. Ideally the model used for simulation would be the same as the model used for specification of the design requirements for nuclear safety.

Salient characteristics to be modelled:

— model the portion of the plant that determines whether the protective system is adequately performing its function; this plant model includes the plant parameters that are to be maintained within certain limits;
— model the protective system with respect to its essential safety function, that is in terms of its effect on the plant; this implies a simulation of the interactions between the plant and the protective system.

Desirable analysis:

— predict plant performance during upset conditions given a protective system with certain specifications; ensure that important plant parameters are maintained within required operating limits;
— assess the accuracy of simulations of plant and protective systems.
— analyse timing characteristics of models since the performance requirements are usually the most essential part of the protective system specification;
— assess the impact of the protective system on production: are the specifications for the protective system such that an unacceptable number of spurious trips will not occur?

Choice of model and notation:
— the model should facilitate cross-referencing between simulations and the specification of requirements for nuclear safety; executable specifications are thus preferred;
— the model should be structured so that completeness of specifications can be assessed;
— the model should assume a continuous time domain to simplify representation and should facilitate specification of performance requirements;
— no particular implementation for digital technology at this point should be assumed.

5.1.3.2. Specification of protective system requirements

The specification of system requirements completes the specification of the protective system by adding to the nuclear safety requirements those requirements not directly associated with safety, such as requirements for maintainability and operability.

Salient characteristics to be modelled:
— abstract representation of interfaces between the protective system and the plant;
— interfaces to operator and maintainer could be explicitly represented at this level, or the operator and maintainer could be considered to be part of the system; in this latter case, their interfaces to the plant and any other external systems should be represented as well.

Desirable analysis:
— conformance to specification of nuclear safety requirements;
— completeness of specifications: they must cover the protective system behaviour for its complete input domain, that is, for all combinations of inputs, and not just for those characteristics necessary for meeting the nuclear safety requirements.

5.1.3.3. Specification of protective system design

The specification of system design will partition the system into subsystems, define the interfaces between the subsystems, and then allocate to these subsystems the system requirements as they have been specified.
Salient characteristics to be modelled:

— the partitioning of the system into subsystems; that is, the definition of the functionality of each subsystem and of the interfaces between subsystems;
— the specification of the time and performance characteristics of the subsystems.

Desirable analysis:

— completeness of specifications of interfaces between subsystems;
— compliance of subsystems and of their interfaces with the required functional behaviour and with the performance requirements, within the constraints of the specification of system requirements;
— hazard analysis, complexity assessment and reliability analysis to demonstrate that the system configuration, making use of subsystems or components of the specified reliability, will achieve the system reliability requirements (this analysis will include assumptions on the contribution of diversity; see Section 3.3.3);
— feasibility analysis to verify consistency with technological constraints.

Choice of models and notations:

— still in the continuous time domain, but may be different for different subsystems, depending on technology choices made for each subsystem.

5.1.3.4. Specification of computer system requirements

The specifications of computer system requirements refine the allocation of functionality assigned to the computer subsystem. The specifications of computer system requirements must be a complete representation of the required behaviour of the computer system and should be at a sufficiently detailed level of abstraction to define precisely the interfaces between subsystems.

Salient characteristics to be modelled:

— detailed interface specifications (refinement of the abstract interface specification made in the specification of system design);
— complete specification of functional and performance behaviour, including the behaviour of input/output devices (for example, sample rates).

Desirable analysis:

— verify that the failure modes and error conditions and any other abnormal conditions of the computer and the devices are accounted for in the requirements;
— check that the discrete time domain model adequately reflects the requirements specified in the continuous time domain model of the higher level specifications.
Choice of model and notation:

- a discrete time model is more closely related to computer discrete sequential behaviour;
- notation should be such that it can be formally verified against the software design.

5.1.3.5. Specification of computer system design

The specification of the computer system design must partition the required functionality specified in the specification of computer system requirements between hardware, software and predeveloped software. This includes the specification of interfaces between hardware and software, and the allocation of performance requirements between hardware and software.

The feasibility of this partitioning must be verified.

5.1.3.6. Specification of software requirements

A necessary prerequisite to providing assurance that the software will serve its intended purpose is to specify precisely the software requirements. These requirements are typically documented in a software requirements specification (SRS).

The SRS is a central document for any engineered software product but is of crucial importance for critical applications: it is the essential basis of the design of the software, and of the validation of the final product. Adequate assurance that a software product has met its requirements cannot be attained without a complete, precise and correct specification of the required behaviour of the software based system.

The functional behaviour and the performance requirements of the software are the salient characteristics to be modelled. Time domain should be discrete. The example given in Section 5.3.1.8 illustrates these specifications.

5.1.3.7. Specification of software design

A rigorous specification of software requirements makes easier the verification of the specification of software design (description), assuming that the description of software design is documented using an appropriately rigorous method (see Section 6). Such a verification provides greater confidence that the design implements the correct solution to the problem. It also allows design errors to be detected earlier in the software life-cycle.

Besides, the verification of the description of software design can provide an audit trail which is useful in demonstrating the correctness and the completeness of the ultimate system to auditing and licensing bodies.
Salient characteristics to be modelled:

— the partitioning of the software system into modules, programs and data structures;
— the relationship and interfaces between these components.

Desirable analysis:

— complexity;
— modifiability;
— conformance with specification of requirements.

Modelling:

— choice of model and notation must correspond closely to the actual implementation of the software;
— modelling entities must have good correspondence with entities in the implementation (processes, tasks, procedures).

In the early stages of system or software design, when the architecture of the system or program is intuitively being ‘fashioned’, it can be very helpful to visualize its constituent parts and the relationships between them in graphical terms. The entities considered here may, for example, be processes and data objects connected by data flows, or they may be subprograms arranged in a call tree. The design methods considered here typically allow such structures to be ‘enriched’ by labelling their elements (for example, labelling arrows representing data flows with the names of the data objects concerned), and ‘refined’, for example by decomposing a component such as a task into a combination of subtasks. ‘Consistency rules’ applicable to such structures can often be checked with supporting software tools.

Structured design methods such as the Yourdon method [5.2] have been widely employed for the design of control system software for many years, and software tools supporting their use are popular. (For a review of such methods see for instance Ref. [5.3]. The most recent method in this category, object oriented design, is described in Ref. [5.4].)

5.1.3.8. An example of a model of system, computer and software requirements

Figure 1 shows a typical computer control system. Each input $m_i$ to the system is represented mathematically as the element $M_i(t)$ of a time function vector and is referred to as a monitored variable. Similarly each output $C_j$ from the system is represented as a time function $C_j(t)$ and is referred to as a controlled variable. To specify the required behaviour of the system, a function, REQ, must be specified. The REQ function maps $M_i(t)$ to $C_j(t)$: thus $C_j(t) = REQ(M_i(t))$. REQ is a model of system requirements.

In order to specify the software requirements, the SRS must define which of the system requirements will be implemented by the hardware. This is done by first
FIG. 1. Schematic diagram of a typical computer control system.
defining the interfaces between the hardware and the software. Data read by the software are represented by a time function vector element \( I_i(t) \), and data written by the software are represented by \( O_j(t) \). To specify the functions that are to be implemented in the hardware or in predeveloped software, the SRS must define a set of relations \( \text{IN} \) that map \( M_i(t) \) to \( I_i(t) \) and a set of relations \( \text{OUT} \) that map \( O_j(t) \) to \( C_j(t) \).

The specification of \( \text{REQ} \), \( \text{IN} \) and \( \text{OUT} \), constrains the design of the software to any design that will produce the required behaviour of \( C \) in terms of \( M \) given a hardware environment that maps \( M \) to \( I \) and \( O \) to \( C \) as specified by \( \text{IN} \) and \( \text{OUT} \). In many cases this degree of constraint on the software design is sufficient. In other cases, more constraints may need to be specified in the SRS with respect to the manner in which the hardware is to be used, for example a requirement that no interrupts be used.

The specification of the \( \text{IN} \) and \( \text{OUT} \) relations may be accomplished by referencing the hardware manuals or hardware design documentation. The specification of \( \text{REQ} \) must be produced by the requirements specifier. The finite state machine model described in Section 5.2 can be used to produce the \( \text{REQ} \) function specification. This approach to specifying requirements was based on work in Ref. [5.5]. An application of this approach to an NPP protection system is presented in Ref. [5.6].

5.2. EXPERIENCE

To date, modelling has been used in the areas of specification of nuclear safety requirements, in system design specification, in specification of software requirements and in specification of software design.

Simulation modelling of the plant and of the protective system has been used to derive the nuclear safety requirements but has not been used as a communication/specification vehicle to document them.

Modelling at the level of system design specification has been used to assess performance and reliability of protective systems.

Specifications of software requirements have recently used modelling as a means of producing precise and rather complete specifications. Specifications of requirements based on modelling the required behaviour of the system outputs as a function of the system inputs provides a technique for producing a rigorous, comprehensive SRS. A technique for modelling the required behaviour of a software system can be based on finite state machines and on the tabular representation of the mathematical functions that define the states of the system and the outputs from the system.

5.2.1. Finite state machine model

The behaviour of a finite state machine is described at discrete instants of time designated \( z = 0, 1, 2, 3, \ldots \), where the time between each event is some arbitrarily
small amount of time $\tau$. Suppose that a system has been receiving inputs and has been responding by producing output signals. If now the $z$th input, $M(z)$, is applied, the response $C(z)$ would depend on $M(z)$, as well as past values of inputs $M(0)\ldots M(z-1)$. Sequences of past values of inputs that cause the same response of the system for a given $M(z)$ can be grouped together into a class. For a finite state machine, the number of classes of input histories that can affect future behaviour must be finite. These classes of input histories are referred to as the states of the machine. The value of the current state of the machine is retained in what are referred to as state variables.

The finite state machine model is defined as follows:

1. $S(z + 1) = \delta\{S(z), M(z)\}$
2. $C(z) = \lambda\{S(z), M(z)\}$

where $S(z)$ is the state of the system at time $t_0 + z*\tau$;
$t_0$ is the time of initialization of the system;
$\delta$ is the next state function;
$\lambda$ is the output function.

To specify the behaviour of a system, for instance of the REQ function introduced in Section 5.1, the state variables necessary to represent the states of the system being specified must be identified, the state transition function for each state variable must be specified, and then the output function for each output from the system must be defined.

The state transition function and the output function can be defined using a tabular representation as described in Section 5.2.2.

5.2.2. Tabular representation of functions

A function maps a number of inputs to a unique output value. In specifying the state transition functions and the output functions, it is important to specify the required output value for all values of inputs to the functions and for all combinations of inputs to the functions, in order for the specification to be complete. Tabular representation provides a means of defining functions that facilitate the process of checking the function for completeness and consistency.

A specification technique has been developed on the basis of experience gained in applying a similar technique to the specification of requirements for computer based safety systems in Ontario Hydro nuclear generating stations [5.7]. That work was based on the method of specification of requirements developed at the Naval Research Laboratory for the A-7 aircraft [5.5]. As well as providing a means for achieving the required attributes, the technique facilitates systematic verification.
techniques that provide auditable evidence that the software design and code meet the requirements in the SRS. Periodic, deterministic programs can be represented as a finite state machine where the program is a function that acts on memory variables that maintain the state of the computer. Once programs are represented in this manner they can be rigorously compared with an SRS that is based on a finite state machine model. This systematic verification is a key mechanism for achieving the required degree of quality assurance for safety critical software in NPPs where a regulatory agency must be convinced of the adequacy of the degree of assurance of quality.

5.2.3. Structured design methods

Structured design methods, strongly based on graphic models, have been used in documenting specifications for software design. They are now embodied in many CASE tools [5.2–5.4]. The proper use of these methods and tools is encouraged, but they must not be misused. Often a structured design method provides an excellent representation of a particular aspect of program behaviour, but its portrayal of other features may be poorly defined or even impossible. The best choice of design method therefore depends strongly on the nature of the programming task and of the solution sought — for instance, whether the program is to be viewed primarily in terms of function applications, or its data structures, or interactions of abstract state machines (objects). Indeed, no single structured design method is likely to be appropriate to all aspects of a project; as Ref. [5.8] explains, it is for good reason that there is a general resistance among software designers to any ‘Method with a capital M’. Certainly, since a structured design method cannot entirely capture a program’s semantics, its use in refining a design should cease at the level where the program code is equally comprehensible. Thus if a structured design method is to be used at all in developing a program to be implemented in an expressive high level language such as Ada or Modula 2, it should be employed for the higher level architectural design, but not in the detailed implementation of relatively low level subprograms.

Finally, if use of a program design language (PDL), or pseudo-code [5.3], is contemplated, one should ensure that it has well defined syntax and semantics. If a program is to be implemented in a modular high level language, with sufficient expressive power to describe the program design, then the implementation language, rather than an imprecise derivative of it, should be used at the earliest opportunity.

5.2.4. Formal methods: a practical assessment

The term ‘formal methods’ is taken here to mean the use of mathematics in software development. Not surprisingly, as in all fields of engineering, some aspects of the subject are more amenable to mathematical treatment than others.
'Formal methods' have been used in the following kinds of activities in the development of 'real' systems (it is to be noted that in any particular project, they have usually been employed for some but not all these activities).

1. The expression of an informal requirement specification in a mathematical form (the mathematical formulation is described here as a *formal specification*).
2. The proof that a formal specification has certain required properties.
3. The expression of a (modular) design in a mathematical form (described as a *formal design*).
4. Proof by mathematical argument that the code of a program conforms to a formal design (described as *formal verification* of the code).
5. The rigorous 'refinement' of a formal design to code.

The practicality of each of these processes are now considered in turn.

**Formal specification**

From the earlier discussion of specification of requirements, it follows that insofar as formal methods are applicable, one would expect formalization of the specification of requirements to be the most cost effective. For many projects, this is the only part of the development that has been performed formally. Much of the mathematics for the specification language Z [5.9], for instance, is taught in high school, and the skill required is not very great (much less than for constructing proofs). The process has been found practicable for large systems: International Business Machines (IBM) is using Z to respecify key interfaces of CICS to improve its maintainability, and so far Z specifications have been produced for more than 100 000 lines of new or changed code [5.10, 5.11]. Where formal specifications have been produced, they have been found much easier to inspect than informal ones. The formalization process itself highlights incompleteness and inconsistencies. (The specification in VDM language of an International Organization for Standardization (ISO) standard for message authentication first revealed a serious ambiguity in the standard [5.12].) Examples of formal specification of hardware and systems are given in Refs [5.13] and [5.14].

**Proof of properties of specifications**

These properties may relate to consistency of a specification or completeness of operation definitions, or proof that the specification meets certain requirements. (See for instance Ref. [5.15], which proves properties of the specification of a storage manager.) It must be said, however, that proof generally requires considerable expertise. Tools now exist to support this work (see Section 12.3.2), but these still require skilful interaction.
It is important to bear in mind the fact that a formal specification is based on a conceptual model of the real world. Engineers are well used to employing models, and know that they have their limitations because they are simplified representations. It may be possible to prove that a formal specification is logically coherent in various ways and that it has certain properties considered important; but since the real world is not a formal system, one could never prove that a formal specification is 'correct'.

Formal design

Whereas a specification of a software requirement states its purpose (what the software is required to do), its design and code state how that requirement is to be met. As in other kinds of engineering design, the first steps in creating a software design will rely heavily on the experience and skill of the designer, who must use intuition in the initial conception of the software 'architecture' (or high level modular structure). As the initial outline form is gradually established, the good designer produces detailed commentary (outlining the nature of processes and their data transactions for instance), to give the initial structure 'semantic substance'. This is another area where the use of mathematical representations can provide clarity and precision. A high level structural representation of a program, with mathematical specifications of its components, is described as a formal design. As in a specification of a formal requirement, the mathematics of the formal design description should be embedded in a natural language commentary, possibly supported by diagrammatic representations.

How can it be verified that a program, as described by its formal design, meets its requirements as described in their formal specification? Sometimes, functions and actions introduced in the formal requirements specification have direct counterparts in the formal design. For instance, many of the required functions and operations in the specification of a database system will reappear in essentially the same form, as specifications of function and procedure subprograms that implement them, in the formal design. In other situations, the connection between the specification and the design is more tenuous, and although Ref. [5.16] mandates the proof that the formal design meets the requirements, this can be very difficult.

It is also important to bear in mind the fact that just as a formal specification is based on a model of the real world, a formal design — and its subsequent use in constructing a program — is based on a mathematical model of program behaviour. As is usual in modelling, only some aspects of program behaviour are captured in this way: there are useful mathematical models of sequential program behaviour, but the models of concurrent behaviour are less easy to employ. Again, the correspondence between formal program description and real system behaviour is limited, since the models do not take into account the effect of compilation, or the behaviour of operating systems or underlying hardware. Thus, if software requirements include
timing constraints, these can be formally specified, but it is not known how to use this specification to produce software that meets the constraints.

Formal verification

For a program or subprogram of very modest size, with a formal specification given in the form of a precondition (stating the conditions that are assumed to apply when the subprogram is called) and a postcondition (stating the intended conditions when execution of the subprogram terminates), it is possible to prove whether or not the code is consistent with its specification. First, the specification and code can be mechanically combined to produce their verification conditions (logical formulae, expressed in a predicate calculus). If these formulae can be proved to be true, then the code conforms to its specification. Software tools are available to support this work, in the form of verification condition generators, and interactive 'proof assistants' or theorem provers (see Section 12.3.2). Formal verification of this kind is performed on an industrial basis (see for instance Ref. [5.17] describing formal verification of the assembly code modules used in the fuel control unit of a jet engine).

It will be appreciated that whether code is to be mechanically verified or not, the preconditions and postconditions of subprograms are indispensable parts of their documentation (see Section 6.3.2). As for the feasibility of applying these techniques, the generation of verification conditions is entirely mechanical, but construction of their proofs usually requires interaction by a skilled user. This can be quite arduous, because it is a difficult intellectual task and because of the limited power of current proof tools. The difficulty of constructing proofs depends very much on the quality of the code: the programmer's familiarity with the ideas of rigorous program development, as described in Ref. [5.18] for instance, leads to a better product, and greatly simplifies its formal verification. Effective configuration management is essential, as is the rigorous control of access (especially insertion of proof rules) to the database.

The use of a formal method for code construction or verification always requires a formal definition of the programming language employed. Very few adequate language definitions exist, though such have been constructed for Modula 2 and Pascal, for appropriate subsets of Ada and for a number of assembly codes.

Refinement of formal designs to code

The application of the formal verification method just described to an entire program, of appreciable size, is not practically feasible. Instead, starting from the formal design, a correct program may be constructed in a series of refinement steps, each small enough for it to be seen what has to be proved, to show that the step is correct. If the correctness is in doubt, the proof can be carried out. The successive
steps involve, for instance, choosing concrete data structures to represent abstract structures in the specification, decomposing high level operations into lower level ones and developing algorithms to implement these. This style of program development, using VDM, is described in Ref. [5.19], and Ref. [5.20] describes applications. Another formal program development method, based on stepwise refinement of abstract state machine models of programs, has been developed [5.21]; this has been applied successfully to the development of railway software.

This approach, however, is not without problems. Each refinement step must be very simple, if it is to be possible to formulate and discharge the associated proof obligations. Not surprisingly, in a practical software development, the number of such steps and the amount of proof work can be enormous.

5.3. CURRENT PRACTICES

Although many methods and models exist for the specification and analysis of software, no one is widely or specifically used for software important to safety in NPPs. Sections 5.1.3.1 to 5.1.3.7 recommend desirable types of analysis that may be performed, and recommend models and notations to use at different stages of development of a protective system. It is more important that the technique used have the following attributes than that any specific technique be used:

(1) It should be capable of handling concepts of partitioning and abstraction;
(2) It should be understandable by the intended audience;
(3) It should clearly define boundaries of the system being modelled;
(4) It should be easy to modify to facilitate changes or refinements during iteration;
(5) It should facilitate checks for completeness and internal consistency;
(6) It should have formally defined semantics for the model and formally defined syntax for the notation.

There are many modelling and specification techniques available in the literature. The degree of usage of any particular technique is difficult to establish. Many of the techniques have attributes (1) through (4). Experience has shown that formal techniques based on mathematical models are necessary to provide the attributes of formally defined semantics and syntax. Tabular representation of mathematical functions has facilitated completeness and consistency checks.

As for the use of formal methods, many extravagant claims are made both for and against. Some of their advocates claim that they can fundamentally improve software development; their opponents say they are too difficult to apply. Most software engineers find it difficult to judge the competing claims, because formal methods are unfamiliar and there is still not much accessible evidence to support one side or the other; Ref. [5.22] may help to dispel some of the myths.
5.4. SUMMARY AND FOLLOW-UP

Application of models within a context of stepwise refinement methods and levels of abstraction provides a means of:

— making communication among designers and programmers less ambiguous and more precise; and
— promoting progress, completeness and accuracy in each analytical stage, so that errors are discovered before their effects can propagate.

The as yet incompletely solved problems associated with the various specification stages listed in Section 5.1 are as follows:

— The degree of rigour of the specification of the nuclear safety requirements has differed in the past and typically the assurance that the protective system is sufficient to meet the safety needs is accomplished by refinements to the safety analysis once the design of the system has been finalized. At that time more details are known that can be taken into account in the simulation models or safety analysis. Deficiencies found at this time are expensive and often difficult to remedy. A precise and complete statement of the essential characteristics of the protective system would provide a means for communicating between the safety engineer and the design engineer of the protective system, and would provide a means for verification that the subsequent specifications are consistent with the specification of safety requirements.

— There is not a high degree of traceability from the low level design up to the nuclear safety requirements and hence the ability to perform end to end verification that the nuclear safety requirements have been met is diminished.

— Effective review of the correctness of the implementation is difficult when only loosely defined specifications of requirements are available for review, and precise but detailed specifications of the implementation are available.

— When changes are made to the design it is difficult to perform effective review that the changed design is still consistent with the higher level requirements.

— Without the use of stepwise refinement, levels of abstraction and precise specifications, it is difficult to provide a means of documenting the rationale for the design decisions so that they may be maintained and made available to designers making changes in the future.

When software requirements are specified using systematic techniques similar to those previously discussed, the following benefits arise:

— The use of mathematics to describe functional requirements allows requirements to be written unambiguously. This reduces the chance that designers will develop a solution to the wrong problem.
— Formalisms allow for the development of systematic methods for performing completeness and consistency checking of the specification of requirements for software, improving its reliability.

The use of formal methods is compatible with 'conventional' modular program development. Their most important contribution, as in the application of mathematics to other engineering design processes, is in clarifying the problem definition and the justification and documentation of the successive design decisions. Whether particular steps are justified by mechanically supported proof or just by careful reasoning is a delicate matter of choice. Until tools to support proof become significantly more powerful, formal code verification will not be practicable for large systems in their entirety, but for safety critical components of NPPs it should be feasible.

To summarize recommendations on the use of formal methods:

(1) Formal methods should be applied throughout program design and development to the specification of software requirements and to the specification of program units.
(2) For safety critical software in particular, code should be formally verified.
(3) The construction of proofs of properties of specifications and designs, and formal verification of code, is still very laborious. The extent to which it will become feasible for large programs will depend largely on the development of more powerful proof tools.

REFERENCES


6. DOCUMENTATION

6.1. ISSUES

It is widely believed that, at the present time, one of the most effective ways of obtaining more dependable software is to improve on the trustworthiness of the successive design and construction processes by which the software is produced. For if the reliability of a software product cannot be assessed directly, confidence in it must largely be based on evidence of the soundness of the process of constructing it. The underlying idea is that it is easier to review and validate the successive steps of the construction process — which in the case of software is purely intellectual — than to validate the completed end product retrospectively, in all its complexity [6.1, 6.2].

It is clear that documentation plays a crucial role in providing the 'transparency' or 'traceability' required by this approach. Very clear and precise documentation of the requirements, the design and the code are necessary if the designers, the programmers and the independent reviewers are to comprehend fully every stage of the development and verify its completeness and correctness. Good documentation is also essential to maintenance: if requirements and design concepts are not adequately documented, changes made during maintenance may not be consistent with them.

Documentation is therefore an essential, quite major part of the design and implementation of a trustworthy software product. The need for it has long been recognized in other fields of engineering, but to some extent computer programming still lacks such professional engineering traditions — and the discipline these impose. It must also be said that to produce appropriate high quality documentation requires training and skill. Even when it is appreciated that the rationale of a complex software product cannot be conveyed adequately by a few comments scattered in its code, the most appropriate alternatives may not be obvious. Natural language descriptions are often ambiguous. Some CASE tools produce a profusion of diagrams and other documentation, which create an illusion of good engineering but whose bulk obscures important issues.

6.2. EXPERIENCE

6.2.1. Introduction

All relatively recent software standards stress the importance of documentation, give a list of the documents that must be produced for certification purposes and prescribe — with varying degrees of precision — what their contents should be (see Refs [6.3, 6.4]).

63
The classification suggested in Ref. [6.1] is followed here as it closely traces the different stages of development of a software based NPP safety system. As is indicated in Sections 3.2 and 5.1, it is reasonable for the purpose of this report to assume the following typical development stages:

1. Specification of protective system requirements;
2. Design of the protective system;
3. Specification of computer system requirements;
4. Design of the computer system;
5. Specification of software requirements;
6. Software design.

Each stage uses information derived in earlier stages and provides information for subsequent ones. The products of each stage are mainly documents, the principal ones here being documents specifying requirements and documents describing designs. Of course in reality, as the design progresses, errors and omissions are discovered, so that some modifications of products of earlier stages are required. (Indeed, the extent to which documentation retains its usefulness depends largely on the effectiveness of procedures to control and incorporate changes, as successive iterations on a design are performed.)

At a very early stage, as a companion to these system and software specifications, a software quality plan must also be established. This plan and its associated documents specify all the criteria that the products are expected to meet and include standards defining the methods, rules and tools to be used in specification of requirements, design and programming. The plan specifies the quality assurance personnel, their authority and responsibilities and mechanisms to resolve disputes. It defines the quality assurance processes to be employed for checking compliance with the given standards, with plans for verification and validation, inspection and testing.

In what follows attention is focused on the provision of adequate system and software documents, and in particular on requirements specifications and design descriptions. It is clear that rigorous construction of these is essential to the proper construction and implementation of a software quality plan. Section 6.2.2 explores the reasons why the provision of adequate documentation has been so difficult, and Section 6.3 describes software development and documentation practices that have proved helpful in overcoming these problems.

6.2.2. Common deficiencies of software documentation

For many years the term program has conveyed only the notion of source or object code. The documentation of a program has been regarded as a quite separate entity, whose connection to the code could possibly be tenuous. This is a very different approach from normal engineering practice which dictates that plans be put on paper and documents be prepared, analysed and revised before construction is begun.
Of the documents which should be produced, particular attention is drawn to those which specify requirements: of the protective system, then of the computer system, and finally of the software. For it is against a statement of requirements that the adequacy of a design, and the fitness for purpose of its implementation, should ultimately be judged. Yet in practice, requirements are often imprecisely stated, and some of them may be omitted altogether. In an empirical study [6.5] it was found that 30% of all errors could be attributed to faulty statement, or misunderstanding, of requirements and specifications. Furthermore, it seems that errors made in these early stages are among those most likely to lead to catastrophic failures [6.6]. Rectification of such errors late in the software development process is also likely to be particularly difficult and costly [6.5, 6.7], because (1) the changes often have a widespread effect on the system; (2) it may be difficult to identify all the components needing modification; and (3) inadequacies of documentation exacerbate the difficulty of determining the reasons for certain design decisions and their impact.

The documentation of designs has received more attention recently, with the proliferation of CASE tools, but such tools must be chosen to suit the application, and their limitations must be recognized. Too often, a design description consists of hundreds of pages of data flow diagrams or function or structure charts that contain a mass of information but do not enable an engineer to comprehend the intended behaviour of the components of the system, let alone the system as a whole. For although such design representations may be very helpful for describing certain aspects of a program, such as its data flows or control sequences, they do not in themselves have sufficient expressive power to fully capture a program's behaviour. The document modifications involved in a design change can be complex and error prone. It can be observed that programmers wisely avoid using such documentation if they possibly can; instead they refer directly to the program code. They may then cease to maintain the documentation properly (except insofar as this can be rigidly imposed).

6.3. CURRENT PRACTICES

As was said earlier, we focus our attention on system and software documentation for requirement specifications and design descriptions. And, taking the same viewpoint as in [6.1, 6.8–6.10], we are essentially concerned with the contents of these documents. Some brief comments on formats are made at the end of the section.

6.3.1. Structure of software documentation

With the gradual systematization of the program development process, the notion has evolved that the various documents associated with a program (such as
the requirement specification, design description, module specifications, their code bodies, verification conditions and proofs, testing plans and their results) should be developed hand in hand, as a coherent whole, throughout the program construction. Fitness for purpose of a program can then be achieved by ensuring consistency within and between these documents (through their systematic production and inspection) at each step of their development, rather than by ‘searching for bugs’ in the completed code.

An example of this approach is the method of documentation of software requirements that has been followed by Ontario Hydro and Atomic Energy of Canada Ltd (AECL) under the direction of the Atomic Energy Control Board of Canada (AECB) to validate the control software implemented at the Darlington nuclear power station [6.8]. Earlier, less refined versions of the same method had been successfully used by the US Naval Research Laboratory for the documentation of flight control software [6.11]. Initially inspired by intuition and software engineering experience, the method has now been explained by a mathematical model which has progressively been made more explicit [6.9, 6.10]. A brief introduction to this model was given in Section 5.2. The method specifies formally what the contents of the various documents should be, but to a certain extent leaves open the choice of the notations to describe these contents [6.9]. The availability of an explicit model has resulted in improved quality of documentation. An illustration of the method, applied to the requirement documentation of the safety feature actuation system of a PWR reactor, is given in Ref. [6.1].

6.3.2. Overview of document contents

In Ref. [6.9] it is suggested that, associated with the successive stages of a computer system and software development project, there should be the following documents:

<table>
<thead>
<tr>
<th>Design stage</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>System requirements</td>
<td>System requirement document</td>
</tr>
<tr>
<td>Computer system design</td>
<td>System design document</td>
</tr>
<tr>
<td>Software requirements</td>
<td>Software requirement document</td>
</tr>
<tr>
<td>Software behaviour</td>
<td>Software function specification</td>
</tr>
<tr>
<td>Software decomposition</td>
<td>Software module guide</td>
</tr>
<tr>
<td>Module interface design</td>
<td>Module interface specifications</td>
</tr>
<tr>
<td>Module internal design</td>
<td>Module internal design document</td>
</tr>
<tr>
<td>Program design</td>
<td>Program function specifications</td>
</tr>
</tbody>
</table>

The system requirement document treats the computer system (the computers, their peripherals and the interconnection network) as a black box. It gives a description of the environment in terms of environment state variables. It describes the
relations between these state variables that result from physical, natural or other con-
straints. It also specifies the additional relations that will have to be established and
maintained by the computer system when it will operate in this environment.

The notation and concepts used in this document must apply to systems built
of analogue circuits as well as to computer systems. This is important for the
documentation of computer systems which are embedded components of larger sys-
tems, and it reflects the fact that the operational requirements do not necessarily dic-
tate the implementation technology to be used. Besides, the document must be easily
understood by system, hardware and software engineers.

The system design document identifies the computers within the system and
specifies the communications between the computers and the environment by
describing the intended relations between the values of the environmental state varia-
tables and the contents of the computer input/output registers. The existence of two
separate documents for specification of system requirements and description of sys-
tem design is helpful in keeping the specification of the system behaviour completely
distinct from the computer design details.

The system requirement document and the system design document together
largely determine the software requirements, and may together constitute the software requirement document. However, it is usually the case that there are further
requirements of the software behaviour. A further document, called here the software function specification, can record additional design decisions and describe the
software behaviour more completely. As before, all these specification documents
remain implementation independent: just as the functions described by the system
requirement document could, in principle, be implemented by digital or analogue
techniques, so the software function specification describes only the intended
behaviour of the software, and not its partitioning into modules or other implementa-
tion features.

The software of a system is organized as a structure of interdependent compo-
nents called modules. A module is a collection of type and object declarations and
programs, which together constitute for instance an 'abstract data type' (a user
defined type, with operations applicable to it, such as a complex arithmetic or spatial
vector type) or an 'abstract state machine' (which could embody for example a
device driver or other kind of interface to a physical system). Each module may cor-
respond to a work assignment. A user has access to the facilities of a module through
its user interface, but the details of the implementation of these facilities are hidden.
The modules must first be defined by specifying their user interfaces precisely; these
specifications are then employed in writing the code that implements the modules.
The software module guide is a document that describes the decomposition of the
software into modules and defines the role of each of them. Collectively, the modules
must satisfy the software requirements. For each module listed in the guide, there
should be a module interface specification document which, treating the module as
a black box, describes how the programs of the module can be invoked from the outside, and the effects of their execution on the environment.

For each module, there should also be a module internal design document which describes the module data structure, which may include objects implemented by other modules, and the effect of each program of the module on this data structure (clear box description). The Program Function Specification specifies the behaviour of each of the module programs, in terms of mappings from data states before the program execution to data states after the execution. This program function describes precisely the effect of a program without describing the intermediate states. The program function can be used as both a specification of the program and a commentary [6.12].

Whether code is to be mechanically verified or not (see Section 5.2.4), it will be appreciated that preconditions and postconditions of subprograms are parts of the documentation. For it is only by specifying very precisely the conditions under which each subprogram can be called and the action it performs that one can have confidence in applying the ‘abstraction through specification’ needed to reason about a program’s behaviour. This documentation is also obviously essential to the reuse of program components, which of course requires that their essential preconditions still apply. (The code verification described in Ref. [6.13], which was applied to well written and thoroughly tested code, led to clarification of approximately 10% of the originally informal module specifications.) When a proof is constructed, the ‘proof log’ produced by the proving tool, which lists the proof steps it performs and the ‘proof rules’ employed, is also invaluable documentation, being the logical justification of the code.

6.3.3. Document formats

It would not be appropriate here to enter into the details of what the formats of documents should be. However, two useful observations can be made.

A computer system in an NPP interacts with a real environment that is very complex. However, in general, only a small portion of the total information that would be required to describe this environment is directly relevant to the operation of the computer system. It is this relevant information that must be summarized very precisely in the system requirement document. A first, critical step in documenting the system requirements is therefore to identify the state variables that exactly characterize the environment in which the system will have to operate: These variables are those that are monitored by the system, or controlled by it, or both (see Section 5.1.3.8). The system requirements document must describe the constraints imposed by the environment on these variables. It must also describe the relations between these variables that must be imposed by the system. It is therefore possible to define these requirements in terms of mathematical relations whose domains and ranges denote the sets of values that are allowed for the environmental variables.
Similarly, the design specifications include descriptions of the behaviour of input and output devices. This behaviour can also be defined by mathematical relations between environmental state variables and variables describing the contents of the input/output registers. The software itself is nothing but a system that can be described by a set of relations whose range is the set of possible input register values, and domain a set of output values.

It is therefore not surprising that it has been found useful and natural to use functional mathematical relations to define rigorously the contents of the documents that contain the system requirements, the design specifications and the software requirements [6.9].

A second observation concerns the representation of these relations. In general the functions that are implemented by computer programs have a great many discontinuities. A program can be in many distinct states. And the same input can produce in each different state a different output; thus the set of outputs is a vector product of input and state vectors. The conditional expressions that describe these piecewise or discrete functions can be very complicated, but tabular representations have been found very convenient [6.9, 6.10] for systems of modest size, such as in an NPP. Common subexpressions can be factored out and, like matrices in linear algebra, the tables give concise and explicit descriptions of input/output relations. Other mathematical forms of specification that can be employed, using model based notations, are discussed in Section 5.2.4 (The Z notation discussed there is essentially a language of relations.)

6.3.4. Management of software documentation

For a sizeable system, the amount of documentation produced will be quite considerable. To ensure (and preserve) its accuracy, the initial production, review and maintenance of all this material requires strict management. For the documentation to be usable, ease of location and reference to particular items of information are also very important. This suggests that all the system documents should be produced within a ‘library’ or database under configuration management.

In practical terms, each specification and design document can be represented in a program development system (effectively a database) by a number of associated ‘objects’, such as its informal description and its formal specification. In the course of software development, further objects associated with each module are progressively introduced, starting with interface specification for the software, and followed eventually by its code body, objects relating to its verification, test and maintenance, and its change history. Every object has an associated ‘status’, controlled by the database management system, that indicates the validity of the object and determines how it can be employed.

A number of systems are available to support software development in this manner: the basic ideas are implemented in the library managers of all Ada
compilers, for instance. Also, systems such as the Rational Environment and the Xinotech Program Composer allow not only navigation through a program's documentation, and exploration of different kinds of objects at will, but also the extraction of selective 'views' of program code, such as procedure call trees.

Thus, the 'traceability' required for software comprehension and maintenance can be substantially improved with mechanical assistance.

6.4. SUMMARY AND FOLLOW-UP

Documentation is an integral part of a software based system: the precision with which the system development is documented determines the quality of the end product and the level of confidence that can be placed in its fitness for purpose. The quality of documentation also determines the ease with which a system can be maintained and the feasibility of performing subsequent system modifications.

Beginning with a detailed specification of system requirements, the successive documents produced should form a progression of 'refinements' of a design towards its detailed implementation, such that at every stage, the design decisions with their logical justifications are patently clear at every stage. Essential to clarity is the 'separation of concerns' in all these documents: for instance, requirement documents describe required behaviour, not implementation details, and documents for module interface specification define modules only as they are perceived externally, their internal designs being described separately.

The proper management of document production and review, and especially of maintenance of documentation, is an important matter, particularly for safety related systems. The task can be lightened to some extent by using an appropriate database management system, but it remains onerous. This topic is discussed in more detail in Section 4.

REFERENCES


7. PROGRAMMING

7.1. ISSUES

It has already been emphasized that for safety critical applications it is essential to establish by logical reasoning that the design and code implementation of a program are consistent with its specification. The extent to which this is possible, and therefore the program, after its construction, can be properly maintained, depends very strongly on the choice of programming language and the manner in which the program is employed. Section 7 discusses important criteria in choosing programming languages for safety critical applications, and good practice in program construction.

7.2. EXPERIENCE

When microprocessors were first applied to process control, the functions they performed were relatively simple. Although their programs were written in assembly code, usually in an ad hoc manner, manual inspection of a program could provide some comprehension of its behaviour, because the amount of code was very small. As the processing power and memory size of microprocessors have increased, control system designers have been tempted to exploit their capabilities, writing ever larger application programs. Even where structured assembly languages have been employed, the assembly code programs produced have often been of a complexity beyond human comprehension.

The code complexity problem was subsequently alleviated significantly by employing high level programming languages. However, these bring problems of their own: of ambiguity for instance. (As a very common example of ambiguity, if a programming language allows function subprograms to have 'side-effects' (that is, to change the values of non-local variables), and if the language does not prescribe the order of evaluation of the function calls that occur in an expression, then expressions can be of uncertain meaning. Most high level languages, including Pascal and Ada, are ambiguous in this sense.) The use of a high level language also involves application of a compiler, itself a large and complex program of uncertain behaviour. Finally, the use of a high level language does not per se guarantee the comprehensibility of a program text: this requires the use of the expressive power of the language to best effect.

7.3. CURRENT PRACTICES

7.3.1. High level versus low level programming languages

A safety critical program should be as simple and as small as possible, fulfilling its essential purpose and no more. If safety critical and also non-safety-critical
functions are to be performed by computers, they should be programmed separately and their programs should be run on distinct processors, in such a way that the execution of safety critical functions cannot be compromised by errors in any other components (for example, through corruption of common memory).

If this principle is respected, it may be the case that a safety critical program will be very small (amounting to no more than a few thousand machine code instructions). If so, careful programming in an assembly language (possibly using a structured assembler) is still the best option, as it avoids the risks associated with using a compiler. Subsets of assembly languages, adequate for this purpose, can be well defined, and small assembly code programs can be formally verified [7.1]. Assemblers are still needed to produce the executable code but these tools are relatively simple, amenable to validation. The choice of assembly code subsets and some issues in low level code programming are discussed in Section 7.3.5.

To produce larger programs, high level languages should be used. For an application still of modest size, it may suffice to employ a language such as Pascal, providing strong typing, structured statements and support for procedural or operational abstraction. For larger applications it is desirable to use a language that supports modular programming; that is, a language that provides 'modules' or 'packages' to support data abstraction and design in terms of abstract state machines. With these features, programming languages now allow quite direct implementation of design entities, greatly simplifying the task of producing programs that match their designs. Indeed, the programming process is becoming one of 'design refinement', with progressively more of the information that would previously have been regarded as 'specification' or 'design documentation' being incorporated in the 'program code'. This transfers a significant amount of the work of producing a correct program, and showing that it is consistent with its specification, from the programmer and verifier (human or mechanical) to the programming language and its compiler (the latter checking conformity of design and implementation, to some extent, through its syntactic and static-semantic analysis of program text).

7.3.2. Choice of a high level language

In very general terms, the essential requirement of a programming language, for safety critical applications, is that it should enable one to write programs of predictable behaviour. To achieve this, the language must have the following properties.

Logical soundness

To allow systematic reasoning about a program, the programming language must be logically coherent and unambiguous, with rigorously defined syntax and semantics.
Simplicity of formal language definition

For high dependability programming, the formal language definition is the necessary foundation for reasoning about programs, and the basis of tools for processing the language, such as compilers and formal verification tools. If the definition of a language is very large (as for Ada), logical coherence of the language is hard to establish, and reasoning about programs can become convoluted and uncertain, and rigorous tool development impractical. However, if the ‘core’ of the language is well founded, as for Pascal and Ada, these difficulties can largely be overcome, by employing only subsets that discard problematic features (see for instance Refs [7.2-7.4]).

Expressive power

The ease with which a program design can be ‘refined’ systematically towards an implementation depends very much on the expressive power of the programming language. For small applications, a language with the expressive power of Pascal is often adequate. For applications of significant size, facilities for modular programming (supporting the use of ‘specifications’ of modules, packages or subprograms, distinct from their code bodies) are necessary.

It must be borne in mind, however, that as expressive power increases, so does the complexity of the language and consequently the size of its compilers and other support tools. Languages whose nature and complexity make it difficult to check fitness for purpose of compiled code should be avoided; especially if their advanced features are in any case not particularly appropriate to the task at hand. For instance, an object oriented programming system may be well suited to the development of graphical user interfaces, where the notions of object classes and inheritance are very relevant and for which correctness is not of paramount importance. However, the use of such systems for implementing safety system control functions would not be desirable, since for applications of this nature they would scarcely simplify the design or programming, and verification of the executable code could be intractable. (This is not to decry the usefulness of object oriented analysis in establishing requirements, or of program design in terms of ‘passive objects’ such as abstract state machines.)

Language security

An insecurity is a feature of a programming language whose implementation makes it very difficult or even impossible to detect some violation of the language rules by analysis of a program’s text, by a compiler for example [7.5, 7.6]. (As an example of an insecurity, in the execution of an Ada program, procedure parameters
that are composite objects, such as records, may be passed by copy-in copy-out, or by reference, the mechanism being determined by the compiler. The Ada language has a rule prohibiting the construction and use of a procedure in a way that the effect of calling it depends on the parameter passing mechanism employed. However, it is so difficult to check whether this rule is obeyed that compilers do not usually do so.) All 'standard' languages suffer from insecurities. For some languages the insecurities can be eliminated, by discarding some language features and introducing additional static-semantic rules, without losing too much expressive power [7.2]. However, other languages (and notably C and C++) suffer from insecurities to such an extent that they do not have any useful, secure sublanguages [7.7].

Verifiability

Verification of a safety critical program should be performed both by analysis of its text and by testing. Checking that the text conforms to the specification should be essentially an audit (or preferably an assessment) of the successive stages of the program construction, using the documentation produced at each stage, and possibly with mechanical assistance (for example, generation and proof of verification conditions). ‘Reverse engineering’ — the process of extracting software design information from code — is intractable for complex programs, although it may be successful with some very small ones; and since its well foundedness is always questionable, it is inappropriate as a basis for verification of safety critical software. To this extent, a language appropriate for systematic implementation will also facilitate verification.

It is important to note that verification of a sizeable program, whether by formal methods or by testing, cannot be performed adequately by treating the program as a single entity. Verification involves checking correspondence of fragments of a program to fragments of its specification: it must be possible to reason about and test program components essentially ‘in isolation’, encumbered as little as possible by contextual information. This emphasizes the need for support of modular programming, with relatively simple scope and visibility rules.

Dependability of compilation

The behaviour of a digital controller is ultimately determined by the executable version of its software. Verification of the source code version is essential, but this does not guarantee the fitness for purpose of the executed code, as the correctness of compilers (and run time systems and post-compilation tools such as linkers and builders) must always be questionable.

The chosen high level language should be supported by a compiler (or cross-compiler) for the envisaged target processor with an approved international or national validation certificate. (It is important to note, however, that certification of a compiler does not imply that a compiler is error free; and indeed, some compilers
with validation certificates have been found to contain errors. Compiler certification is based on satisfactory behaviour in a number of tests; these may be numerous — typically several thousand test programs are used — but no testing process could adequately exercise a program as complex as a compiler. Nevertheless, compilers with validation certificates are strongly preferred.

Since the correctness of compilers cannot be guaranteed, thorough non-intrusive testing of their output is essential. For safety critical applications it is also very important to check compiler output against source code. (This difficult process is discussed further in Section 12 on Tool Support. The demonstration of equivalence of PL/M-86 source code and PROM contents is treated in Ref. [7.8].) Again, in choosing a high level language, it should be ascertained whether compilers for it exist whose mappings from source to compiled code are reasonably simple and well documented. The simplicity of run-time systems is another important consideration (which is also discussed in Section 12).

These criteria (and some others) are discussed at greater length in Refs [7.2] and [7.7]. (Although the criteria considered in these papers differ, there is substantial overlap between them, and the authors’ general conclusions are very similar. Reference [7.7] compares a number of languages, including C (and C++), Pascal, Modula-2 and Ada (and particular subsets of some of these) and makes specific recommendations.)

7.3.3. Programming: general considerations

Many papers, textbooks and manuals have been written on programming. The purpose of this section is to draw attention to aspects of programming that are particularly important to the production of safety critical software, rather than to attempt to give a comprehensive treatment.

Whether a program is to be written in a high level language or in an assembly code, the same general principles apply to its construction. The program should be conceived in a modular fashion, each module serving a well defined purpose (and specifically, representing a carefully chosen abstract data type or abstract state machine). Each module should be introduced by first specifying, very precisely, its interface with the rest of the program; it may be appropriate to express this specification in a mathematical notation (for example, a formal specification language such as VDM or Z) that can be mechanically checked, at least for syntactic and type consistency. The specification should be accompanied by concise natural language commentary. In the same way, each constituent procedure subprogram should be chosen to serve a single purpose, and should be introduced by first defining its effect (a transformation of the state of its calling environment). This can be achieved by specifying its preconditions and postconditions, in a formal notation (with natural language commentary, where appropriate). Likewise, function subprograms (whose
execution should only evaluate a property of state, without side-effects) should be introduced by specifying their preconditions and the values they return.

The program construction proceeds by a succession of stepwise refinements of the program units (modules and subprograms), where for each unit its constituent program units are chosen and specified, and so on. When eventually the bodies of subprograms and modules are implemented in code, the implementation details are ‘hidden’ (confined in scope and visibility to the unit bodies) as far as possible. In this way a program is produced as a hierarchical arrangement of program units, each with a specification stating its purpose, and a code body to fulfil that purpose.

When a high level ‘modular’ programming language is employed, features of the language (such as support for separate definition of the specifications and bodies of modules or packages, and subprograms, and for incorporation of such program units in a library system) can be used to describe the skeletal design of a program, and subsequently to add its details. Thus, insofar as it may be appropriate to use the programming language to express features of a design, the processes of ‘design’ and ‘programming’ may be performed in tandem. However, in using lower level programming languages, which do not support design so well, the design and coding activities are quite distinct.

Verification of the code requires demonstrating (through formal proof and by testing) that the body of each unit does indeed implement its specification. In verifying program units that contain subunits, formally or non-formally, the behaviour of the unit as a whole can be considered logically in terms of the specifications rather than the implementation details of the subunits.

The method of modular programming described here, using ‘abstraction through specification’, contributes to program correctness, and simplifies program construction, verification and validation, and maintenance in several different ways. Firstly, it provides a clear and unambiguous description of the design and implementation of a program, from the topmost module level down to the level of code implementation at modular level. At every level, the specification and implementation of a program unit are comprehensible to the (human) reader, allowing their consistency to be checked by manual inspection, unit testing and formal verification. Furthermore, the clarity of the documentation facilitates the checking that functional and safety requirements are met.

Programming is greatly simplified and is made less error prone by the fact that in specifying or coding one program unit that employs another, the programmer need only consider the specification of the latter and not its implementation details. This means furthermore that the task of developing a program can be shared between teams, or individuals, without risk of confusion: each programmer only requires formal specifications of common interfaces. Other benefits are the simplification of construction of test cases for unit test, easier detection of program errors, and considerable savings in maintenance (since the ramifications of code changes are easier to establish, and in any case more localized).
7.3.4. Programming in high level languages

The dominant consideration in coding individual program units should still be the need for program text to be easily comprehensible and amenable to rigorous analysis. Only language constructs with well-defined semantics should be employed, but with the proviso that the expressive power of the language should be exploited to make program text ‘self-documenting’ as far as possible and to exploit the capabilities of a compiler for coherence checking.

The scope of program entities should always be as small as possible. Data types should be chosen to allow type checking to provide maximum protection (for example, enumeration types should always be employed in preference to numeric encodings, and the values of integer variables should be constrained to lie within their permissible ranges through their type declarations. All constraints should be statically determinable (making it possible, in principle, to determine the memory requirements and allocate memory positions to all data at compile time). Real arithmetic (as opposed to integer arithmetic) should only be used with great caution. (For an account of its pitfalls, which are not always taken into account even in compiler construction, see Ref. [7.9].) Every subprogram should be designed to fulfil a single, well defined purpose; the text of its code body should not occupy more than a page. It should be single entry single exit, with a simple control structure (not containing loops with multiple entry points, for instance).

Care should be taken not to employ language constructs that have very complicated semantics, or that can introduce insecurities, or whose use inevitably leads to the generation of low level code that cannot easily be tested or otherwise verified. For instance, one should avoid use of dynamic pointers, and all language features whose compilation inevitably introduces dynamic heap storage allocation.

The use of a good naming convention, appropriate indentation of text, and the judicious inclusion of comments can markedly assist human comprehension of a text. Comments should be very precise, and add useful information; for instance, the inclusion of loop invariant assertions is strongly encouraged. Self-evident marginal comments that simply paraphrase code statements should never be included.

7.3.5. Programming in low level languages

Many of the recommendations of the previous section apply equally well to assembly code. In particular, the need for subroutines to be small and for their control structures to be relatively simple is rather obvious, as are the benefits generally of good layout and informative comments.

Many microprocessors have in their instruction sets some instructions that are not clearly defined, and also instructions (such as indirect jumps and self-modifying code) that severely complicate reasoning about code behaviour. Such instructions can also make testing and formal verification intractable, and should not be employed.
The kinds of instructions that should not be used (and that are banned by the ‘codes of practice’ commonly enforced, for instance in the safety critical avionics industry) are discussed in some detail in Ref. [7.10].

7.4. SUMMARY AND FOLLOW-UP

In this discussion we have only considered the use of imperative languages (such as assembly codes, Pascal and Modula-2). Two other possibilities require mention, namely object oriented programming (in languages such as Smalltalk or Eiffel) and logic programming (in Prolog for instance). Object oriented programming would not be particularly well suited to the implementation of control functions, but some of the control functions arising in an NPP could be defined (and thus implemented) clearly in a logic programming language. However, this would involve the use of a logic program compiler or interpreter and run time system of considerable complexity, which could not be verified. The complexities of the support software, for both object oriented and logic programming, rule out their use in safety critical or even safety related components of an NPP, for the present at least. Finally, the lack of mature and robust support for functional programming also rules this out.

The most important conclusion is that safety critical software components of an NPP should be as small and as simple as possible, and that they should be run on processors dedicated to the purpose. If, as is very desirable, each distinct component consists of only a few thousand lines of code, then rigorous development in an assembly language is still the safest approach. Although the kinds of entities employed in good software design (such as modules) are not directly supported by assembly languages, the principles can still be applied. Formal specification of requirements and formal description of the design is advocated, and for small systems formal code verification is possible.

Use of a high level language contributes greatly to the clarity of the implementation, lightens the burden of producing it, and allows quite extensive checking (by a compiler) for coherence of the program text. Conformity of this text to the design specification is checked much more easily than correctness of an assembly code program. The only inhibiting factor — but an important one — is the need to employ a compiler to generate the executable code. The difficulty of verifying a compiler thoroughly, and of checking its output against its input, is always severe, and especially so for the larger programming languages. The size of each safety critical software component of an NPP should therefore be severely restricted, to the extent that it can be implemented satisfactorily in a relatively simple high level language, such as Pascal (or better, a well defined subset of this).

For software whose failure would not jeopardize the safety of the plant in any way, the use of a larger language providing direct support for modular programming (such as Modula-2 or even a secure Ada subset) would be appropriate. It is to be
hoped that in the near future compilers for modular languages will produce object code amenable to rigorous analysis, in which case their use for producing safety related software could be contemplated.

REFERENCES


8. VERIFICATION AND VALIDATION

8.1. ISSUES

Verification and validation deal with the issue of how software can be shown to be in conformance with requirements. An important aspect is demonstrability: it is not enough that software meets its requirements; it must also be possible to demonstrate to a third party (such as a licensing body) that all requirements have been met. Verification refers to the stepwise checking that each product of the software development process is correct with respect to preceding products (specifications and design descriptions) and standards. Validation refers to the higher level checking to determine whether all user requirements and safety requirements have been met, both by the specifications and by the final product. The other side of verification and validation is the finding of errors. If correctness cannot be demonstrated, then an error must exist and must be corrected.

Verification and validation should cover the entire system and all of its functionality. However, the techniques described in Section 8 can also be used for auditing the development of a system, for example by regulators or their representatives. In such cases, a subset of the system and its documentation is selected for audit and a technique is chosen that best covers the regulatory requirements.

There are a few fundamental principles which are generally agreed upon, but which are not yet generally applied:

1. It is essential to get project wide agreement on the requirements that the final product must meet and on its quality attributes.
2. Each development stage must produce some kind of product which can be verified (generally a document).
3. Verification must take place throughout the development. It is much easier and less expensive to correct errors found in the early stages of development than those found in the later stages.
4. The major development products must be verified and validated by people independent of the people who developed the product.
5. The use of multiple verification and validation techniques is better than the use of a single technique. Independent verification should use techniques different from the techniques used by the developers.

8.2. EXPERIENCE

Software development in general has been notoriously poor at abiding by the aforementioned principles, with the result that software projects have a reputation for being late and over budget, and software products have a reputation for being of poor quality.
There are two main reasons given for not getting project wide agreement on the requirements and quality attributes: ‘Everyone knows what it’s supposed to do’ and ‘No one knows yet what it’s going to be able to do’. The first reason is used to justify the lack of a precise requirement specification. However, when requirements are not carefully and completely documented, the discovery of differences of understanding is delayed until the software is in its final stages of testing. This situation occurs with safety system software when application (nuclear safety) experts do not communicate all their requirements to the software experts who are developing the product. The second reason is used to justify the lack of a requirement specification because what can be done with the low level technology is driving what will be done. This situation often occurs with user interfaces, with the result that the operator has to adapt to the new technology rather than being given a product adapted to human needs.

It has been unusual to find software development projects proceeding through carefully documented and verified stages because there has been so little agreement about what stages there should be and how each stage should be documented. Also, there is a perception that documentation is intangible and therefore useless. The result has been a strong pressure to start writing source code as soon as possible so as to have a tangible product, no matter how error filled, to demonstrate some progress.

Similarly, because the work of independent verifiers is intangible, it is less likely to be funded than the work of developers. When time and/or money is in short supply, independent verification is generally the first area to be cut unless mandated. One must have a long term view of a situation to recognize the benefits of independent verification. The use of multiple techniques is also a short term cost for the potential of a long term gain. The perceived cost of an undetected error must be fairly high in order to justify the use of a second or third verification technique.

8.3. CURRENT PRACTICES

8.3.1. General

Verification and validation techniques can be grouped into several categories. The techniques of each category have their particular strengths and weaknesses. In the subsections which follow, informal techniques are described first and more formal techniques later. Less formal (less mathematical) techniques apply to less formal documents, but have the strength of finding the more essential, broader errors affecting the principal aspects of the system. More formal methods require formal documents and are better at finding subtle errors and errors of detail. For any system, a combination of techniques should be used with a good balance of strengths.
8.3.2. Reviews, walkthroughs, inspections and audits

Reviews can be divided into several subtechniques, such as walkthroughs, inspections and audits. The common feature is that human interactions play a dominant role. In contrast to more formal techniques such as static analysis or testing, it is the human, intuitive understanding that is important in reviews.

Reviews by a verification and validation team can be conducted at any stage of the development process. They generally consist of an independent person or team examining the work of the developers with the goal of finding errors as soon as possible. A difficulty with all types of reviews is that they do not have a fixed scope. The review process could expand to fill any amount of time and there is no fixed end point since one can never be sure that all errors have been found. A common solution is to use checklists, either project specific or generic, to control the scope of the review. When all items on the checklist have been covered, then the review stops. Volume 2 of Ref. [8.1] gives checklists for reviews of all phases of software development. Chapter 3 of reference [8.2] gives detailed guidelines and checklists for code inspections.

Maintaining the independence of the reviewers can also be a problem. If reviewers are allowed to discuss the project with developers and are allowed to suggest alternatives or solutions to problems, then they are contributing to the development and can no longer be considered independent. They are less likely to find errors in a design they have suggested. To control this, some project managers insist that all interaction between developers and reviewers be indirect, by means of written communication. The comments from reviewers to developers must be the identification of problems or errors only. A problem identification report must be separate from a change proposal.

A walkthrough is generally understood to mean a meeting at which the developer describes a development product to a team of colleagues and/or independent reviewers in an orderly, stepwise manner and the reviewers question the developer and try to identify errors and problems. If the development product is a detailed design or code, the walkthrough often consists of stepping through the algorithms and trying to understand what the computer will do at each step. Walkthroughs can also be used at early stages of a project and can facilitate better and more uniform understanding of the requirements.

Inspections are similar to walkthroughs except that the initiative comes from an independent inspector rather than the developer. The inspector must study the documentation under review prior to the meeting and set the agenda for the meeting by asking questions.

Inspection and walkthrough techniques have been structured into a method in which the roles of the team members are strictly defined and the process is carefully controlled [8.3, 8.4]. The roles are:
— *Moderator*: keeps the process focused on the problem at hand and makes sure that the results are properly recorded;
— *Implementer*: presents the material to the group and explains step by step the expected operation of the logic;
— *Inspector*: studies the material in advance and asks detailed questions;
— *Designer*: understands the previous phase of the development and looks for errors and discrepancies between the result of the previous phase and the material being presented;
— *Tester*: suggests test cases for ‘input’ to the process; understands expected ‘output’ and looks for discrepancies.

These techniques do allow a thorough review and better understanding of a development product, but are not good at finding errors of omission. The process leads reviewers to make the same mistakes as the developer if there is a reasonable sounding explanation hiding the error. It requires a special skill to ‘play computer’ and to recognize the potential problems a real computer may encounter when executing an algorithm.

Walkthroughs can also be useful and successful when conducted internally amongst a development team. If there is no need to maintain independence, they can help the team to work together and understand each other’s contributions. Promoting such a team understanding of the system helps in making it more internally consistent.

Audits are reviews in which the initiative is with the auditor. The auditor asks the developer questions about the product, who must provide the answers. The auditor generally chooses to restrict his/her attention to certain parts of the product which are chosen randomly or by regulatory or management criteria. Some problems experienced with audits are discussed in Section 4.2.3.

### 8.3.3. Testing

For a discussion of testing, see Section 9. Note that testing is considered to be a method of verification and validation rather than considering verification and validation to be two types of testing.

### 8.3.4. Specification animation or executable specifications, simulation and prototyping

The techniques of ‘specification animation’, simulation and prototyping are similar in that their goal is to produce a partially working version of the software before the real code is written, so that it can be examined by potential users and tested. Specification animation refers to the technique of executing a specification either directly on a computer (if the specification is written in a sufficiently formal,
executable language such as a logic language) or by mechanical translation into executable code. Such an animation is usually not fully functional nor can it operate in real time. It may operate on a computer other than the target computer. Simulation refers to a similar technique applied to the design. Often, the target computer will be simulated on a more general purpose machine, or a variety of algorithms will be simulated in partial form to evaluate design possibilities. Prototyping refers mainly to a partial or intentionally crude implementation built to evaluate possible design choices or possible user interfaces.

All of these techniques are of the most use when the correctness of the specification or the trade-off between design possibilities are unclear. A partially working version of the software can be shown to users even while the requirement specification is being written, in order to provide focus for the discussion and to bring out requirements which the user may have forgotten to mention. A prototype of the user interface is valuable when the proposed system is unlike any existing system. Because software and computer technologies are so new and are continually changing, use of a simulation to evaluate design possibilities may result in improved designs. These techniques are primarily development oriented rather than verification oriented, but they do provide some sort of early validation.

A problem with all of these techniques is that the animation/simulation/prototype is not the product, yet the people working with it often lose sight of the difference. It is important to evaluate the prototype only in the aspects where it resembles the product and to refrain from evaluating the aspects which differ. In developing real time and interactive systems, it is almost impossible to create a prototype system that has the correct timing, yet this is obviously an important aspect of the eventual system. The value of an animation or prototype also depends on the test cases that are chosen. This technique tends to find errors which concern the principle of the software system or its feasibility, but cannot give confidence as to the completeness or overall correctness of the specification. Finally, this kind of prototype is a 'throwaway' prototype and must be thrown away rather than being used as a building block of the real product.

8.3.5. Data flow, information flow and control flow analysis

The techniques of data flow analysis and control flow analysis can be applied to source code, assembly code or particular kinds of design documentation. Data flow analysis traces the movement of information through the system between input and output. If the analysis is to be done on code or on formal design documentation, then automated tools exist which will assist with the analysis (Refs [8.5–8.8]). Data flow analysis generally consists of checking:

— that all input data are used;
— that variables are set before they are used to compute another value;
— that all outputs are given a value;
— that outputs are derived from the correct set of inputs;
— that variables, inputs and outputs always have values of the correct type;
— that the use of global variables is restricted.

Control flow analysis traces the execution sequence of the software. Again, automated tools can assist the analysis if done on code or formal design documentation. Generally, a graph is constructed to model possible execution paths through the software and then an analysis of the graph can be made to determine various properties:

— whether only ‘structured’ constructs (sequence, alternation, iteration) have been used;
— the existence of unreachable code;
— the number of entry and exit points for routines and loops;
— whether the size and complexity of subprograms are within acceptable limits.

Data and control flow analysis, in practice, have not been found to be very effective for full verification because of the limited sorts of errors that they are capable of detecting. However, they do find those classes of errors very well and are usually cheap and easy to use (checking can be largely automated), so it is generally worthwhile to do this.

Both data and control flow analysis can only provide information about the properties of the design or code; in order to evaluate the results it is also necessary to have project standards concerning which properties are to be allowed or disallowed. Analysis can 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 [8.3].

Data and control flow analysis can also be used as methods of generating and selecting test cases for white box testing (see Section 9).

8.3.6. Code review and analysis

There is generally a need to verify the code against the design. This is called code review. One technique for performing this verification is to extract or recreate design information from the code and to compare the recreated design against the original. This process of code analysis is similar to the technique of ‘reverse engineering’, but the design information extracted from the code should not be used to substitute for or replace missing design information.

Code analysis is good at detecting errors, detecting additional unintended functionality added during the coding, and providing important evidence about the validity of the coding process. To be valid, the analysis must be performed by people
who did not take part in the coding. Code analysis is also useful to define and select test cases for parts of the code that are found not to be covered by the tests specified from the requirements.

The main problem with this kind of code analysis is that it is difficult, laborious and expensive to apply to any but the simplest code. It works best if the coding process is simple and reversible. Since these cases are rare and not identifiable in advance, the benefits cannot be guaranteed and may be doubtful.

8.3.7. Formal verification and program proofs

If the specification and design of the software are written in (or translated into) a formal language, then it is possible to perform a formal verification that the specification has been met. A formal language is a language with precisely and completely defined syntax and semantics; it is possible to determine unambiguously whether or not a given statement is syntactically correct and it is possible to determine a precise meaning for each syntactically correct statement. Some types of formal verification are checks that all statements in the specification are coherent, that design and code are syntactically correct, that the design is consistent in meaning with the specification, and that the code is consistent with and correctly implements the design.

Proof of properties of specifications

Properties of specifications may relate to consistency of a specification or completeness of operation definitions, or proof that the specification meets certain essential requirements. (See for instance Ref. [8.9], which proves properties of the specification of a storage manager.) It must be said, however, that proof generally requires considerable expertise. Tools now exist to support this work (see Section 12.3.2), but these still require a considerable amount of skilful interaction.

It is important to bear in mind the fact that a formal specification is based on a conceptual model of the real world. Engineers are well used to employing models, and know that they have their limitations. It may be possible to prove that a formal specification is logically coherent in various ways and that it has certain properties that are considered important; but since the world is not a formal system, it could never be proved that a formal specification is 'correct'.

Program proving

Program proving deals with the effect that program statements have on the program data. Currently the predicate logic described in Ref. [8.10] forms the mathematical base.
Proofs may be conducted either in parallel with software development or during a separate verification stage. The basic method is to have a predicate on the program data which is true at the beginning of the program and a predicate which is required to be true at the end. Proving is done by demonstrating that the effect of the program statements is to cause the final predicate (postcondition) to be true if the beginning predicate (precondition) is true. If the program statements include a loop, then it is also necessary to demonstrate that the loop will terminate.

Advantages and disadvantages

The advantages of using formal methods are that a formal specification is much better than an informal specification as a basis for product development: it can be validated against user needs at an early stage, can serve as a contractual document between customer and developer, and can be used to answer questions about what the product should do for use in later verification, including testing. Formal verification gives a high level of confidence in the completeness and correctness of the product with respect to the specification. Since formal methods include precisely defined syntax and semantics, it is possible (although difficult) to automate some of the syntax checking and proof checking, thus reducing the chance of human error.

The disadvantages of formal methods and proofs are that they require program understanding to a certain degree. As a consequence, the proofs are error prone, costly and time consuming. Proofs are difficult to perform and can be meaningless, as the predicates become very long even for medium sized programs. To write formal specifications and to do program proving requires specially trained people because thinking in terms of predicates on data spaces is not common among programmers.

Because a full formal verification is very laborious and expensive, it is not often done in practice. Two possible compromises are suggested:

(1) if the safety critical portion of the software can be identified and clearly separated from the non-critical portions, then it may be possible to do complete formal verification on the safety critical portion; or

(2) if a formal specification exists, then it may be satisfactory to perform a less formal, more intuitive, but still mathematical verification, in the form of a demonstration rather than a complete proof [8.3].

The practical use of formal methods is not widespread. Many formal languages have been proposed, but none has achieved predominance. There are still disagreements about the semantics of some of these languages. It is not clear whether they are sufficiently flexible to describe all necessary meanings. Not enough people have been completely trained in their use. If experts in formal methods are brought onto a project, then there is a danger of incomplete or faulty communication between the application experts and the formal methods experts, similar to that mentioned in
Section 8.2. Formal proof is difficult for humans to do correctly and is often too intuitive to be automated. The most successful approach to date has been to restrict the complexity of systems to be formally verified. Research is now being focused on how to formally verify more complicated systems composed of formally verified simple components.

8.3.8. Symbolic execution

Symbolic execution is, in some ways, a combination of data flow analysis and formal verification. Symbolic execution depends on having a formal description of the software (source code is in a formal language, even if the specification and design are not) and traces the flow of information through the software from input to output. Inputs are replaced by mathematical symbols of the appropriate type, and the operation of the software on these mathematical entities is simulated (by hand or by an automated tool). The result should be a mathematical expression for the outputs in terms of the inputs. This mathematical expression must then be compared against the specification.

Symbolic execution is limited to fairly simple pieces of software, and requires a mathematical understanding of the requirements (see Ref. [8.11]).

8.3.9. Safety verification

In addition to or as part of their functional requirements, safety critical systems also have specific system safety requirements which should be verified. The software safety can be verified by an extension of the system safety analysis: identifying the hazards of a system and tracing them back to and through the software. A hazard is a system state which can lead to an accident, given certain environmental conditions. Risk is the product of the probability of a hazard occurring, the probability that the hazard will result in an accident and the worst loss associated with the accident.

Software can increase the risk by allowing the system to reach a hazardous state, by failing to detect and recover from a hazardous state, or by failing to mitigate the damage when an accident occurs. By tracing hazards back through the software, with these possibilities in mind, it is possible to detect logic errors, determine where fault tolerant or fail safe procedures should be initiated, guide placement of run time checks, and pinpoint critical functions and test cases. This can be done in a way which is separate from and complementary to functional verification; safety verification can detect situations in which the functional requirements specify an unsafe process.

The most commonly used technique for safety analysis is the fault tree approach. This technique is described in detail for Ada software in Ref. [8.12]. The methods would be applied in a similar way for other programming languages. The
basic principle of the technique is that of mathematical proof by contradiction: it is hypothesized that the software has produced an unsafe output and it is shown that this could not happen because the hypothesis leads to a contradiction. Other techniques are also possible with this same basis (such as Petri-net analysis or state reachability graphing), but the fault tree method has the advantage of focusing only on identified hazards rather than on the complete state space of the software.

Safety verification is important for safety critical systems and software because it focuses on safety rather than functionality. However, for this very reason, it should be used in addition to, not instead of, other verifications.

8.4. SUMMARY AND FOLLOW-UP

Effective verification and validation are more widely preached than practised. For applications where safety, security or financial risk are major concerns, there is a growing tendency for software projects to include independent, formal and semi-automated verification and validation. Section 8 describes a number of verification and validation techniques and indicates their strengths and weaknesses. Since no single technique is perfect, a set of diverse techniques should be used. For safety critical software, this set should include formal verification by an independent team as well as testing (see Section 9). A technique which promotes understanding and communication between application experts and software experts will lead to a better final product.

Further development of verification and validation techniques is needed. In order to have proper and effective verification and validation it is necessary to have:

— the ability to quantify the improvement in quality resulting from the use of the various verification and validation techniques (see Section 10);
— properly trained and experienced personnel; and
— automated tools which could be used without extensive training and user participation.

The acceptance and use of software verification and validation by industries involved with safety critical applications is expected to solve some of these problems by providing a larger base of documented projects and a pool of experienced engineers.

REFERENCES


9. TESTING

9.1. ISSUES

In this report, testing is considered to be a particular method of verifying and validating software, rather than verification and validation being two types of testing. Because testing is such a widely used method and because there are so many types of testing, Section 9 specifically covers issues relating to testing.

Software testing has been defined as:

"The process of executing a program with the intent of finding errors". [9.1]

Myers’s emphasis in Ref. [9.1] is that the goal of testing is to find errors in the software, not to show that the software is correct. In addition to this definition, there are some other important requirements of testing (which also serve to distinguish testing from other verification and validation methods):

1. an executable program or piece of software;
2. a computer and associated systems to execute the program;
3. test cases: inputs used by the program;
4. an independent method of determining whether or not the outputs of the program are correct.

The main issues in software testing are requirements (3) and (4): How should test cases be generated and selected (and how many are necessary), and how can the correctness of the outputs be determined?

9.2. EXPERIENCE

9.2.1. State of the art of software testing

Reference [9.1] is often quoted as a basis for software testing programmes. Its title is indicative of the perceived nature of the field: ‘The Art of Software Testing’. The author, and many current practitioners, believe that software testing is an art: meaning that it is something that is done by humans on a subjective basis, some people being more skilled than others, some techniques working better than others, but with no mathematical foundation and no understanding of why and how it works. This belief, although still widely held, is somewhat outdated. More recent research has established a better mathematical foundation for testing than for most other areas of software engineering [9.2].

Software is intangible. It is pure information. Therefore, a software error is an information error: a misunderstanding; a design error. A software error manifests
itself as a fault or faults in the code; a fault being a piece of code which incorrectly performs a function or performs an incorrect function. A failure is a discrepancy between the required behaviour of the system and the actual behaviour of the system. Testing is a process of observing a system's behaviour under a selected set of circumstances and comparing actual behaviour with required behaviour. The goal is to find errors, but failures (or a lack of failures) are what is observed. The indirect process of tracing failures to discover the design errors is what complicates understanding of testing as a means of error detection.

9.2.2. Classifications of testing

Software testing is a very large field. Many kinds of testing have been used and proposed with a variety of purposes and techniques. In order to understand where the techniques fit into the overall field of testing, Section 9 shows some of the classifications of software testing. These can be thought of as a set of independent or semi-independent classification schemes:

Types of testing by system knowledge

Black box: only knowledge of external input and output requirements is used to select test cases and to determine expected behaviour.

White box or clear box or glass box: knowledge of the internal details of the software design is used to select test cases and to determine expected behaviour.

Levels of testing

Unit testing: individual subroutines, procedures and functions are tested through the use of a special 'test harness' (an external interface to the unit) or a special test environment.

Module testing: each module is tested separately through the use of test drivers which exercise the complete module interface and function.

Integration testing: testing which specifically tests the interactions between modules and the interactions between software and hardware.

System testing: testing done on the complete system of hardware and software (and operator).

Approaches to testing

Non-incremental: each unit or module of the software is tested separately and in isolation by using special test drivers and test harnesses. When all pieces have been tested and errors removed, the entire software is assembled.
Bottom up: testing begins with unit tests of the lowest level units (those which do not call any others). Testing of higher level units (those which call other units) is done by incrementally building the software. Test drivers are needed to call and pass parameters to the highest level of unit being tested.

Top down: testing begins with the highest level unit (a unit which calls other units but is not called by any other). Testing of the higher level units is done by initially replacing lower level units by 'stubs' (non-functional components) for units and modules which have not been completed. As testing proceeds, each stub is replaced by a fully functional component and test cases are added which exercise the component's functions.

Functional increments or thread testing: The system is built and released for testing as a series of operational systems which make more and more functions available on an incremental basis.

**Purposes of testing**

*Debugging:* testing done by the software developers with the direct intent of removing mistakes (bugs) in the code.

*Design verification:* testing which checks whether the software design has completely and accurately addressed all software requirements.

*Prototyping and specification animation:* testing of a preliminary or incomplete version in order to reveal problems and errors in the specification of system requirements and especially the user interface.

*System acceptance:* testing performed by the customer or customer's representative to determine whether or not the system meets the minimum criteria for acceptance.

*Certification:* testing performed by a certification body or their representatives to determine whether or not a system meets certification standards.

*System validation:* testing to determine whether a system meets user needs and/or correctly models the real world.

*Reliability estimation:* testing to allow quantitative estimation of the expected reliability of the system (this point is extensively discussed in Section 10).

*Stress testing:* testing a system under extreme loads (quantity or rate of inputs) to discover errors and/or to determine the limits of the system.

**Issues in testing**

(These are included as a classification because certain testing methods and techniques have been developed specifically to address one or more of these issues.)
— test coverage
— operating profile
— generation and selection of test cases
— quality of specification
— testability of requirements
— generation of expected outputs/desired behaviour
— timing and performance.

9.3. CURRENT PRACTICES

9.3.1. Logic coverage

Logic coverage [9.1, pp. 37-44] is a set of criteria for evaluating test cases used for white box testing. It is applicable to small units of code being tested for debugging or design verification. The criteria are applied as a method of test case generation and selection in order to ensure test coverage. The criteria can also influence design and coding: it is easier to show logic coverage for some design and coding constructs than for others.

Described here are six progressively more rigorous criteria:

— statement coverage
— decision coverage
— condition coverage
— decision/condition coverage
— multiple condition coverage
— path coverage.

The weakest logic coverage criterion is statement coverage which requires only that test cases be selected which cause every statement in the program to be executed at least once. This criterion is weak because it only requires one test of each statement (which may not be enough to discover an error) and because many decision statements will not be properly tested (in particular, IF statements with no ELSE clause). This criterion is sometimes strengthened by applying it at the object code level, rather than at the source code level.

Decision or branch coverage requires that test cases be selected such that each decision in the program has a TRUE and a FALSE outcome at least once. For multi-way decisions (such as CASE statements), each branch must be selected at least once. Decision coverage usually also satisfies statement coverage and is a stronger criterion. However, it is still rather weak because of the fact that decision statements often contain multiple conditions (combined with boolean operators AND and OR). It is sometimes recommended that this weakness be avoided by insisting that programs only contain single condition decisions, but this impairs readability and understandability.
Condition coverage requires that test cases be selected such that each condition in each decision takes on all possible outcomes at least once. Unfortunately, it is easy to envision a decision containing multiple conditions whose overall outcome could remain constant even though each individual condition goes through all possible outcomes. Therefore, this criterion is better combined with decision coverage. Decision/condition coverage requires that sufficient test cases be selected that each condition and each decision take on all possible outcomes at least once. However, when source code is translated into machine code, decisions and conditions are often reorganized; thus this criterion has the weakness of not exploring all possible outcomes of the decision when it is translated into machine code.

Multiple condition coverage requires that test cases be selected such that all possible combinations of all conditions in every decision are invoked at least once. This is quite a strong criterion and will cover the logic of programs containing sequence and branching. The remaining weakness is that not all possible paths will be covered, especially if the program contains loops.

Complete path coverage is the strongest criterion for logic coverage. It requires that test cases be selected such that all possible paths through the program are exercised at least once. Path coverage is often infeasible for even fairly 'small' programs containing loops. Reference [9.1] contains an example of a small program of about 20 statements (a DO loop with nested IF statements) which has about $10^{14}$ possible paths.

In the nuclear industry, another type of coverage is often used: accident and transient coverage. This is a validation criterion which is intended to ensure that all risky situations will be handled safely by the system.

### 9.3.2. Equivalence partitioning

Equivalence partitioning [9.1, pp. 44–50] is a method of generating and selecting test cases for black box testing. It can be used at any level of testing where a black box specification of the item under test exists.

Equivalence partitioning consists of two steps. This first is to identify the equivalence classes of input conditions in the specification. There are two types of equivalence classes: valid equivalence classes and invalid equivalence classes. A valid equivalence class is a set of inputs for which the program is expected to produce valid outputs such that if one member of the set does not detect a fault, then it is unlikely that any other member of the set would detect a fault. (If the black box specification specifies a piecewise continuous function, then each piece constitutes a valid equivalence class.) An invalid equivalence class is a set of inputs which the program is expected to recognize as invalid on the basis of a single condition. The second step is to identify test cases (particular inputs) which cover all equivalence classes. The rules are that test cases should be generated such that each covers as many valid equivalence classes as possible until all valid equivalence classes are
covered, then a test case should be generated for each invalid equivalence class so that it covers one and only one invalid equivalence class.

For example, suppose a program is required to calculate the area of a triangle given three inputs representing the lengths of the three sides. Some valid equivalence classes are supplying exactly three inputs, three inputs representing a real triangle, all inputs positive, all inputs integers and all inputs real numbers. Some invalid equivalence classes are fewer than three inputs supplied, more than three inputs supplied, three inputs representing a straight line, three inputs which do not represent a real triangle and one or more inputs negative.

This technique is a good way of making sure that all aspects of a program have been tested for system or component acceptance. The criteria of equivalence class coverage can also be used to assess system validation tests and reliability tests. However, this technique is not particularly good at finding errors; the assumption of equivalence is not really valid for black box, digital systems. Some members of an equivalence class may be handled incorrectly by a program while other members of the same equivalence class are handled correctly.

9.3.3. Boundary value analysis

Boundary value analysis [9.1, pp. 50-55] is a method of generating and selecting test cases which are expected to have a good chance of detecting errors based on industry experience. It is often associated with equivalence partitioning and is also used for black box testing of components having a good black box specification.

Boundary value analysis differs from equivalence partitioning in two respects:

— rather than selecting any element of each equivalence class, multiple elements are selected at the edges or boundaries of the class;
— analysis is also done on the output space to determine output equivalence classes, and inputs are chosen to generate outputs at the boundaries of the output equivalence classes.

This technique is good for finding errors and debugging. It is used for system validation and can also be considered as a kind of stress testing (especially when volume and speed boundaries are tested). The main difficulty is that it often requires creativity and intuition to find all the boundaries which should be tested, and it is difficult to prove that boundary coverage is complete without examining internal details of the program. On the other hand, if the internal details of the program are known, then the correctness of the boundary handling can be determined by static analysis rather than testing.

9.3.4. Cause–effect graphing and logic tables

Cause–effect graphing [9.1, pp. 56–73] is a technique of translating a natural language specification into a particular formal language (a cause–effect graph and
logic tables) which lends itself to the generation of test cases. Reference [9.1] describes this technique and shows an example. The cause–effect graph is a boolean logic graph which identifies all possible input cases (causes) and all possible output cases (effects) and links them with a graphical expression of the boolean predicate dictated by the specification. This graph can be mechanically transformed into an equivalent logic table form (tools can perform this process automatically) and test cases can be generated to cover each predicate (each path in the graph or each column in the logic table) and/or the boundaries of each predicate.

The benefits of this technique are similar to the use of any formal method: a systematic, mathematical expression of requirements facilitates complete verification and testing with automated tools. The process of translating a natural language specification into a formal language often uncovers ambiguities and incompleteness in the specification. The disadvantages are that the translation is difficult and costly to perform on large systems (a piecewise approach is often used) and the resulting formal specification must be carefully verified against the original specification and validated against real user needs. These tasks are complicated by the fact that special training and tools are required in order for people to write, read and verify formal specifications.

9.3.5. Functional testing

The method of functional testing [9.2, 9.3 pp. 53–59] is to identify the functions in the software and to test the input/output behaviour of each function over test data which are selected specifically to distinguish the correct function from incorrect functions. This method relies on the software requirements being described in a functional form (so that there is a way of determining if the output is correct for a given input) and the software being designed and coded using a limited number of analysable constructs. With these restrictions, a mathematical foundation for functional testing is given in Ref. [9.2].

The design constructs for functions which allow analysis are the same as those used for 'structured design': sequential expressions, conditional branching and iteration. It is well known that these constructs are sufficient for almost all software. Reference [9.2] provides simple rules for selecting test cases to test functions adequately using these constructs.

Functional testing provides better assurance of complete logic coverage, at the expense of restricting the design method. It should also be pointed out that some of the theoretical basis for functional testing requires an additional 'competent programmer assumption'. This assumption can be interpreted as meaning that the programmer (or designer) will not implement a totally different function than the one required and/or will not actively strive to implement a function containing an error that is concealed.
Functional testing must be supplemented with other data flow based testing to ensure that the data are passed correctly between functions, inputs, outputs and internal storage (see Section 9.3.10).

9.3.6. Program mutation testing (error seeding)

Program mutation testing [9.3 pp. 42–48] is a technique for measuring the adequacy of test data, not a technique for testing. The technique consists of ‘mutating’ a program by inserting errors. The test data set is then run on the mutant programs (programs known to contain an error) and the fraction of non-equivalent mutants which are detected by the test data set is calculated. This fraction is taken to be a measure of the likelihood that the test data set will be able to detect real errors in the program to be tested.

The usefulness of this technique depends on a number of factors. There must be a need to determine the adequacy of test data. There must be a method of generating mutant programs: a way of creating ‘errors’ which are representative of real errors which may occur. It must be easy and inexpensive to run the test data on many programs. The results of mutation testing are statistical in nature and do not give absolute assurance of the adequacy of test data. This technique has not been shown to be practical for any but very small programs.

Much theoretical and empirical work has been done on this technique (see reference list in Ref. [9.3]). Mutation scores have also been used to evaluate test data generation techniques. Obviously, this technique must be used in conjunction with other techniques and is most useful as a tool for evaluating a test programme.

9.3.7. Random or statistical testing

Random testing [9.3 pp. 59–60, 9.4] is a black box testing technique generally performed at the system level for the purpose of operational reliability estimation. It consists of randomly choosing a subset of the set of all possible input values and determining the correctness of the outputs for those inputs. To get a statistically valid reliability estimate, the following assumptions should hold:

1. test runs are independent;
2. for each input, the chance of failure is constant;
3. the number of test runs is large;
4. failures are rare;
5. all failures during testing are detected;
6. the distribution of inputs under real operating conditions (the operating profile) is known.

If these assumptions are valid, then a series of failure free test runs can be used to estimate the operational reliability, which is the probability that any given input
encountered under normal operation will result in a failure. (See Ref. [9.4] for a derivation of the theoretical mathematics of this reliability estimate.) Based on the assumptions, random testing can be considered as a sequence of Bernoulli trials and failures per demand can be considered to follow a binomial distribution with the chance of failure on any demand being represented as \( \theta \). If \( n \) random tests are carried out and \( x \) program failures are found, then \( \theta^* \), the \((1 - \alpha)\)100\% upper confidence bound on \( \theta \), represents the largest value such that:

\[
P(x, n) = \sum_{i=0}^{x} \binom{n}{i} \theta^i (1 - \theta)^{n-i} \geq \alpha
\]

(9.1) where \( P(x, n) \) is the probability of at most \( x \) failures in \( n \) runs.

This means that there is a probability of \( 1 - \alpha \) that the calculated bound \( \theta^* > \theta \). For the special case of \( x = 0 \) (no failure in \( n \) runs), this gives:

\[
(1 - \theta)^n \geq \alpha \text{ or } \theta^* = 1 - \alpha^{1/n}
\]

(9.2)

For the situation where \( \theta \) is small and \( n \) is large, the Poisson approximation to the binomial may be employed, giving the result:

\[
(1 - \theta)^n = e^{-n\theta}
\]

(9.3)

Thus if the hypothesis

\[
\theta^* \leq 10^{-k}
\]

is to be true with confidence \((1 - \alpha)\)100\%, then the number of failure free test runs necessary is:

\[
n = 3.0 \times 10^k \text{ if } 1 - \alpha = 95\%
\]

and

\[
n = 4.6 \times 10^k \text{ if } 1 - \alpha = 99\%
\]

Random testing is one of the few ways of getting a numerical estimate for the reliability of the software. The disadvantages are that the assumptions necessary for validity of the estimate can be hard to meet for some kinds of software. It is especially difficult to ensure that test runs are independent if the system cannot be totally reset between runs. It is also very difficult to find out the operating profile for a new system, and the operating profile usually changes over time, often as a result of introducing the new system into the environment. The definition of software reliability is controversial since it is fundamentally different from the definition of
reliability for hardware. There is no general agreement on the method of applying this technique or on its validity.

Some studies have shown that random testing can also be a cost effective way of discovering software faults since the faults which are most likely to cause failures in real operation will be the ones most likely to be detected by random testing [9.5]. On the other hand, since accident situations are infrequent and difficult to predict, this may not be a good way of testing safety system software. The cost effectiveness of this technique will depend on the cost of generating and evaluating the random tests, and on how well the tests find the faults of most concern. For software which must be extremely reliable, this technique will not be appropriate since random testing will be unlikely to produce failures.

9.3.8. Grammar based testing

Grammar based testing [9.3 pp. 60–62] is a testing technique which relies on the system being specified in such a formal manner that a regular grammar for the input and output functionality of the system can be constructed. This has been shown to be possible primarily for systems modelled as finite state machines (FSMs), but is theoretically possible for all computer software.

The technique consists of using a requirements language processor (RLP) to translate the formal requirement specification into a state transition matrix representation of the finite state machine model. The RLP assures that there are no inconsistencies in the requirements and that the FSM is deterministic and all states are reachable. A regular grammar can then be constructed and augmented (manually) to indicate system information and observable outputs. This is passed to a test plan generator (TPG) which generates test scripts (sequences of executable inputs and expected outputs for the system). An automatic test executor (ATE) executes each test script and reports whether or not the system responds in the desired manner.

Such semiautomated testing tools are available for restricted kinds of systems. They have the usual benefits of automated tools: reduced time and costs once the initial investment is recovered. However, there is the need to have the system specification written in a compatible, formal manner. Another potential problem is the length of the test scripts if the grammar contains loops. Some systems have states which, while reachable, only occur after an extremely long series of inputs. Others have an intestably large number of internal states.

9.3.9. Data flow guided testing

Data flow guided testing [9.3 pp. 62–64] is a white box testing technique which is complementary to logic coverage techniques and functional testing. The method is to perform a data flow analysis to determine variable relationships from the detailed design information. This can be done at the individual instruction level, but
is more commonly done for blocks of code (a sequence of instructions executed together and with a single entry and exit).

Three strategies are used in data flow testing:

(1) Block testing: testing each block of code to activate all elementary data contexts for every instruction;

(2) Definition tree: trace each output back to inputs and test all data contexts of the inputs; and

(3) Data space: similar to definition tree, but trace all data items, not just outputs.

In these strategies, a data context is the set of all possible definitions of all input arguments to an instruction or block. A variable definition is the assignment to a variable of a new value.

This testing technique is generally very difficult to apply in practice without automated tools. It is also necessary to have the detailed design information documented in a way which facilitates data flow analysis. With such tools, it is generally possible to perform a static data analysis on the code as well as the design, so there may not be much additional benefit in performing data flow guided testing.

9.3.10. Real time software testing

Real time software is software driving a computer which must interact with real world devices or objects such that the timing of the interaction is dependent primarily on those external devices and objects. Real time software has a number of associated characteristics which make testing more difficult than for non-real-time software [9.3 pp. 66–73].

(1) Generally, real time software is developed and debugged on a computer other than that used for the eventual system. The development computer (the host) usually contains a cross-compiler or cross-assembler so that executable code can be generated which will drive the target computer. Testing of the software on the host computer cannot be considered sufficient because of the differences between the executable codes and the differences in timing behaviour of the computers. In some cases, this problem has been addressed by having the host computer simulate the target computer's instruction set as well as the system environment. However, the additional complexity of such a simulation makes the introduction of new errors almost certain.

(2) System testing should be performed on the software running in the target computer. However, this may be difficult to control if the target computer is embedded in a larger system. In order to perform thorough testing, it is necessary to obtain access to the input and output signals of the target computer. If the larger system into which the target computer is installed does not allow such access, then a 'test harness' must be constructed. Since this target
Computer testing is so important, and since it may be required again after system installation, it is best to design systems containing embedded real time computers and software such that test access is facilitated.

(3) In order to perform valid system level testing on the target computer and software, it is necessary to simulate the operating environment as accurately as possible. This must include the timing behaviour of the operating environment. The simulation must also take into account possible failure modes of the external devices.

(4) Testing of real time behaviour must be based on precise timing and performance specifications. The equipment used for conducting the tests (creating test inputs and monitoring outputs) must have the capability to record the timing of the input and output signals to an accuracy better than that expected of the target system. Performance specifications must be written so that test cases and expected timing of outputs can be generated independently of the software development.

(5) If a real time system contains any memory (as almost all do), then a single test case must consist of a time sequence of inputs (and outputs) with a duration similar to the expected time between system resets, or the expected duration of data in memory. This requirement forces another design guideline: real time systems should be designed with period resets and/or limited duration memory. Without such a guideline, the length of each test case makes proper testing unreasonably time consuming.

Methods of testing real time software are not as well developed as those for other kinds of software. One of the main problems is in establishing a formal language for expressing timing requirements. Many of the existing formal specification languages are deficient in this area and have to rely on incomplete models of time. The software specification is sometimes written by assuming the timing properties of the target computer and operating system. This method facilitates software development and testing, but makes the software less maintainable and portable.

If the system to be tested is to replace an existing system, then it is useful to run the systems in parallel for a trial period. This has the advantages that the existing system can be used as the 'oracle', the new system is tested with real inputs, and the operator has a chance to see the new system in direct comparison with the old. The disadvantage is that such a trial period may give the operator a false sense of security: it is incorrect to assume that the reliability of a system under normal operating conditions is the same as its reliability in accident conditions.

9.3.11. Regression testing

Regression testing is not a different testing technique, but refers to the process of retesting software after a change is made. Theoretically, any change to any part
of the software could introduce or reveal errors in any other part. It is incorrect to assume that since a previous version performed correctly under a particular test, the new version will perform correctly under the same test. For this reason, it is important to save all test cases (and expected results) in such a form that they can be rerun on any subsequent version of the software.

The amount of regression testing done after a change depends on the change control policies, the design of the software and the type of testing. It should not be necessary to rerun module tests on modules not affected by a change. However, it is advisable to rerun all integration tests and system level tests after any change. In particular, if random testing is being used, it should be repeated entirely every time the software is changed. (Note that although the generation/selection of test cases should be on a random basis, the actual test cases must be recorded, so that if an error is detected the test case can be repeated.)

Regression testing can be very expensive and time consuming. It is partly this cost which is leading investigations into more automated testing, and into methods of designing software in the form of clearly separated modules.

9.3.12. Program instrumentation

Program instrumentation is an analysis of software operation which requires modification and execution of the software. The software is run in a modified environment so that more details of the execution process are visible to the tester. Predictions must be made about how the software is supposed to run and these predictions are checked against test runs.

This technique requires either a special programming environment (often referred to as a debugging compiler) and/or the insertion of extra statements into the source code before compilation. In both cases, the purpose is to interrupt the normal execution of the software and to make intermediate values and the values of internal variables externally visible.

Difficulties with this kind of analysis are as follows:

— Detailed knowledge of the design of the software is required in order to make the detailed predictions about how the software is supposed to run for comparison with actual test runs. For this reason, it is most often used by programmers to 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.

— A large amount of data may be generated, making it impractical to document the process completely.
— The process of instrumenting the program may actually change its behaviour; the program may behave differently when the instrumentation is removed, thus negating some of the usefulness of the testing.

— The instrumented program may require more hardware resources than are available on the target processor; if it has to be run on different hardware, again the test results may not be valid.

This form of analysis is widely used by programmers, and most modern compilers include debugging features. There is intense competition amongst compiler vendors to provide the most powerful and user friendly tools for programmers. Naturally, in this competition, the goal is to provide tools which will allow programmers to be most productive; the emphasis is on getting the software working as quickly as possible, not on careful and thorough verification. Therefore, this technique cannot be used as a substitute for complete testing of the final product in the target environment.

9.4. SUMMARY AND FOLLOW-UP

Testing is the process of running software on a computer using carefully selected inputs and comparing the outputs with expected results so that errors can be found. In Section 9, a number of techniques used for testing software are described. Each technique has its own strengths and weaknesses. For safety critical software, it is important to select a set of testing techniques such that the strengths of the individual techniques are complementary: the weaknesses of one technique should be compensated for by the strengths of another. It is also advisable to cover the weaknesses of all testing techniques by using other verification and validation techniques in addition to testing (see Section 8).

There are a few general principles which apply to all forms of testing:

(1) There must be a specification or description of what a piece of software should do so that test cases can be generated and so that expected results can be determined.

(2) Testers should be independent of developers and should have the attitude that they are trying to make the software fail; a test is not successful unless it finds a previously undetected error.

(3) Testing must be completely documented. Test cases, expected results, the testing environment and actual test results must be recorded so that tests can be repeated. An undocumented, unrepeatable test is useless because if a failure is detected, the software is known to be unreliable but the cause of the failure will be almost impossible to trace.

(4) The design of the software has a great effect on the ease and cost of testing. Although designers should be independent of testers, they should be aware of the test programme and should design to facilitate testing.
REFERENCES

10. FIGURES OF MERIT

10.1. ISSUES

In the approach to the demonstration of system fitness for purpose, it has been attempted in recent years to move from one based upon expert opinion to one based upon more quantitative measures of performance and quality. For nuclear reactor protection equipment, the IAEA [10.1] specifically encourages that this be attempted.

Section 10 discusses briefly the possibilities of deriving meaningful quantitative measures (figures of merit), as well as the level of confidence which can be placed in these figures, when software is introduced to form part of the engineered system.

In the context of this report, it is the reliability related aspects of system performance and quality that are of primary importance. The reliability targets of a trip system can be expressed as the number of failures per demand and the spurious trip rate; while those of the control systems that must continuously operate will be mean time between failures. This latter figure can quite readily be turned into a probability of success of the mission.

The reliability argument for installed hardware and analogue equipment is well established. As software is introduced to form part of the engineered system, then, in principle, a numerical approach following the relevant IAEA Safety Fundamentals [10.1] should also be applied to the software. This, however, raises a number of difficulties.

The demonstration of correct and reliable functionality has traditionally been by testing, review and inspection. It is desirable that quantitative measures be derived for the output of these three activities. To a certain extent, this is possible for hardware systems: one makes use of random failure data for hardware components, relying on the assumption that these random failures dominate in the overall failure rate. This means making the generally accepted assumption that for well designed and tested hardware, design faults are rare and can be neglected in the calculation of expected failure rate.

This assumption is not regarded as being valid for software, as the software can fail solely because it contains design faults. These faults cannot be ignored. They are more likely to exist in software because of the unique nature and the increased complexity of most of the software contained in trip and protection systems. Moreover, because of the discrete nature of software, a trivial error (such as of punctuation or addressing) can cause drastic deviations from the intended behaviour. For software, there is no notion equivalent to that of tolerance for analogue circuits, and therefore no notion of ‘small’ error.

Since software errors are design errors only and cannot be neglected, some people argue that software failures are therefore systematic, and that software
reliability figures based on probabilistic arguments are not valid. Others argue that this is not true because the occurrence of a software failure is the result of the activation of a design fault by a given (sequence of) input(s), while the program is in a given state. From the uncertainty of this input process (and from the size of the input space, which precludes exhaustive testing) arises the uncertainty of the software failure process. It is then meaningful to ask what is the probability that a randomly selected input will cause a program to fail, and to consider probabilistic measures of reliability based on the evaluation of the size of the input space, the number of inputs tested and the number of failures observed.

However, as noted in Refs [10.2, 10.3], this probabilistic approach also suffers many handicaps. It is usually impossible to test a system for all possible input data, be it hardware or software. Even in simple cases, complete testing would take hundreds of years. But software has an additional complication. Testing a device which has a continuous behaviour can be based on interpolation. If a function is performed correctly at two not too distant points, it can be assumed correct in all points of the intervening interval. This assumption, often valid for physical devices, cannot be used for software program functions which can have arbitrarily large numbers of discontinuities. The selection of a meaningful set of input test data (test coverage) is therefore much more delicate for software than for hardware.

Another problem [10.4] is that it is impossible to give a statistical justification to a reliability figure which belongs to the ultrareliability region (below $10^{-6}$ failures/h) on the sole basis of experimental or in-service testing. If we wish to ensure a failure rate of $10^{-6}$ h with a 95% confidence level, then we need at least $10^6$ h (114 years) of testing without observing a failure. If quantitative ultrareliability assessments are sometimes possible for hardware systems, it is because these systems can be constructed of redundant subsystems. Under certain conditions, it is then reasonable to assume that the subsystems are independent, in particular free of design flaws, and that their failures are not correlated (common cause failures). When this key assumption of independence is valid, in order to demonstrate that the probability of total failure is less than $10^{-6}$ if the hardware redundancy is, say, equal to three, it is sufficient to demonstrate by testing each subsystem separately that each has a failure probability of less than $10^{-2}$.

Unfortunately, independence can seldom be reasonably assumed between software subsystems. First, all software faults are design faults, which are replicated in redundant copies. Second, even if redundant versions of software are programmed independently by distinct programmer teams, it is not feasible to prove that correlated errors are not present. In fact, several experiments suggest that there will be shared errors [10.5]. These correlations cannot be neglected in the reliability calculations because they are of the same order of magnitude as the reliability that has to be achieved by the whole system ($10^{-6}$ in the foregoing example). But, for this very same reason, they are equally impossible to quantify by testing.
In the face of these problems, many specialists share the view that, at least for the time being, one of the most promising ways to increase the trust that can be placed in software products is to increase trust in the design and construction processes by which the products are obtained. If the reliability of the end product cannot be directly assessed, its trustworthiness can presumably be improved by using a better construction process (see for instance Ref. [10.6]).

A natural consequence of this approach is that the process of assessing and commissioning software would not be based upon simple random failure calculations but also on other quality attributes of the product and of the people and processes that are used to make the product. Unfortunately, many of these quality attributes are not easily quantifiable. Besides, even for those attributes for which measurable properties of the code exist, and which, such as maintainability and legibility, are viewed as having a direct impact on reliability, another problem exists. The relation of the measurable property of the code to the attribute, and also to the reliability and performance of the system, remains undefined and unquantifiable.

Thus, although a large variety of metrics and evaluation tools have been proposed for software, the usefulness of these metrics remains somewhat doubtful and controversial.

10.2. EXPERIENCE

There is some relevant experience available from the development and production of software based systems important to safety. Unfortunately, few valuable data on the use of metrics in this field have reached the public domain. Information presented in this section must therefore be regarded as based upon somewhat limited information.

The size and the complexity of hardware systems have allowed the quantification of their reliability in a relatively direct manner. For reasons already discussed in Section 9, this direct approach is not in general possible for systems containing software: software is discrete and generally too complex to allow simple demonstration by testing and by interpolation on a representative subset of the input conditions. Possible approaches which apply to software are discussed in the following.

10.2.1. Program correctness

The reliability of a software/hardware system depends on the correctness of the system design and on the random failure behaviour of the supporting hardware. If the software could be demonstrated to be fit for its purpose and correct, then, under the assumption of no hardware design flaw, the hardware figures could be used to bound the reliability of the entire system.
The use of formal techniques that rely on mathematical proofs is emerging as a means to prove certain program correctness properties (see Section 5.2.4). However, such methods are currently limited in their scope of application by practical problems and restricted to small programs; see for instance Ref. [10.7]. Besides, specifications and user's needs must also be translated into formal statements, and errors are likely to be introduced during this process.

Thus, at least for the time being, formal proofs do not yet permit the claim that a program will never fail. Nor do they permit figures of merits for correctness to be qualified and ascribed when correctness has only been established for certain functions or parts of the code. The probability of software failures must therefore be bounded in its own right and by other means. Some possible approaches are discussed in the following.

10.2.2. Testing software

The most obvious source of reliability figures is testing. As reported in Section 9, there are many approaches to testing software, with many different purposes (see Section 9.2.1); one of them being to obtain quantitative estimates of the expected reliability of the system.

Exhaustive testing, which means 100% coverage of all possible execution paths of the program, would be the ideal, but is not usually achievable because of the combinatorial explosion of the number of distinct paths in nearly all practical cases of safety system software.

However, as explained in Section 9.3.7, a statistically valid reliability estimate can be obtained if test cases are selected randomly and independently over the set of all possible input values (random testing). Besides, other figures of merit that can be used as part of a subjective process of confidence building are related to the completeness of the test coverage; examples are: percentages of the identified explicitly stated requirements, of the executable statements and of the branches of the control flow graph. Tools exist that examine the source code, derive sets of input data that exercise specific paths through the code, and compute test coverages.

In addition to random testing, much attention has been focused on systematic testing strategies: structurally and functionally based ones. However, it is not easy to derive from the results of these tests an estimation of the system reliability. Further, there is evidence that, if increasing the reliability is the goal rather than finding the maximum number of bugs, then random testing is superior, the reason being the large variation in the rates at which bugs lead to failure [10.8, 10.9].

It is worth recalling that for all forms of dynamic testing it is assumed to be possible to detect an error when it occurs; this may not be the case when, for instance, the correct value of a result may be unknown. Some programs or parts of programs can indeed be 'intestable'. The primary reason for using computer based systems is that they can be used to execute complex trip algorithms; consequently,
it must be expected that cases will arise for which it will be difficult to determine the correct output.

10.2.3. Probabilistic risk assessment

An alternative way of targeting the results of testing is based on the assumption that it is possible to partition the protective software into different portions, and to identify for each portion the system hazards that the software portion is supposed to provide defence against. This can be done, for instance, by a fault tree analysis of the software similar to the one described in Ref. [10.10]. In this facet tree description of software, the failure of a software portion is a bottom event that contributes to the occurrence of the top event. From random test results of the software portions, probabilities can be assigned to the bottom events and risk figures can be derived for the system top event.

However, this is a demanding task for real plant systems, probably pushing probabilistic safety assessment techniques to the limit. Besides, in software fault trees, there can be an interdependence of faults that makes probability assignments difficult [10.11].

10.2.4. Reliability growth models

A classical technique used for deriving figures of merit for software is the reliability growth model. The typical way of applying this approach is to record successive execution times between failures, together with the number of faults removed during that debugging process. These data are then used in statistical models to estimate and predict either the time by which the next failure is to be expected or the number of remaining faults. An overview is given in Refs [10.12, 10.13].

There are a number of practical difficulties with the approach, particularly with respect to its application to high reliability software. First, the models used are based upon statistical assumptions that are not usually valid for software. They assume that the interfailure time $t_i$ between the $i$th and the $(i+1)$th failure is a random variable (that is, a variable whose value depends upon the outcome of a purely random experiment) whose values are identically and independently distributed with a given distribution function $F_i(t)$. Further, they assume that successive $t_i$ are independent random variables: that is, that successive failures are mutually independent events. Another difficulty is the problem of having a statistically meaningful number of data (failures) for the models to work. Since, by definition, failures of ultrareliable software are rare events, the models fail to yield statistically valid results in this region.

For these reasons the predictions of software reliability generated by the various growth models that have been proposed tend both to differ dramatically in quality and to differ between themselves. In an attempt to eliminate these problems, recalibration techniques have been proposed [10.13]. Unfortunately, these
techniques also are based on the very same statistical assumptions as those mentioned earlier. Thereby, the improvements they may bring in certain cases can only be asserted a posteriori, and at any rate they are gained only so long as the reliability requirements are modest.

10.2.5. Software attributes

The process by which a system and its software are produced is reflected in the quality of the end product. As is said in Section 10.1, the quality of the end product depends on properties of the production process as well as on those of the product itself. For both types of properties, several sets of quality criteria have been proposed. Among the first was that of Ref. [10.14]. A similar set/structure of quality criteria (attributes) for the software to be used in an NPP protection system is proposed in Ref. [10.15].

The approach adopted in developing these sets of quality criteria is to take some property of software and then develop a tree like structure of attributes that contributes towards this property. Thus, for example, portability could be considered as having attributes such as device independence and self-containedness. The purpose in constructing such a structure is to allow the properties to be discussed in a structured manner, with the hope also to reduce them, whenever possible to a set of measurable basic attributes. Unfortunately, many of those basic attributes are neither quantifiable nor mutually commensurable. Further, the impact that most of these attributes have on reliability is only indirect.

Some proposed attributes are perhaps more relevant for nuclear applications, for instance the requirement for software simplicity or low complexity. Several sets of software complexity measures exist [10.15–10.18]. However, as discussed below, the practical benefits that can be obtained from these measures are not obvious.

10.2.6. Software metrics

The term ‘software metric’ can refer to two types of quantities that attempt to measure some characteristic or attribute of software. The first type are the product metrics: those quantities that can be derived directly from a piece of software; usually the source or assembly code. These metrics are descriptive of that piece of software and exist in many forms, many of which are quantifiable. They can for example refer to size using: lines of code, number of operators, number of variables; or to complexity: number of intermodule references, depth of nesting of loops or interrupts. The second type are process metrics which include cost and effort quantities. A standardized collection of metrics is given in Ref. [10.19] with a guide on how to apply it in Ref. [10.20]. The values for the metrics are usually derived by means of tools because of the great effort that any manual work normally requires. Many metrics and tools are available, which is perhaps an indication that despite a
large amount of work no single or small set of metrics has been found that is globally useful. Care should be taken when drawing conclusions from the application of metrics because the validity of most metrics has not been substantiated. Many complexity or product metrics are limited to superficial features of programs such as their textual or syntactical characteristics. They take hardly any account, even at the level of a single subprogram, of deeper semantic properties that may result from programming principles based on hierarchical decomposition, abstraction and information hiding.

It has also proved particularly difficult to collect process metrics with any high degree of accuracy. A considerable number of studies have been completed but to date it has not been possible to establish a correlation between these metrics and software failure behaviour. Some work undertaken in the Commission of the European Communities’ ESPRIT REQUEST project and by British Telecom has reported the development of research tools that are able to identify anomalous modules from process metrics collected early during the development cycle. However, the details of the models and how they are applied have not been described. Thus, there remains some way to go before it is possible to derive practical figures of merit. The least controversial metrics might be the measures of software test coverage, and the descriptive complexity metrics when they are used to assist in identifying parts of software deserving restructuring or simplification.

10.2.7. Operational feedback

It is unfortunate that the benefits available from operational feedback are relatively small. This is analogous to the limited benefits available from testing, and arises from the long periods of operation (beyond 10 years) that are required before the results have any significance. If the system does not fail, the demonstrated system reliability increases. However, if the demands on a system are rare, the increase of confidence is very small; and if the system fails, the demonstrated system reliability falls. For failure of a reactor shutdown system, the loss of confidence is dramatic as even a single failure is very significant for a high reliability system.

Thus, during operation, the demands placed on all systems with safety relevance should be monitored. The actual demand profile can then be evaluated and compared with the expected profile; for example, that used to target the testing. Clearly any discrepancy will require action as it may have an impact upon the perceived performance of the system. For example, if the number of demands placed upon the system is greater than expected, then the safety case will need to be re-established and further confidence building measures may be required to improve the level of dependability of the system.
10.3. CURRENT PRACTICES

It is very difficult to recommend practices for deriving figures of merit for software. This will continue to be the case while the debate over quantifying software reliability continues. Although a number of figures of merit can be derived for software and its development process, the key route to quantifying reliability of software in the short term would appear to be through the use of testing.

Since exhaustive testing of software is usually impossible, measures of software test coverage are indispensable. They have to be based upon statistical evidence. In order to draw the greater benefits, testing should concentrate on parts and functions that are identified as intercepting the greatest number of fault scenarios and those fault scenarios that give the greatest reduction in risk. The number of test cases selected for a particular cause of trip should be proportional to the size of the risk associated with the failure of that function. This approach, proportional sampling, minimizes the number of test cases required to show a particular degree of reliability or safety or low risk.

They are possible additional benefits to be gained through the use of metrics; these include the use of descriptive complexity metrics to assist in identifying parts of software that might benefit from restructuring or simplification.

The product and process based metrics collected during the development process might also have some potential use, despite their lack of quantitative relationship to the reliability claim. For example, having established a quality plan and identified standards and coding rules, it is appropriate to collect data on the number of violations of the rules. The verification exercise and its output during the software development may provide some documentary evidence of the level of conformance [10.15]. However, there are currently no rules to convert these data directly into figures of merit relating to software reliability.

10.4. SUMMARY AND FOLLOW-UP

Means exist to derive various figures of merit for software, but their usefulness in demonstrating high software reliability is controversial. The most practical way for deriving useful figures of merit for software is to use testing, applying a mixture of deterministic and probabilistic methods.

Meetings arranged by the IAEA have promoted the exchange of knowledge and experience obtained in the development and justification of computer based systems; for example, those in Saclay [10.21] and Chalk River [10.22]. A common feature of many of the papers presented is the conclusion that no single method can or should be trusted for deriving figures of merit.
REFERENCES


11. MAINTENANCE

11.1. ISSUES

In the past, on many software projects, insufficient priority has been assigned to ensuring that the software produced, and the processes related to it, are amenable to later revision. It was generally felt that if the job was done properly the first time, there should be little requirement to make revisions, and that the cost of performing revisions would be very small compared with the initial design cost.

Experience has shown that over the lifetime of a product, the need for change is inevitable. This can be the result of new requirements or improved technology, or simply to correct newly discovered deficiencies. If 'maintainability', and the methods to be used to perform software maintenance are not addressed during the initial design of the system, the implementation of software changes can be expensive, time consuming and prone to error. If care is not taken, successive changes can lead to a deterioration of the product and divergence from the original design intent.

The principles for maintainability apply to both safety critical and non-safety-critical software. Whether driven by safety or economic requirements, the advantages gained by following them generally outweigh the additional initial effort involved. The activities to be followed during software maintenance must be dictated by the safety category assigned.

11.1.1. Summary of issues

The issues surrounding software maintenance are summarized in Table I. These are discussed in more detail in the following subsections.

TABLE I. SUMMARY OF SOFTWARE MAINTENANCE ISSUES

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Maintainability of software</td>
</tr>
<tr>
<td>2.</td>
<td>Cost effectiveness of software maintenance activities</td>
</tr>
<tr>
<td>3.</td>
<td>Extent of testing required after software maintenance and automation of testing</td>
</tr>
<tr>
<td>4.</td>
<td>Maintainability of test software</td>
</tr>
<tr>
<td>5.</td>
<td>Version control of software and configuration management (compatibility of components of a system) and ensuring the software maintainer does not deviate from the design intent</td>
</tr>
<tr>
<td>6.</td>
<td>Maintenance of user configurable software</td>
</tr>
<tr>
<td>7.</td>
<td>Maintenance of independence for minor activities</td>
</tr>
</tbody>
</table>
11.2. EXPERIENCE

11.2.1. Maintainability of software

Maintainability of software is required for all project outputs (for example, specification of software requirements and description of software design). Maintainability should be built in on the basis of clearly stated maintainability requirements. It is not inherent, nor can it easily be added subsequently.

The requirements for maintainability often conflict with other requirements and usually result in the need for compromise. (For example, full and complete implementation of the principles of information hiding might result in a system that meets all requirements of maintainability, but does not meet specified performance requirements.) It is therefore advisable to apply those principles of the software design which are included for the sake of maintainability.

Due to the burden of maintenance, difficulties in making desirable but unpredicted changes might discourage their implementation.

11.2.2. Cost effectiveness

Software maintenance activities are, by nature, time consuming and expensive. As a result, the customer is sometimes required to live with undesirable, inconvenient software deficiencies during a long and protracted maintenance phase. To complete the maintenance activity requires a return to the start of the life-cycle in order to understand and implement the change throughout the life-cycle, ensuring the changes are completed correctly and ensuring greater ease for subsequent changes. The cost of correcting deficiencies increases following completion of each step in the design life-cycle specific to the changes being implemented. Software maintenance costs can be reduced considerably by building maintainability into the design.

When schedules and budgets show signs of being overrun during the design, coding and testing phases of a project, there is a tendency for the quality of documentation and the addressing of maintenance issues to be compromised. This can lead to an inferior product subsequently being made worse by the burden of maintenance.

11.2.3. Software maintenance testing

For any project, the extent of rework can be variable, from very simple (a set point or constant change) to very complex (major structural changes). Where changes are very simple, the tester must make a decision, based upon experience, analysis and judgement, on how much testing needs to be done to provide a sufficient level of confidence in the product, although in some cases the regulatory body will insist on retesting using the complete data set. If no guidelines exist, it is difficult for the tester to make such a decision objectively.
The preparation of automated, maintainable test cases, for all phases of testing, may add cost to the initial software development programme, but this additional cost is more than recovered in later maintenance phases. More complete and consistent testing, in a shorter period of time, can be achieved by using automated tests and can ensure that the principles of 'regression testing' are followed. Although most tests can be automated, some small amount of manual testing is usually necessary (such as verification of displays). The time taken to resolve failed test cases can be shortened if some amount of diagnostics is incorporated in the test cases.

11.2.4. Maintainability of test software

If care is not taken to design and organize test software for maintainability, the effort required to maintain it can be of the same order of magnitude as that required to maintain the deliverable software. For this reason, the test software should be designed using the same principles of maintainability as the deliverable software.

11.2.5. Version control and configuration management

A tendency exists on the part of the designer to create too many versions of a generic software package to meet specialized needs, rather than to build flexibility into the generic package. The resultant cost of maintaining a multiplicity of versions is excessive.

Software maintenance is often made more complex by the introduction of new requirements at various stages of the development process. This results in the generation of ever more versions that need to be controlled, both formally and non-formally. This additional effort translates into additional cost and increased probability of error, even with the use of automated tools for controlling the software versions.

In multicomputer systems, where operation of software on one machine is dependent upon the operation of different software on another machine, additional complexity in configuration management is added. Since the designer usually has little control over when the software in a given machine is replaced, it must be made clear to the user which system configurations have been tested for use and which have not. This is particularly important where the designer has purposely separated the safety critical logic on one platform and the non-safety-critical software (such as human-machine interface software) on another.

Configuration management and change control require considerable attention to ensure that all necessary changes, and only those changes, are properly incorporated. A Configuration management system should:

1. uniquely identify the official versions of each software item;
(2) identify the versions of each software item which together constitute a specific version of a complete product;
(3) identify the build status of software products in development or delivered and installed;
(4) control simultaneous updating of a given software item by more than one programmer;
(5) provide co-ordination for updating of multiple products (hardware and software) in one or more locations as required;
(6) identify and track each change request from suggestion through release.

Change control procedures should identify, document, review and authorize any changes to the software items under configuration management.

Before sanctioning a change, its validity should be confirmed and its effects on other items examined.

Methods to notify the changes to those concerned and to show traceability between changes and modified parts of software items should be provided.

11.2.6. Maintenance of user configurable software

Note: The extent to which safety critical software is user configurable must be defined with care. It would normally be limited to the ability of the operator to change set points or other constant values.

The customer is generally motivated towards making changes, largely because of failure to understand why the designer requires so much time to deliver revised software, and is confident of being able to make changes much more quickly. Also, there may be extreme pressure to have changes made. However, the customer generally does not have suitable facilities, expertise or independence to design and implement software changes in a cost effective and reliable manner.

The amount of training and effort required for the user to configure the software efficiently and usefully is generally underestimated. This results in a reluctance on the part of the user to make full use of the user configurable features provided, and may slow down the rate at which confidence is gained in the system. Also, instructions given to users to implement their own configurations may be inadequate or not geared to their level of familiarity with the details of the code.

11.2.7. Maintenance of independence for minor activities

The requirement for small changes will occur, and their potential for introducing errors or changes to design principles should not be taken lightly.

During the initial development phase, it is advantageous to establish rules of independence (for example, between functional and software design, between design
and testing, between primary and backup systems). This approach is practicable when the volume of work is high enough to keep all independent groups effectively employed. For later maintenance activities, especially where the changes required are very minor, it is costly to maintain a number of independent groups, each dedicated to a very small portion of work.

11.3. CURRENT PRACTICES

11.3.1. Development for maintainability and cost effectiveness

For any new software development project, cost estimates should include the costs of maintaining the product over its useful life. Design decisions should be based not only on their impact on the initial development cost, but also on their impact on the cost of maintenance. Major effort should be made and discipline should be exercised (in following maintainability guidelines) in establishing requirements in order to avoid compromising good design and software maintainability. Introduction of new requirements late in the development or maintenance life-cycle should be avoided. Such efforts should result in considerable reduction in the complexity and cost of maintenance.

To arrive at a maintainability strategy, a thorough and comprehensive review of the past history of similar applications should be performed, to determine as accurately as possible those areas where change is most likely to be required. The design should facilitate the implementation of changes with a high level of confidence, and rely not on a formal review and verification of a large amount of low level code, but on a more easily reviewable set of high level modular design documents. The ability to map, module by module, between the high level design and the code should be evident. Administrative mechanisms can be established to ensure that quality is maintained in the production of all planned project outputs.

The application of information hiding has proven effective in achieving maintainability. This is a software design technique in which the interface to each module is designed to reveal as little as possible about that module's inner workings. It results in modules which are loosely coupled. Each module should be well defined in its responsibility and function, and the interfaces between modules should be equally well defined and stable.

Care should be taken in the selection of computer hardware, to ensure that its ability to meet performance requirements is not marginal, as using good maintainability techniques may result in degradation of performance of the final product. There may be a tendency to compromise on maintainability requirements, resulting in software which is less maintainable.

The environment in which the requirement and design documents and software are developed and maintained must be taken into consideration when attempting to
achieve a high degree of maintainability. In general, this means judicious selection or development of tools.

If possible customization of designs should be avoided, to the maximum extent possible, so that software can be used on more than one project. This allows costs to be shared between projects and reduces the probability of error for new projects.

11.3.2. Software maintenance testing and test software

Where possible, test procedures should be automated. Test facilities and tools must be selected carefully to improve the capability of the designers and testers to perform all necessary tasks. Test software should be designed for maintainability using, in general, the same rules used for the deliverable software. The configuration control of test software and test cases is as important as that for deliverable software, and the environment for performance of testing must be rigorously defined to ensure consistency of testing. Additional effort is required to produce guidelines to assist the tester in making decisions related to the amount of testing required, especially after very simple software changes. (This topic is covered in more detail in Section 9.)

Automated test cases must be reviewable by future testers, supervisors, assessors and auditors. This is extremely important to permit maintenance to be carried out in the future, when the original designers and testers might not be available. To improve the confidence level of the tester, tests should be made automatically audit-able (whether they have passed or failed).

11.3.3. Version control and configuration management

Version control and configuration management of both software and documentation should be on-line activities for the maintainer.

11.3.4. Maintenance of user configurable software

It must be impressed upon the user that the only software that the user is authorized to modify is 'user configurable' software; other software changes should be made by software designers or maintainers. This will often be difficult for a user to accept unless user needs can be addressed early in a project, such that the anticipated needs to change are covered in user configurable software.

The degree of involvement of customers in software maintenance should be determined after careful consideration of their capabilities, the degree of independence achievable and the cost effectiveness of providing development facilities at the site location. It is paramount that software users and maintainers have a very good appreciation of each other's needs and the environment in which each works. Only then can a co-operative effort be made in deciding what changes need to be made.
and by whom. An effective training programme for users in applying user configurable software must be included in the initial development programme and in any later maintenance phase.

The effort required by the user to maintain the configurable portion of the software must be properly assessed at an early stage. The user should plan his resources accordingly.

11.3.5. Maintenance of independence for minor activities

Independence rules should be evaluated at the start of each maintenance phase, to determine the most practical and effective method of achieving the intent of independence. It is anticipated that, for very small jobs, independence rules might be relaxed considerably from those which would be necessary for major jobs. This must be tempered with the realization that small changes are capable of causing the introduction of major errors. Some independent review should take place for all changes to safety system software; the extent must be defined in the maintenance plan.

11.4. SUMMARY AND FOLLOW-UP

It has become apparent that any software developed for a safety critical application will most probably require maintenance several times during its useful life. If such a reality is not catered for during the initial stages of design and determination of methods, the owner may be forced to contend with long duration and expensive maintenance plans in order to effect changes.

Maintainability should be sought for all the output products from a project and if possible the process by which the product was produced. Maintainability, to be effective, should be considered as part of the design process using techniques which produce units that function mostly autonomously, thereby precluding a change in one part of the program affecting the execution of another unrelated (in theory) part.

The maintenance team are often presented with the onerous task of implementing software changes with out of date or incomplete documentation. To avoid this, good configuration management is essential, preferably by a system that prevents access to the incorrect version. A case study in module testing is presented in Ref. [11.1]. This reference contains a list of some relevant testing principles. It provides further support for automation, auditability and repeatability, while raising other issues such as cost effectiveness and standardization.

A discussion of the IEEE Standard for Software Verification and Validation Plans (ANSI/IEEE Std 1012-19986) and how it may improve the maintainability of software is presented in [11.2].
REFERENCES


12. TOOL SUPPORT

12.1. ISSUES

The use of appropriate tools can significantly influence the dependability of software development and the feasibility of establishing its fitness for purpose. In the last few years the choice of tools supporting relatively well established methods (such as Yourdon design) has increased substantially and, furthermore, many new kinds of tools have been produced, offering support for more recent techniques (such as object oriented design). The first tools supporting the emerging formal specification and development methods have also made their appearance. The activities of the software development process for which tool support is now available include:

1. requirements capture
2. formal specification
3. hazard analysis
4. modelling
5. theorem proving
6. structured design
7. formal design description and refinement
8. static code analysis and formal code verification
9. compilation
10. testing.

Environments are also now available that to some extent facilitate the use of some of these kinds of tools in combination and provide configuration management of the objects they produce.

The system designer, faced with a vast amount of promotional literature on tools, much of it cast in extravagant terms, may find it difficult to make judicious choices. Help in this matter is provided by various comparative surveys of tools, such as the STARTS Guide [12.1], and since some of these are comprehensive, we will not attempt to catalogue here all the kinds of software tools that might be employed for software development. Rather, this section focuses attention on those aspects of tool support for software development that are especially problematic at the present time, with the current shift towards the use of high level languages and the increasing emphasis on program development by formal methods.

12.2. EXPERIENCE

12.2.1. Introduction

The use of appropriate tools is strongly recommended, for several reasons. The most obvious benefit is the mechanization of some phases of program production
(such as the generation of executable code by compilers), which greatly reduces the human effort involved and the risk of introducing errors in the process. Equally important in the development of safety critical programs is the enforcement of rules of construction (for example, the checking of consistency of data flows by informal design tools, type checking and other checks of coherence by formal specification tools, and checking of consistency of information flow by static code analysis tools). Finally, their provision of documentation in standard formats is a major benefit, particularly significant in the development of high dependability software.

The best choice of tools and the most effective ways of employing them are not usually obvious. It is important to distinguish between the choice of software development methods and the choice of tools to implement those methods. The choice of methods should be governed principally by the nature of the application (which determines, for example, whether an initial outline design is best conceived in terms of the input/output functions computed by its components, or in terms of interaction between abstract state machines or objects) and the form of the implementation language (which, if it is high level, can usefully be employed from a much earlier stage in the development process). The choice of a tool is governed by its suitability for the method application and also practical factors such as the ease of incorporating its use in the overall software development process, and evidence of its dependability.

12.2.2. Particular problem areas

At the present time, the dominant issues in selecting and using tools in NPP software applications are the following:

1. There is a general problem in determining the dependability requirements of tools to be employed in producing software for safety related applications, and of establishing whether particular tools meet such requirements.
2. Previous sections of this report have discussed the possibility of using formal methods in software development for NPPs. Would the tool support for formal methods be adequate for these purposes?
3. The advantages of using high level languages for programming have been stressed repeatedly in this report. Whilst their use facilitates design and should lead to improvements in code correctness at source level, it involves reliance on compilers. Are adequate compilers available? Can the correctness of the code they produce be established?
4. Recently application generators have emerged which produce applications software from different kinds of system descriptions. For instance, tools exist to produce control system software from block diagrammatic descriptions of such systems. Are such application software generators acceptable for producing safety related NPP software?

Section 12.3 discusses these questions.
12.3. CURRENT PRACTICES

12.3.1. Dependability of tools

The tools used in the development of safety critical software must have sufficient safety dependability to ensure that they do not jeopardize the safety of the end product. All such tools should be of a very high standard, but the integrity requirement of a particular tool used in software development will depend on its role in achieving system safety, and what other tools or processes would mitigate the consequences of failure. In planning the development of a safety critical NPP system, a hazard analysis and safety risk assessment should be applied to the tools to be employed, identical in principle to the hazard and risk analysis of the system in which the software is to be embedded.

In considering the dependability requirements of tools, it is helpful to consider classes of tools, to which similar general arguments will apply. In particular, Int Def Stan 00-55 [12.2] employs the following useful classification:

Transformation tools such as source code generators and compilers, which transform a text from one level of abstraction to another, usually lower level.

Verification and validation tools, including static code analysers, test coverage monitors, theorem proving assistants and simulators.

Clerical tools used to produce, modify, display and print the objects of the system design and assessment — including editors, printer drivers, print spoolers, terminal drivers and window management software.

Infrastructure tools including operating systems, public tools interfaces and version control tools.

A transformation tool for safety critical software whose output is used without further review should be assigned to the highest dependability level. The requirements for the tool can be reduced by including formal verification (as well as testing) of its output. (Requirements of compilers in particular are discussed further in Section 12.3.3.) For verification and validation tools, the functions performed by a tool should be formally defined and the development of its source code should conform to the requirements of safety critical software development for NPPs; for its compilation, a validated compiler should be employed (see Section 12.3.3), and the executable code should be tested to the extent required for safety critical software. Formal verification of the compiled code can be omitted, on the grounds that the output of the tool will be closely scrutinized. The dependability requirements of clerical tools are relatively low, since their output is checked by independent verification and validation, and the large amount of operational experience increases their assurance. In using infrastructure tools, it is necessary to provide safeguards against
corruption of tools and other objects they deal with; and if the tools perform legitimate modifications of objects (such as by incrementing a version number), downstream checking is required against omissions or inconsistencies that may have been introduced.

12.3.2. Tool support for formal methods

As is mentioned in Section 6, many advocates of formal methods believe that their greatest cost benefits are to be obtained by using them simply to formulate precisely those aspects of functional and other requirements that are amenable to mathematical description. In constructing such specifications it is very helpful to employ a tool that provides support for the simple but nevertheless essential tasks of editing and typesetting (for most specification languages use some of their own non-standard symbolisms), and also for syntax checking, type checking and other rudimentary consistency checks. These facilities are provided at relatively low cost by tools developed specifically to meet these purposes: examples are the Specbox VDM tool [12.3] and the Fuzz editor for Z [12.4]. If one wishes to prove properties of specifications and to discharge proof obligations in refining specifications, it is usually necessary to employ a theorem prover; a few important examples are described in Refs [12.5–12.8], and a fairly extensive catalogue of theorem provers is given in Ref. [12.9]. A proof assistant (or 'short rein' theorem prover) designed specifically for code verification is described in Ref. [12.10].

Concerning the effectiveness and practical utility of such methods and tools, in particular the construction of requirements specifications, there is now substantial evidence of the benefits of expressing these in mathematical notation as far as this is reasonably possible; tools such as SpecBox and Fuzz ease this task considerably.

As for proving properties of formal specifications or discharging proof obligations in refining formal designs, these are difficult tasks for which present day theorem provers are barely adequate. The formal development of a sizeable software system, mechanically discharging all the resulting proof obligations, has not yet been attempted; most practitioners of formal verification are very reserved about their present and likely future achievements, emphasizing that a theorem prover cannot be regarded as a production facility, for automatic certification (or rejection) of program components. They do nevertheless claim that theorem provers provide valuable insight: from Ref. [12.8]:

"Our belief in the correctness of our current proof owes as much to the increased understanding of the problem that we obtained through arguing with the theorem prover as it does to the fact that the theorem prover now accepts our proofs."

It is also argued that, because formal verification is easier if the property being verified is achieved by direct and simple means, the use of formal verification
encourages the programmer to build more elegant and believable programs in the first place. As Ref. [12.11] has shown, these benefits can largely be realized by verification through "rigorous argument", without mechanically checked formal proof. Until theorem provers become substantially more powerful, their application will remain very limited.

12.3.3. Compilers

For reasons stated previously, a program of appreciable size can only be developed satisfactorily by employing a high level programming language. However, the executable version of the program must then be produced by a compiler, itself a large and complex program — significantly more complex than a safety critical NPP program. No formally verified compilers have yet been produced for 'standard' high level languages, and the complexity of these tools makes it impossible to achieve very high levels of assurance for them, even through application of thousands of tests. The question which arises, therefore, is how the required level of correctness of compiled code can possibly be attained.

The first essential is to employ only a compiler that has been developed within a recognized quality control system, such as ISO 9000 [12.12]. The existence of an approved international or national compiler validation certificate would be an important advantage. (Validation of Pascal and Modula-2 compilers for instance is undertaken by some members of ISO, such as the British Standards Institution, and Ada compiler validation is undertaken by agencies approved by the United States Department of Defence Ada Joint Project Office.)

As is explained in Section 7, the basis of such compiler validation is testing, and although it is extensive, it does not guarantee correctness. Even validated compilers have been found to contain serious errors. It is therefore essential to test the compiled form of software very thoroughly, in precisely the form in which it is eventually to be employed. (It will be noted that testing using tools that 'instrument' source code, for instance to monitor test path coverage, is not appropriate here; non-intrusive testing is essential.) Evidence of satisfactory broad based use of a compiler, over a substantial period of time, should also be taken into account in compiler selection.

For safety related applications, it is usually considered important to check the validity of the translation from source to compiled form, at the very least by manual inspection, and if possible by formal verification of the compiled code. This has been attempted in a number of projects, but the task is extremely difficult. The requirement of traceability, from source to object code, is a relatively recent one: previously, by far the dominant considerations in compiler selection have been execution speed and compactness of compiled code, and in striving to meet these demands, compiler manufacturers have produced more and more effective code optimizers, which have made object code less and less intelligible.
With the increased use of Ada (or sublanguages of it) for safety related applications, and the emphasis on traceability from source to compiled code in the Ada 9X Requirements [12.13], Ada compiler manufacturers in particular are making efforts to simplify object code verification. The verification would become somewhat easier if, for instance, certain optimizations could be inhibited on demand, to preserve simplicity of mappings from source data ‘objects’ to processor memory locations, and to prevent some forms of ‘code motion’ in compiled code. Simplicity and uniformity of data management (for example of parameter passing mechanisms) and the documentation of object code by the compiler (for instance to define allocations of variables to registers and their lifetimes) are also essential.

It is of interest how the recent (Draft) DO-178-B [12.14], the international standard employed in the certification of safety critical civil airborne systems, treats the problems of compilation: a compiler is an example of a ‘software process automation tool’ (or in the terminology of Section 12.3.1, a ‘transformation tool’), whose output must be verified, without placing any reliance on the tool. (It might be possible to dispense with some of the output verification if the tool could be ‘qualified’, but this would require its development to meet the DO-178-B standard, a condition that could not be met for a compiler of a sizeable high level language.) The verification of a compiled program to DO-178-B therefore depends very strongly on requirements based testing: it is necessary to produce evidence that this testing has ‘covered’ the requirements. At the same time as this testing is performed, test coverage of the code must be monitored. Then ‘structural coverage analysis’ is performed, to confirm completeness of the coverage. It is to be noted that this analysis may be performed on source code, except where object code is not directly traceable to source code statements; for example, code implementing array bound checks. Where this occurs, additional verification should be performed on the object code (possibly by further tests).

As already observed, this requirement to verify object code is very hard to meet, with existing compilers of large high level languages. The extent to which compiler manufacturers successfully address this problem should be one of the criteria in compiler selection.

Finally, we note that in general, when a high level programming language is used, the object code will contain a run time system of routines that may be invoked by the compiled code in the course of its execution. In safety critical applications; this run time system may be a cut down version of a standard one, containing support for only a subset of the programming language features. However, in that it constitutes a permanent part of the application software, it should be developed to the same standard, and indeed DO-178-B requires this. At least one Ada compiler manufacturer is producing a cut down version of an Ada run time system, to DO-178 standards, with a view to certification by aviation authorities such as the US Federal Aviation Administration, for safety critical applications [12.15].
12.3.4. Application generators

A number of tools are available that generate or configure programs from graphical or textual definitions of their applications. Such tools have existed for many years, a classic example being a parser generator, which produces a parser from a definition of the syntax of a programming language.

Some such tools may be relevant to the production of software for NPPs. For instance, there exist tools for generating control system software from diagrammatic representations of the required system functions. On a larger scale, there also exist configurable process controllers, which support information monitoring and control of distributed systems. The potential advantages in using such tools and configurable systems are clear, but can they be employed with confidence? The dependability requirements of code generators are discussed in Section 12.3.1 (for these are effectively ‘transformation tools’). Configurable systems will be considered in Section 13.

12.4. SUMMARY AND FOLLOW-UP

With regard to the application of formal methods, the rudimentary tool support needed for constructing VDM or Z specifications is available, and is found to be helpful and relatively easy to use. Proof of properties of specifications and the discharge of proof obligations in refining specifications is very difficult, and existing proof tools are barely adequate for the purpose. More powerful proof tools are emerging but the task will always be intellectually demanding.

Increasingly, ‘transformation tools’ of different kinds are being used in the production of software for control functions. Compilers may still be the most common, but application generators that produce code from engineering design descriptions in graphical form are also becoming very popular. In the very long term, perhaps some of these tools will be formally developed. (There is currently much interest in the formal development of compilers; see for instance Ref. [12.16].) In the meantime, the certification of software produced by transformation tools is a major problem. Insofar as such certification must rely very heavily on requirements based testing, there is a great incentive to apply all possible means — including formal methods — to establish precisely what those requirements are.

REFERENCES

[12.2] UNITED KINGDOM MINISTRY OF DEFENCE, DIRECTORATE OF STAN­
DARDIZATION, The Procurement of Safety Critical Software in Defence Equip­
ment, Interim Defence Standard INT DEF STAN MOD 00-55, Part 1: 


(1988).

[12.5] CRAIGEN, D., KROMODIMOELJO, S., MEISELS, I., PASE, B., SAALTINK, 


DOLMAN, W.C., “The formal verification of safety-critical assembly code”, 
Safety of Control Systems, IFAC SAFECOMP 1988, Pergamon Press, Oxford and 


[12.12] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, ISO 9000 
Standard series: Quality Management and Quality Assurance Standards, Part 3: 
Guidelines for the Application of ISO 9001 to the Development, Supply and Main­

[12.13] UNITED STATES DEPARTMENT OF DEFENSE, OFFICE OF UNDER 
SECRETARY OF DEFENSE FOR ACQUISITION, Ada 9X Requirements (Ada 

[12.14] RADIOTECHNICAL COMMISSION FOR AERONAUTICS/EUROPEAN 
ORGANIZATION FOR CIVIL AVIATION EQUIPMENT, Software Considera­
tions in Airborne Systems and Equipment Certification, RTCA Document 
RTCA/DO-178B/ED-12B, RTCA, Washington, DC, EUROCAE, Paris (December 


ESPRIT ProCoS Symp., Ebberup, Denmark, October 1991, ESPRIT, EEC, Brus­
13. USE OF EXISTING SOFTWARE

13.1. ISSUES

Here the term existing software means software developed outside the project in hand. This could consist of library routines, or it could be an operating system, for instance. The use of existing software may be advantageous not only for economic reasons: under certain circumstances it can contribute positively to software assurance. Section 13 outlines the circumstances under which the use of existing software can be beneficial, indicates the extent to which its validation is necessary, and draws attention to some contractual conditions that it may be necessary to establish.

13.2. EXPERIENCE

In developing a system for an NPP, the opportunity may arise to use existing software for some of its components. The following are illustrative examples.

(1) Some software modules serving precisely the same purpose may have been produced in a previous system development for an NPP.

(2) If a high level programming language is used, a run-time system will be required. An apparently appropriate run-time system may be available, for instance from the vendor of the compiler to be employed.

(3) A sizeable system may contain components such as graphical user interfaces, a database, communication subsystems and/or an operating system. Apparently suitable components of all these kinds may be available 'off the shelf'. It is possible to purchase a general purpose process control system that supports monitoring, control and information display, and that can be configured for particular applications.

The use of existing software may show promise of reducing development cost, or of elapsed time to project completion. If it is believed that existing software has been thoroughly validated, and it has been heavily exercised in other systems under very similar operating conditions, this could contribute to software assurance. The possibility of employing existing software should be treated with caution, however, as difficulties may be encountered in assessing the quality of the software and in maintaining it.

Existing software developed outside an NPP project, that is to be used in the final equipment, should meet the requirements of the software standard relevant to the application. If a hazard and risk assessment of the potential application of the existing software indicates that it is not safety critical, and there is extensive evidence
that the existing software has performed satisfactorily in similar circumstances, approval of the software may not require access to its code in source form, or other details relating to its development. However, if the application is safety critical, and the software was not developed in accordance with the relevant standard (or a similar one), it is extremely difficult to establish its adequacy retrospectively. If the system developer does not own the intellectual property rights (or copyright) to the existing software, he or she may not be able to obtain from the supplier all the documentation relevant to the supplier's development and maintenance of the product. Even if the documentation is available, it may be inadequate for retrospective verification and validation. The system developer may then be tempted to try to create the documentation required for the existing software by 'reverse engineering' its code. However, as mentioned in Section 7.3, this is an arduous process, which is only likely to succeed for well written programs of modest size; it should never be relied upon as the principal means of precisely establishing the intended behaviour of a program.

A related concern is the adequacy of the specification of the interface between a module of existing software and its environment (in our case, the interface between the existing software and custom built system software for an NPP). In the particular case of a run time system, one also requires details of its interface to its compiler, especially if the run time system is to be 'cut down' (as is commonly done for safety critical applications).

If the use of a general purpose process control system is envisaged, it is necessary to consider the validation of:

1. the components of the system to be employed; and
2. either the configuration process — which involves checking both the description of the required system and the procedures applied to this description — or the configured product.

Finally, it is necessary to bear in mind the fact that if the existing software is a widely used commercial product, the system developer is not likely to be able to obtain a guarantee of vendor support, over a long period of time, for a particular version. Neither will he or she be able to exercise much influence over the form of future versions, or over the quality of their production. Unless long term support is contractually guaranteed, or can be provided by the system developer, the use of a commercial product must involve financial and other risk. At any rate, existing software should be considered in the first place for tasks where no changes due to maintenance are expected.

13.3. CURRENT PRACTICES

It must be ensured that existing software developed outside a software development project for an NPP which is to be used in the final equipment conforms to the
requirements of the applicable software standard. A record of all existing software items employed, and evidence that they conform to the requirements of the standard, should be included in the software documentation.

If a potential application of an item of existing software is shown by hazard analysis and risk assessment to be safety critical, then:

(1) The producer of the equipment should carry out its verification, if the item was not produced as required by the applicable software standard, by creating all its specification and design documentation and applying all verification, testing and analysis procedures, as mandated for new software by the standard;

(2) The developer of the software must ensure that either (a) he or she possesses a copy of all documentation needed to maintain the item and is technically able and legally empowered to do so, or (b) the long term maintenance of the item to the appropriate software standard is guaranteed by a contract with the developer of the software, this contract giving the developer the right to establish that the maintenance is carried out satisfactorily.
14. PERSONNEL QUALIFICATION AND TRAINING

14.1. ISSUES

The quality and safety of the software in NPPs depend strongly on the people who specify, design, verify and maintain it. The outstanding issues in this area can be summarized as two questions:

(1) What qualifications should be required for software engineers working on all aspects of safety systems?
(2) How can these qualifications be achieved by software engineers, controlled by the development organization, and checked by the regulator?

These questions reflect the wider state of affairs in all areas of software development: there is no agreement on what qualifications are required for people developing software, and no widely accepted standards for software professionals.

14.2. EXPERIENCE

Software is a relatively young field, and has changed rapidly with the development of faster, more powerful computers, the development of new algorithms and the development and evolution of programming languages. People who have many years of experience were trained on systems which are now obsolete. People who have been trained on the latest systems have little working experience. There have been at least three major shifts in the way we think about programming (flow charts/sequential processing; structured programming; object oriented programming) and there has been insufficient time for any single way of thinking to become predominant.

It is often the role of universities and professional societies to specify the qualifications which define a given profession. In North America, some steps in this direction have been taken by the Association for Computing Machinery (ACM) and the US Institute of Electrical and Electronic Engineers (IEEE) Computer Society. However, the universities have not yet managed to co-ordinate their degree requirements even to the extent of deciding to which department software belongs: some regard it as a branch of mathematics, others regard it as a branch of engineering, still others regard it as a science. In the United Kingdom, the British Computer Society Safety Critical Systems Group has drafted guidelines for a course syllabus for safety critical systems developers [14.1]

Another complicating factor is that the rapidly changing nature of the field quickly makes qualifications obsolete. It is very difficult to find appropriately qualified instructors to teach techniques which have been recently introduced or which
are still under development. It is also very difficult to get the agreement of current practitioners on a set of qualifications: they all have different qualifications and no one wants to exclude themselves. These difficulties are compounded by the fact that the proper use of such methods as formal specification and verification requires mastery, rather than passing knowledge, and mastery requires time and experience.

In industry, software 'engineers' gain their education and experience in a haphazard fashion. It is not uncommon for current practitioners to have been primarily self-taught in the areas of software development and verification with an ad hoc set of short training courses supplementing their knowledge of particular programming languages and methods. Computer science or computer engineering graduates are often poorly prepared for software development work, having been taught the latest research interests of their university professors, but not the basic software engineering methods [14.2]. Technical school graduates are taught the quirks and foibles of a variety of programming languages and computers, but not the general principles which can be applied to all software and computers.

14.3. CURRENT PRACTICES

Section 14.3 summarizes some efforts that have been made towards defining qualifications for software developers. Only with such agreement can the appropriate training courses be put in place and evaluation of current practitioners be done.

14.3.1. Three aspects of qualification

Reference [14.3] suggests a three part set of qualifications for software engineers:

(1) **Knowledge:** software engineers should be expected to be knowledgeable in the following areas, with emphasis varying depending on specific roles within a project (knowledge is verifiable through training course results, educational records or records of specific experience):
   - discrete mathematics
   - formal specification and design methods
   - static and dynamic verification methods
   - microelectronics
   - quality assurance
   - social and safety issues.

(2) **Skills:** software engineers should be expected to be able to apply their knowledge in the areas outlined above. Skills are verifiable through records of specific experience and products which have been developed using those skills.
(3) *Attitudes*: software engineers should be expected to accept responsibility for the software and documentation which they produce. They should understand the possible implications of software failures and documentation errors and should be concerned about potential dangers to fellow employees and the public. Attitudes are extremely difficult to verify.

### 14.3.2. Knowledge and skills

The knowledge required of software engineers is acquired through an educational programme, work experience and on the job training. Some work has been done to try to define and reach agreement on what should be the content of an educational programme (a curriculum) which would produce qualified software engineers. However, it is unlikely that international agreement will be possible; to date it has not even been possible to achieve national agreement.

If an agreement can be reached concerning the knowledge required of software engineers, there are a number of ways that agreement could be applied. It could be used as a basis for accreditation of university or college programmes; it could be used as a basis for membership in a professional organization; or it could be used to test individuals. The success of such usage will depend on whether the industry sees any benefit in hiring individuals who have acquired this knowledge.

One example of a set of knowledge requirements has been specified by the ACM/IEEE-CS Joint Curriculum Task Force [14.4] in North America. Another has been proposed in Ref. [14.2].

In the United Kingdom, the Ministry of Defence Draft Interim Defence Standard 00-55 [14.5] identified the need for everyone involved with safety critical software to be "trained, experienced and qualified". The recommendations for training were of a preliminary nature and were mainly given to identify the need for standardized training courses and a qualification program. People developing safety critical software systems should be "Registered Safety Critical Software Specialists (RSCSS)" and should have a "broad and deep understanding of all aspects of Safety Critical Software, and the methods, tools and techniques that are applied". The RSCSS training was estimated to require about 16 weeks over a period of one to two years. The draft standard did not include a description of the required curriculum of such a training programme, but stated that further work is needed in conjunction with learned societies, academia and industry.

When the standard was officially issued, the section on Quality of Staff had been reduced to the following sentence:

"The Design Authority shall demonstrate to the MOD safety assurance authority and the MOD(PE) PM that the seniority, authority, qualifications and experience of the staff to be employed on the project are satisfactory for the tasks assigned to them." (Ref. [14.6], Section 13.)
The second part of this standard, entitled ‘Guidance’, states ‘No guidance required’ regarding the quality of staff.

In the nuclear industry, no standards exist specifically concerning the qualifications of software engineers. However, it is generally felt that not only should software engineers be knowledgeable concerning software development and verification; they should also be knowledgeable about the application of the software. Furthermore, it is important that software engineers have a broader knowledge of the context in which the software will be placed. They should know something about the computer hardware, the sensors and the actuators which will make up the system, and the way in which those other components are designed, built and qualified.

Knowledge assessment cannot be entirely based on educational qualifications. Practising software engineers should be assessed on the knowledge they have gained through experience. Records should be kept of work experience as well as training. This is often done in the form of a résumé (curriculum vitae), but should be formalized to indicate the knowledge and skills gained.

14.3.3. Attitudes

The attitudes required of software engineers working on safety systems are no different from the attitudes required of any other engineers. The fundamental propositions concerning safety culture have been described in a report by the International Nuclear Safety Advisory Group for the IAEA [14.7]. Although this report focuses on the operation of NPPs,

"The safety of the plant also depends critically on those who previously designed, constructed and commissioned it." (Ref. [14.7], Introduction.)

In the third section of this report, the universal features of safety culture are described:

"In all types of activities, for organizations and for individuals at all levels, attention to safety involves many elements:

— Individual awareness of the importance of safety.

— Knowledge and competence, conferred by training and instruction of personnel and by their self-education.

— Commitment, requiring demonstration at senior management level of the high priority of safety and adoption by individuals of the common goal of safety.

— Motivation, through leadership, the setting of objectives and systems of rewards and sanctions, and through individuals’ self-generated attitudes.

— Supervision, including audit and review practices, with readiness to respond to individuals’ questioning attitudes.

— Responsibility, through formal assignment and description of duties and their understanding by individuals.” (Ref. [14.7], Section 3.)
From this quote, the attitudes of individuals cannot be assessed separately from the safety culture of the organization. However, the report does include sample questions which could be used as a basis for an audit or assessment of these attitudes.

Different countries have set different requirements concerning the responsibility for software products. However, it is generally acknowledged that software documents have a similar status to engineering drawings: they should be signed by people who acknowledge their responsibility for the correctness and safety of the product described.

14.3.4. Teamwork

It is unusual and inadvisable for the development of safety critical software to be the responsibility of a single person. It is therefore important for software engineers to know how to work as a team. The organization and management of software development teams is discussed in Section 4.

14.4. SUMMARY AND FOLLOW-UP

Section 14 summarizes some current efforts in defining the qualifications required of personnel developing safety critical software. This area needs to be discussed and debated at greater length until agreement is reached. It is important to have a method for evaluating software development personnel since ultimately the safety of a system rests on the qualifications, skills and trustworthiness of the people who built it.

REFERENCES


15. BALANCE IN SOFTWARE BASED SYSTEMS

15.1. ISSUES

Engineering projects inherently involve decisions that reflect tradeoffs among items such as cost, schedules, reliability, functionality, complexity, reviewability and maintainability. Maintaining a proper balance among such items is particularly difficult with software projects because of the evolving nature of the discipline, the wide availability of competing methods intended to achieve a variety of related aims, and the difficulty in assessing whether the expected contribution of a particular method is actually realized. Regulatory licensing interaction can also affect this balance, particularly if the ground rules are not well established at the start of the project.

The extent to which a proper balance is achieved determines whether or not the project will be a success and whether or not a safe system will result.

It is not possible to define a generalized, prescriptive approach for choosing an optimum balance because this will, in fact, be dictated by national requirements, the technical aspects of the particular project and the particular skill sets of the people involved.

15.1.1. Aspects of balance

There are three aspects of balance corresponding to three distinct phases of the system design process which need to be considered:

(1) The resources that are needed to produce software that is safe to the required degree of confidence will determine the balance between the use of software as opposed to alternative technologies in implementing specified system functions;

(2) The distribution of the resources allocated for a particular system, in the proper balance among system, hardware and software safety features, and verification processes will facilitate the achievement of optimum system safety;

(3) The distribution of resources allocated for software implemented functions among alternative software design and verification processes in proper balance will contribute to the achievement of optimum safety.

The first aspect recognizes the desire to minimize cost while achieving the required safety for a system to be implemented. If the methods for achieving or ensuring safety using software are too expensive, then another technology (such as hardware logic) will be chosen. Sometimes unavailability of a suitable alternative technology makes a software based system the only choice (for example, where substantial complexity is involved). In this case, if the methods for achieving the
necessary safety using software are very expensive, the viability of the entire project (or industry) in which the safety system in question is being used can be jeopardized. For NPP safety systems, an acceptable level of safety determined by current nuclear safety objectives must be achieved no matter what technology is employed. If software is chosen, resources needed to achieve this safety level must be available. However, with software, the difficulty is in determining when satisfactory assurance of the required safety has been achieved.

The second aspect recognizes the fact that even after it has been decided to implement a software based system, critical decisions must be made about how much of the safety assurance will be achieved by design decisions about the system features (for example, amount of redundancy, separation, isolation, diversity, channelization) and by the astute partitioning of functionality between hardware and software.

For the implementation of the portion of the system assigned to software, the third aspect considers the available alternatives for achieving the required degree of safety assurance. Here decisions must be made both on the software process and on software safety features to be used.

The foregoing three aspects correspond to three distinct phases in the system design process. At each of these phases, an appropriate balance must be struck among several factors if a software based system is to be implemented safely and cost effectively.

In addition to creating an appropriate balance within each of the foregoing aspects, an appropriate balance should be obtained between the three aspects.

15.2. EXPERIENCE

Section 15.2 presents relevant experience on the issue of balance gained in NPP software projects important to safety. This experience deals with the balance at the three levels/aspects identified above:

(1) balance in the selection of technologies (that is, balance between software and alternative technologies);
(2) balance among system, hardware and software features;
(3) allocation among available software production methods.

15.2.1. Balance in technology selection

At the highest level of consideration, all the issues of balance for implementation of a particular system are basically limited to tradeoffs between cost and safety. In the absence of safety considerations, the system engineering objective is to implement the required functionality in the manner that results in the lowest expected lifetime costs, all factors (including obscure and indirect ones) being considered.
How to strike a proper balance between cost and functionality is a decision that is made by comparing the cost involved in making further functionality increments against the benefits provided by the improvements, usually measured in economic terms. Similarly, technology choices would be based on the ratio of expected costs to benefits, selecting the technology with the lowest ratio.

The introduction of safety considerations into the technology selection makes the process less straightforward and more subjective, because it is difficult to express safety increments in terms of costs or benefits. The decision is especially difficult for designers of systems that perform safety related functions, since the tendency would be to want to quantify the safety benefits in economic terms to facilitate making good engineering decisions.

Despite the stringent reliability requirements on safety systems, it is generally recognized that, beyond a certain point, there is a diminishing return in safety: disproportionately high costs will be incurred for little increase in system reliability. Even if the costs were ignored, additional safety functions may not provide any real increase in system reliability when the associated added complexity is considered. A major objective at this level must be to minimize system and software complexity. (If sufficient simplification is infeasible because of the functionality required, strategies can be employed such as isolating the complex portions or providing a simplified backup to mitigate the inherent risks in implementing complex systems.)

15.2.2. Balance between hardware, software and system aspects

At the second level, the issue concerns balancing the efforts in the software area towards achieving safety with corresponding efforts in the hardware and systems implementation areas.

Even when it has been determined that a software implementation would be the best solution, appropriate attention must be paid to the qualification of the hardware environment in which the software executes. Of particular importance is the need to ensure that the hardware satisfies any assumptions made by software designers.

Historically, for designs of systems important to safety, techniques such as redundancy, separation and diversity have been used to prevent failures due to postulated events, component failures, design errors, or maintenance errors [15.1]. (The concept of implementing an integrated, multifaceted strategy for avoiding or reacting to safety related faults is known as defence in depth. In a software based system, use is made of self-checks to convert potentially unsafe hardware failures into a safe state where possible.)

Safety systems in general have stringent reliability targets which can only be achieved using redundancy, with a 2-out-of-3 or 2-out-of-4 voting as the most common approaches.
Ideally, the safety systems should meet their required safety reliability targets without relying excessively on self-checks. However, the addition of software self-checks to improve system safety reliability may be a necessary tradeoff as it attempts to improve the safety reliability at the expense of increased false negative actions, complexity and greater difficulty in verification and testing. Furthermore, software self-checks, if relied on too heavily, may in fact reduce the overall system safety reliability since:

1. hardware failures which are detectable by software self-checks tend to be random and localized in nature, affecting only one channel at a time; and
2. software errors, which may be introduced inadvertently by complex self-checks, tend to affect all channels at the same time.

15.2.3. Balance between software design approaches

At the third level, the issue concerns selecting the best balance from the software design and verification methods available that would achieve the required safety confidence in the most cost effective manner. There are basically two approaches [15.2]:

1. fault elimination: eliminate the possibility of a safety fault occurring in the software; and
2. fault coverage: identify all possible sources of safety faults in the software and provide suitable remedial action.

In the first of these approaches, the software would have to be error free to the required degree. This could be achieved by the use of the most rigorous software production and verification processes, and would also require faultless operation and maintenance of the system. The second of these approaches involves providing extra logic or facilities so that any fault occurring would be detected and reacted to in a predetermined, controlled manner. In practice, neither of these two approaches is by itself completely feasible. Error free software is as yet unachievable, and complete fault coverage would require perfect knowledge of system failure modes and significant additional software size and complexity to be able to react to each failure mode. Therefore, a practical approach has been to select an appropriate contribution from both of these approaches.

For fault elimination (avoidance or removal), there are four main methods:

1. Good software engineering practices (such as use of a clean room): making it inherently difficult to introduce faults into the design; part of this is the need to obtain the best possible people with the required knowledge, experience and skills for all of the jobs to be done.
2. Testing (systematic and statistically valid random testing): overlapping testing against clear, explicit and comprehensive specification documents.
(3) Formal verification: use of methods such as static analysis or program function table analysis for detecting obscure faults.

(4) Hazards analysis: systematic re-examination of the design (by fault trees, for example) from the perspective aimed at identifying weaknesses which could lead to unsafe situations.

*Fault coverage* can also make good use of the output from a hazards analysis. It addresses not only faults that arise in the design being produced, but also faults that occur due to external failures in the system being controlled. Effective fault coverage for software based systems is best achieved by an appropriate mix of the following complementary methods:

(1) Functional redundancy: diversity, coupled with a selection mechanism;

(2) Specific design features: watchdog timer protection; baton or thread checks; reasonableness checks; pattern checks; time/rate checks (these must be balanced against the extra complexity that they add);

(3) Appropriately rigorous design methodology: complete, precise specifications; stepwise refinement; systematic design decomposition; encapsulation: separation and isolation of safety critical system portions; all processes defined and audited; verification at each design step; use of appropriate, qualified tools;

(4) Periodic in-service testing throughout the service life of the system to detect system degradation (this is especially important for detecting random faults in poised systems, but it has not been shown to be of significant value for detecting design faults unless the test cases vary over time).

The application of suitable fault coverage techniques provides increased confidence that the system will perform appropriately no matter what conditions are presented to it.

Designers must recognize that the nuclear industry is highly regulated. Hence each decision must bear in mind regulatory requirements. It is therefore particularly important to provide reviewable documentation that permits the regulator to understand the process that was followed as well as the results of each step in the process. Recent international experience indicates that regulators are becoming increasingly rigorous in the area of software. Licensees will be required to use the best available software engineering methods. It is obviously desirable to reach agreement with the regulator at the outset of the project.

Unfortunately, what often happens during the course of a project, or from project to project, is that additional process layers are added to the development process, as mandated by the regulator, without deleting the old process layers. (This is sometimes referred to as 'ratcheting'.) Experience in Canada has been that regulators can require licensees to adopt new methods during the course of a project. In the USA, licensees of the Eagle 21 system were made aware of additional requirements near the start of the project, but probably did not anticipate them. Future plants
will probably be subject to further requirements, but this situation is not unique to software based systems. In the United Kingdom, licensing of the Sizewell B plant has resulted in the application of new methods [15.3].

In addition to increasing costs and causing schedule delays, the additional process steps can also detract from safety by diluting the effort over many development activities.

15.3. CURRENT PRACTICES

To be able to make sound 'balance' decisions, it is vital to define at the outset of the project or the system design a systematic, comprehensive set of objectives and corresponding design principles and policies. Also, to minimize the chance of licensing requirements not being fulfilled, the licensee and the regulator should agree up front on the design principles and policies.

The design principles and policies should be referenced when making all subsequent design and resourcing decisions throughout the system development and maintenance life-cycle. This ensures that the system design evolves consistently and that the end result will satisfy the requisite characteristics established for the particular application.

Objective success criteria should also be established consistent with the design principles and policies. The success criteria are then used to determine whether or not the process used and the completed product satisfy the balanced set of attributes defined as requirements on the system. Past practice has used success criteria based on weighted software assessment attributes (such as cost, schedule, reliability, functionality, complexity, reviewability, maintainability and self-checking) evaluated by means of checklists to measure the degree to which each attribute has been satisfied. The weighting used makes implicit assumptions about the marginal contribution of each assessment attribute. The method suffers somewhat in that it is subjective in nature, and there are not yet sufficient data to establish the correlation between assessment scores and safe and reliable software. Therefore, it would be useful to collect data on this and other methods which could form the basis for a more objective success criteria.

One obvious software attribute that should not be overlooked is 'maturity'. In building up the system, choices must be made about how much of it to build and how much to buy. If the purchased portions are mature and can reliably be integrated with the custom built portions, then this can be a cost effective way of achieving a safe system. However, care should be taken to ensure that each portion has attributes (such as reliability, maintainability, portability and extensibility) matching the rest of the system.

Additionally, when a mature system must undergo a partial change, the extent to which the benefits of 'maturity' are jeopardized must be assessed. Generally,
regression testing beyond the proximity of the change must be carried out unless it can be clearly demonstrated that the effect of the change can be localized.

The maturity of a software production process is also an important consideration in achieving an appropriate balance. There is a tradeoff between a process which has more potential than another but which may not yet be mature enough to confirm whether the full scope of the potential can be realized in practice (for example, formal methods versus the recognized limitations of testing). In this case, it may be prudent to use effectively the portion of the process for which benefits have been confirmed (for example, use of a formal specification).

In addition to assessing the quality attributes of bought and built software products, the relative contributions from competing development processes (such as the many types of testing or verification processes) must be assessed for their effectiveness and to determine when a specific process can be eliminated because it has been superseded by the coverage provided by another adopted process. Similarly, the marginal utility of additional increments of effort applied to a particular process must be assessed, mindfully of the limited resources, the need for coverage, the design objectives, and the relative worth of other processes. For example, in using a particular formal verification method, it may be possible to cover 99% of the constructs in the software and for an extra quantum of effort and cost, it may be possible only to extend this coverage to 99.3%. By applying the same quantum of effort and cost using an alternate verification method, better confidence may result that 100% coverage of all software constructs can be achieved.

Currently, as an overall strategy until one particular technique comes to the fore, the most prudent approach will continue to be to select an integrated, synergistic system/software/hardware approach. For the software portion of the effort, use of a suite of complementary and supporting processes in high marginal utility proportions is advisable [15.4].

The following practices have proven effective and efficient in producing safe software based systems:

(1) The use of an external watchdog timer is highly effective in detecting most types of computer failures and is simple to implement.

(2) Other simple and yet effective checks that are recommended include range checks of variables, consistency checks between redundant inputs, and checksums of critical variables.

(3) Software self-checks should be used sparingly; in particular, if a software self-check is so complex that it requires scheduling to execute, it should be scrutinized to ensure the benefits really outweigh the added complexity.

(4) Software fail-safe strategies should be adopted to achieve ‘inherent’ fail-safeness as part of overall software structure instead of adding complexity whose whole purpose is to check for failures.
Different strategies should be used for safety critical protective (poised) systems and safety critical control (continuously used) systems since control systems generally require more immediate and robust reaction to any detected faults.

For protective systems, the use of external monitors to detect failures may be more effective than complex built-in self-checks since, in the former case, the size and complexity of the safety critical software is minimized and isolated from non-critical software.

The IEC 880 standard contains material useful in selecting an appropriate balance [15.5].

15.4. SUMMARY AND FOLLOW-UP

There are many competing methods available aimed at achieving software which will operate safely. It is doubtful that any single approach (software quality assurance/testing/formal methods/diversity/reliability quantification) can efficiently eliminate the probability of unsafe errors to the required degree. While it may be intellectually satisfying to drive one method to completion, it is more efficient to apply more than one method synergistically.

No one single technique or method of software engineering is sufficient by itself, and applying one particular facet (such as testing) results in diminishing marginal returns insofar as the efficient elimination of faults is concerned. Accordingly, adoption of a balanced approach is necessary to optimize globally the effort and resources applied in using the effective individual techniques. For example, the use of formal methods by themselves is not sufficient to acquire adequate confidence in safety critical software, but, when used in the right proportion with testing and other methods (such as hazards analysis), will lead to effective and efficient software verification and the elimination of safety related faults.

The method of how to select the proper suite of complementary and supporting methods is not well established. Using intuition to select between available processes or to decide on the relative effort to be applied to alternative processes is not good because it is too easy for a single characteristic of one process to sway its evaluated utility negatively or positively. However, some techniques are available for comparing processes that have several relevant characteristics that must be used in the decision making. One technique has been applied by Ontario Hydro and AECL-CANDU for making decisions about the best system configurations for NPP safety system applications. The technique (known as the multiobjective decision criteria technique [15.6]) was designed to make decisions relative to nuclear safety when many nonlinear factors must be considered. It is particularly suitable for addressing the issue of diminishing marginal returns for different system attributes insofar as safety is
concerned. It is too early to report on the degree of success of the method, but it appears to be useful.

Safe computer based systems have been implemented for systems important to safety in NPPs, but not as effectively or efficiently as they could have been. This has been primarily because of the difficulty in assessing the contributions of different design approaches, design features and implementation processes. Further work needs to be done to quantify the relative and marginal contributions of various approaches and to establish what balance is required in order to satisfy particular safety objectives. The overall goal remains to produce designs that ensure the most cost effective coverage of all safety needs.

In summary, the resolution of the balance issue is necessary if the software based systems are to be considered feasible for use from both an economic and safety perspective.

REFERENCES


Annex I

EXISTING STANDARDS AND STANDARDIZATION WORK RELATING TO SOFTWARE IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS

Existing standards and standardization work are listed in Table I-I. This listing is intended to provide an overview only and is not comprehensive.

### TABLE I-I. EXISTING STANDARDS AND STANDARDIZATION WORK RELATING TO SOFTWARE IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS

<table>
<thead>
<tr>
<th>Standards and Guides</th>
<th>Scope</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nuclear computer system standards</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>International</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEC 880: Software for Computers in the Safety Systems of Nuclear Power Stations, International Electrotechnical Commission (1986)</td>
<td>Highly reliable software for computers to be used in the safety systems of NPPs, including safety actuation systems</td>
<td>Being updated</td>
</tr>
<tr>
<td>IECSC45A/WG-A3: Software for Computers Important to Safety for Nuclear Power Plants, IEC-880 Supplement, International Electrotechnical Commission (Draft, 1991)</td>
<td>Takes into account new developments in formal methods, tools and use of pre-existing software; updating IEC 643 on classification; scope will be expanded to cover safety related systems</td>
<td></td>
</tr>
<tr>
<td><strong>Canada</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standards and Guides</td>
<td>Scope</td>
<td>Remark</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>Japan</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>USA</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Other computer safety or critical applications</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>International</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEC SC65A/WG10: Functional Safety of Programmable Electronic Systems: Generic Aspects, International Electrotechnical Commission</td>
<td>Aspects of safety related systems using computers; provides a classification of software reliability levels; for the specification, design and validation of system requirements</td>
<td>Provides a basis for the software standard; available as IEC(65A) 123</td>
</tr>
<tr>
<td>Standards and Guides</td>
<td>Scope</td>
<td>Remark</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>EWICS TC7: Dependability of Critical Computer Systems, Vols 1, 2, 3, Elsevier (1988, 1989, 1990)</strong></td>
<td>All aspects of safety related computers, including design safety and reliability assessment, quality assurance, maintenance and modification</td>
<td>Guidance documents useful as a basis for other standards to avoid duplication of work</td>
</tr>
<tr>
<td><strong>Germany</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DIN VDE 0801/01.90: Grundsätze für Rechner mit Sicherheitsaufgaben (Principles for Computers in Safety Related Systems) (1990)</strong></td>
<td>Treats both hardware and software aspects</td>
<td>Draft standard</td>
</tr>
<tr>
<td><strong>United Kingdom</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MOD 00-55: The Procurement of Safety Critical Software in Defence Equipment, Part 1: Requirements, Part 2: Guidance, Ministry of Defence (1991)</strong></td>
<td>All aspects of safety critical software; deals with development, verification and validation using formal mathematical methods in conjunction with dynamic testing and static path analysis</td>
<td>Emphasizes formal methods; the first standard in which formal methods are mandated</td>
</tr>
<tr>
<td><strong>MOD 00-56: Requirements for the Analysis of Safety Critical Hazards for Computer Based Systems, for Ministry of Defence (1989)</strong></td>
<td>Safety analysis and risk analysis to be done in order to be able to specify the integrity class of a computer system</td>
<td>Extension of MIL-STD-882B the purposes of the United Kingdom</td>
</tr>
<tr>
<td><strong>SafeIT, Vol. 1: The Safety of Programmable Electronic Systems; Vol. 2: A Framework for Safety Standards, Department of Trade and Industry, United Kingdom (1990)</strong></td>
<td>Proposing both a framework for standards on computer based systems in safety related applications and a way to derive them</td>
<td>Relevant for nuclear protection systems</td>
</tr>
<tr>
<td><strong>Software engineering</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ESA BSSC 1A: Software Engineering Management Standard; ESA BSSC 1B: Software Configuration Management Standard, European Space Agency (1984)</strong></td>
<td>Software in satellites and ground stations</td>
<td>Fairly general</td>
</tr>
<tr>
<td>Standards and Guides</td>
<td>Scope</td>
<td>Remark</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
</tbody>
</table>
Annex II

APPLICATION OF SOFTWARE BASED SYSTEMS
IMPORTANT TO SAFETY IN NUCLEAR POWER PLANTS:
EXPERIENCE OF MEMBER STATES

Experience of Member States with software based systems important to safety in NPPs is listed in Table II-I. The Table does not provide comprehensive information but only examples of experience as presented by participants in the Technical Committee Meeting of June 1991.
<table>
<thead>
<tr>
<th>Plant</th>
<th>Utility</th>
<th>System/function</th>
<th>In-service year</th>
<th>System developer</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belgium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doel 1/2</td>
<td>Electrabel</td>
<td>Protection system against external accidents:</td>
<td>1991</td>
<td>ACEC</td>
<td>Graphical programming tool COGITO for PLCs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Signal processing and protection</td>
<td></td>
<td>Siemens</td>
<td>Assembler language</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— Actuation functions</td>
<td></td>
<td></td>
<td>STEP-M on Siemens TELEPERM-ME technology</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Canada</td>
<td>Safety critical applications</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Darlington</td>
<td>Ontario Hydro</td>
<td>Shutdown Systems 1 and 2 (SDS1 and SDS2); fully computerized reactor protection system</td>
<td>1990</td>
<td>AECL and Ontario Hydro</td>
<td>SDS1: Fortran and Assembler on a GA16 computer; SDS2: Pascal and Assembler on an LSI-11 computer; All software formally verified</td>
</tr>
<tr>
<td>Point Lepreau and Gentilly 2</td>
<td>New Brunswick Power Corporation and Hydro Quebec</td>
<td>Shutdown Systems 1 and 2 (SDS1 and SDS2); programmable digital comparators (PDCs) used to implement trip decision logic</td>
<td>1983</td>
<td>AECL</td>
<td>Ladder logic programming</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>informal review</td>
</tr>
</tbody>
</table>
### Safety related applications

<table>
<thead>
<tr>
<th>Location</th>
<th>Operator</th>
<th>System/Function Description</th>
<th>Year</th>
<th>Manufacturer/Platform</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pickering A/B</td>
<td>Ontario Hydro</td>
<td>Digital Control Computers (DCCX and DCCY); process control of: reactor, boiler pressure, boiler feedwater and level, turbine generator, primary system pressure</td>
<td>1971/1983</td>
<td>AECL</td>
<td>Mostly Assembler on Varian computer; verification and validation by testing</td>
</tr>
<tr>
<td>Bruce A/B</td>
<td>Ontario Hydro</td>
<td></td>
<td>1977/1985</td>
<td>An INTEL 8085 microprocessor; ladder logic programming</td>
<td></td>
</tr>
<tr>
<td>Darlington</td>
<td>Ontario Hydro</td>
<td></td>
<td>1989</td>
<td>OH180 is a custom built computer based on an INTEL 8085 microprocessor; ladder logic programming</td>
<td></td>
</tr>
<tr>
<td>Point Lepreau</td>
<td>New Brunswick Power Corporation</td>
<td></td>
<td>1983</td>
<td>Ontario Hydro OH180 is a custom built computer based on an INTEL 8085 microprocessor; ladder logic programming</td>
<td></td>
</tr>
<tr>
<td>Gentilly 2</td>
<td>Hydro Quebec</td>
<td></td>
<td>1983</td>
<td>ABB Atom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Distributed process control using OH180 microcomputer</strong></td>
<td>1989</td>
<td>ABB Automation MASTERPIECE 200</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Imatran Voima Oy (IVO) Application Implemented into the ABB: Stromberg distributed process information systems</td>
<td>(Not safety classified, but required and validated by the safety authority)</td>
</tr>
<tr>
<td>Darlington</td>
<td>Ontario Hydro</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finland</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Safety critical applications**

None

**Finland**

**Safety related applications**

<table>
<thead>
<tr>
<th>Location</th>
<th>Operator</th>
<th>System/Function Description</th>
<th>Year</th>
<th>Manufacturer/Platform</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Olkiluoto 1 and 2</td>
<td>Teollisuuden Voima Oy (TVO)</td>
<td>Low power level control (Feedwater control)</td>
<td>1991 (Unit 1)</td>
<td>ABB Atom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1991 (Unit 2)</td>
<td>ABB Automation MASTERPIECE 200</td>
<td></td>
</tr>
<tr>
<td>Leviisa 1 and 2</td>
<td>Imatran Voima Oy (IVO)</td>
<td>Critical safety function monitoring system</td>
<td>1990</td>
<td>IVO: Application ABB: Stromberg power system platform</td>
<td></td>
</tr>
<tr>
<td>Plant</td>
<td>Utility</td>
<td>System/function</td>
<td>In-service year</td>
<td>System developer</td>
<td>Remarks</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>----------------------------------------</td>
<td>-----------------</td>
<td>------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td><strong>France</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Safety critical applications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1300 MW(e) PWR EdF</td>
<td>(20 units)</td>
<td>SPIN: Reactor protection system</td>
<td>1985 onwards</td>
<td>Merlin Gerin</td>
<td>8 bit microprocessor based system; assembler language</td>
</tr>
<tr>
<td>1400 MW(e) PWR EdF</td>
<td>(Site: Chooz B, 2 units)</td>
<td>SPIN: Reactor protection system</td>
<td>Under construction</td>
<td>Merlin Gerin</td>
<td>16 bit microprocessor based system; C language ('Safety' Kernel) and Assembler language</td>
</tr>
<tr>
<td></td>
<td><strong>Safety related applications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1300 MW(e) PWR EdF</td>
<td>(20 units)</td>
<td>Digital control automation (Controbloc)</td>
<td>1985 onwards</td>
<td>CEGELEC</td>
<td>Dedicated logic of integrated circuits for the ALU; microprocessor based communication system; specific language for application software</td>
</tr>
<tr>
<td>1400 MW(e) PWR EdF</td>
<td>(site: Chooz B, 2 units)</td>
<td>Digital control automation (Contronic E)</td>
<td>Under construction</td>
<td>Hartmann and Braun</td>
<td>16 bit microprocessor based system; PL/M86 language and Assembler language</td>
</tr>
<tr>
<td>Location</td>
<td>Operator</td>
<td>System Description</td>
<td>Year</td>
<td>Provider</td>
<td>Language</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>------</td>
<td>-------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Mühlenheim Kärlich</td>
<td>RWE</td>
<td>DRB (Digital Computer Block) evaluates core data: flux, pressure, coolant flow</td>
<td>1987</td>
<td>BBC</td>
<td>Assembler language</td>
</tr>
<tr>
<td>Würgassen Isar</td>
<td>Preußen Electra</td>
<td>TK 250, Digital Neutron Flux channels, processing power range, intermediate range, startup range</td>
<td>1988</td>
<td>Hartmann and Braun</td>
<td></td>
</tr>
<tr>
<td>Philippsburg-1</td>
<td>Preußen Electra</td>
<td>Bayern Werk, Baden Werk</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Safety related applications**

<table>
<thead>
<tr>
<th>Location</th>
<th>Operator</th>
<th>System Description</th>
<th>Year</th>
<th>Provider</th>
<th>Language</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>All German BWRs</td>
<td>HEW, RWE, Bayern Werk</td>
<td>Control rod motion computer: monitoring control rods and controlling their movements</td>
<td>1975</td>
<td>AEG, KWU</td>
<td>Assembler</td>
<td>Help for plant startup, plant operation; Assembler</td>
</tr>
<tr>
<td>Konvoi PWRs</td>
<td>HEW, RWE, Bayern Werk</td>
<td>Prisca, supervision of core data</td>
<td>1985</td>
<td>Siemens/KWU</td>
<td>FORTRAN, Assembler languages</td>
<td></td>
</tr>
<tr>
<td>Gundremmingen</td>
<td>Bayern Werk, RWE</td>
<td>Maintaining margin for minimum critical power ratio; monitoring and control of power distribution</td>
<td>1985</td>
<td>Siemens/KWU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plant</td>
<td>Utility</td>
<td>System/function</td>
<td>In-service year</td>
<td>System developer</td>
<td>Remarks</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------</td>
<td>------------------------------------------------------</td>
<td>-----------------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>India</strong></td>
<td><strong>Safety critical applications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KAPP-1</td>
<td>Nuclear Power Corporation</td>
<td>Programmable digital comparator system</td>
<td>1992</td>
<td>BARC</td>
<td>Protective system based on Intel 8085, Assembler language; information system based on Intel 8086, C language</td>
<td></td>
</tr>
<tr>
<td>NAPP-1/2 and KAPP-1</td>
<td>Nuclear Power Corporation</td>
<td>Control for on-power refuelling system</td>
<td>1989/91, 1992</td>
<td>BARC</td>
<td>Distributed computer configuration; control computers (8085) use Assembler for system tasks, process control language (PCL) for application logic</td>
<td></td>
</tr>
<tr>
<td>NAPP-1/2 and KAPP-1</td>
<td>Nuclear Power Corporation</td>
<td>Reactor regulating system</td>
<td>1989/91, 1992</td>
<td>BARC</td>
<td>Three independent channels each based on Intel 8085, Assembler language</td>
<td></td>
</tr>
<tr>
<td>NAPP-2 and KAPP-1</td>
<td>Nuclear Power Corporation</td>
<td>Channel temperature monitoring system</td>
<td>1991, 1992</td>
<td>BARC</td>
<td>Based on Intel 8086, C language; protection hardware external to computer</td>
<td></td>
</tr>
</tbody>
</table>
### Safety critical applications

<table>
<thead>
<tr>
<th>Plant</th>
<th>Operator</th>
<th>System Description</th>
<th>Year</th>
<th>Vendor</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dungeness B</td>
<td>Nuclear Electric</td>
<td>Single channel tripping system (reactor protection): computerized systems for monitoring fuel channel temperature and tripping</td>
<td>1991</td>
<td>AEA Technology</td>
<td>Assembly: ASM86 and M6809; Software subjected to MALPAS compliance analysis and executable code compared with source using NE toolset.</td>
</tr>
<tr>
<td>Sizewell B PWR</td>
<td>Nuclear Electric</td>
<td>Fully computerized primary protection system (reactor protection)</td>
<td>Due for operation in 1994</td>
<td>Westinghouse</td>
<td>Languages: PL/M86, PL/M51, ASM 86, ASM 51 for Intel computers; Software subject to MALPAS compliance analysis and executable code compared with source using NE toolset.</td>
</tr>
</tbody>
</table>

### Safety related applications

a Many plants in the United Kingdom use computerized control systems; for example, a typical advanced gas cooled reactor incorporates the following control loops, each based on a dedicated microprocessor: (a) bulk gas outlet temperature, controlled by regulating control rods; (b) reactor gas mass flow, controlled from the load controller via inlet guide valve controllers; and (c) boiler feed flow, controlled by adjustment of the turbine feed pump and feed regulating valves.
<table>
<thead>
<tr>
<th>Plant</th>
<th>Utility</th>
<th>System/function</th>
<th>In-service year</th>
<th>System developer</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>United States of America</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Safety critical applications</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haddem Neck</td>
<td>CT Yankee Atomic Power Co.</td>
<td>Reactor trip system</td>
<td>1990</td>
<td>Foxboro</td>
<td>8086 microprocessor, PLM 86 language, 25 000 lines of code</td>
</tr>
<tr>
<td>Zion</td>
<td>Commonwealth Edison Co.</td>
<td>Signal conditioning and bistable</td>
<td>1992</td>
<td>Westinghouse</td>
<td>Intel 286/12 single board computer, Intel 8286 microprocessor, PLM 86 language</td>
</tr>
<tr>
<td>D.C. Cook</td>
<td>Indiana/Michigan Power Co.</td>
<td>Signal conditioning and bistable</td>
<td>1993</td>
<td>Foxboro</td>
<td>8086 microprocessor, PLM 86 language, 25 000 lines of code</td>
</tr>
<tr>
<td>South Texas Project</td>
<td>Houston Lighting and Power Co.</td>
<td>Qualified safety parameter display system</td>
<td>1987</td>
<td>Westinghouse</td>
<td>Intel 8840A single board computer, Intel 8186 microprocessor, PLM 86 language</td>
</tr>
<tr>
<td>ANO2</td>
<td>Entergy Operations</td>
<td>Core protection calculator upgrade</td>
<td>1990</td>
<td>Combustion Engineering</td>
<td>Microprocessor based system, C/C++ language</td>
</tr>
</tbody>
</table>
### Safety related applications

<table>
<thead>
<tr>
<th>Location</th>
<th>Company</th>
<th>System</th>
<th>Year</th>
<th>Manufacturer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turkey Point</td>
<td>Florida Power and Light Co.</td>
<td>Emergency diesel generator sequencer</td>
<td>1991</td>
<td>Allen Bradley</td>
<td>PLC-2/16, Intel 80C188 and Intel 80C51 microprocessors, commercial grade ladder logic programming, PLC-2 utilities software and PLC-2 program development and documentation software</td>
</tr>
<tr>
<td>Brunswick</td>
<td>Carolina Power and Light Co.</td>
<td>Leak detection</td>
<td>1993</td>
<td>General Electric</td>
<td>NUMAC (GE proprietary)</td>
</tr>
<tr>
<td>Palo Verde</td>
<td>Arizona Public Service Co.</td>
<td>Diverse auxiliary feedwater</td>
<td>1992</td>
<td>Modicon</td>
<td>Relay ladder logic programming actuation system; coded using Gray-Soft GS984 commercial software package, DOS/IBM operating system</td>
</tr>
<tr>
<td>Haddem Neck</td>
<td>CT Yankee Atomic Power Co.</td>
<td>Auxiliary feedwater</td>
<td>1992</td>
<td>Woodward</td>
<td>Xilog 16 bit microprocessor, ladder logic programming and C language; menu oriented editor (MOE) is used to configure software</td>
</tr>
</tbody>
</table>
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACM</td>
<td>Association for Computing Machinery, New York</td>
</tr>
<tr>
<td>AECB</td>
<td>Atomic Energy Control Board of Canada</td>
</tr>
<tr>
<td>AECL</td>
<td>Atomic Energy of Canada Ltd.</td>
</tr>
<tr>
<td>ATE</td>
<td>Automatic test executor</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
</tr>
<tr>
<td>CICS</td>
<td>Computer Software Configuration Item</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite state machine</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrochemical Commission</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronic Engineers</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>MOD</td>
<td>United Kingdom Ministry of Defence</td>
</tr>
<tr>
<td>NPP</td>
<td>Nuclear power plant</td>
</tr>
<tr>
<td>PDL</td>
<td>Program design language</td>
</tr>
<tr>
<td>PSA</td>
<td>Probabilistic safety assessment</td>
</tr>
<tr>
<td>PWR</td>
<td>Pressurized water reactor</td>
</tr>
<tr>
<td>RLP</td>
<td>Requirements language processor</td>
</tr>
<tr>
<td>SCS</td>
<td>Safety critical software</td>
</tr>
<tr>
<td>SRS</td>
<td>Software requirements specification</td>
</tr>
<tr>
<td>RSCSS</td>
<td>Registered Safety Critical Software Specialist</td>
</tr>
<tr>
<td>STARTS</td>
<td>Software tools for application to large real-time systems</td>
</tr>
<tr>
<td>TPG</td>
<td>Test plan generator</td>
</tr>
<tr>
<td>VDM</td>
<td>Vienna Development Methodology (for software)</td>
</tr>
<tr>
<td>Contributors</td>
<td>Institutions</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Archinoff, G.H.</td>
<td>Ontario Hydro, Canada</td>
</tr>
<tr>
<td>Asmis, G.J.</td>
<td>Atomic Energy Control Board, Canada</td>
</tr>
<tr>
<td>Baldwin, J.</td>
<td>AEA Technology, United Kingdom</td>
</tr>
<tr>
<td>Ball, R.A.</td>
<td>Nuclear Installations Inspectorate, United Kingdom</td>
</tr>
<tr>
<td>Bloomfield, R.</td>
<td>Adelard, United Kingdom</td>
</tr>
<tr>
<td>Carré, B.</td>
<td>Program Validation Ltd, United Kingdom</td>
</tr>
<tr>
<td>Chandra, U.</td>
<td>Bhabha Atomic Research Centre, India</td>
</tr>
<tr>
<td>Cocher, L.</td>
<td>NPP Bohunice, Czechoslovakia</td>
</tr>
<tr>
<td>Cortazar, S.</td>
<td>Comisión Federal de Electricidad, Mexico</td>
</tr>
<tr>
<td>Courtois, P.-J.</td>
<td>AIB-Vinçotte Nucléaire, Belgium</td>
</tr>
<tr>
<td>Crane, R.</td>
<td>AECL-CANDU, Canada</td>
</tr>
<tr>
<td>Ehrenberger, W.</td>
<td>Fachhochschule Fulda, Germany</td>
</tr>
<tr>
<td>Gould, D.</td>
<td>Previse, Inc., Canada</td>
</tr>
<tr>
<td>Haapanen, J.P.</td>
<td>Technical Research Centre of Finland, Finland</td>
</tr>
<tr>
<td>Henry, J.-Y.</td>
<td>Commissariat à l’énergie atomique, France</td>
</tr>
<tr>
<td>Hohendorf, R.J.</td>
<td>Ontario Hydro, Canada</td>
</tr>
<tr>
<td>Hughes, G.</td>
<td>Nuclear Electric plc, United Kingdom</td>
</tr>
<tr>
<td>Ichiyen, N.</td>
<td>AECL-CANDU, Canada</td>
</tr>
<tr>
<td>Joannou, P.</td>
<td>Ontario Hydro, Canada</td>
</tr>
<tr>
<td>Krs, P.</td>
<td>Czechoslovak Atomic Energy Commission, Czechoslovakia</td>
</tr>
<tr>
<td>Leveson, N.G.</td>
<td>University of California, Irvine, United States of America</td>
</tr>
</tbody>
</table>
Manninen, T.T. Imatran Voima Oy, Finland
Mauck, J.L. United States Nuclear Regulatory Commission, United States of America
Pacala, D. NPP Research Institute (VUJE), Czechoslovakia
Pachner, J. IAEA
(Scientific secretary)
Parnas, D.L. McMaster University, Canada
Rata, J.-M. Electricité de France, France
Rodriquez-Alvarez, J.M. Comisión Federal de Electricidad, Mexico
Shildt, G. Technical University of Vienna, Austria
Solman, N. Elektroprojekt, Croatia
Taylor, R.P. Atomic Energy Control Board, Canada
Varga, C. Nuclear Service, Hungary
Vidovszky, I. Atomic Energy Research Institute, Hungary
Viola, M. Ontario Hydro, Canada
Wall, D.N. AEA Technology, United Kingdom
Wassyg, A. Alan Wassyg Consulting, Canada

Advisory Group Meeting
Vienna, Austria: 22–26 April 1991

Technical Committee Meeting
Vienna, Austria: 1–5 June 1992

Consultants Meetings
Ottawa, Canada: 17–18 June 1991; Vienna, Austria: 28 October–1 November 1991;
Ottawa, Canada: 14–18 September 1992

168
LIST OF RELATED IAEA PUBLICATIONS

1979  *Safety Functions and Component Classification for BWR, PWR and PTR: A Safety Guide* (Safety Series No. 50-SG-D1)


1984  *Safety-Related Instrumentation and Control Systems for Nuclear Power Plants* (Safety Series No. 50-SG-D8)


1988  *Code on the Safety of Nuclear Power Plants: Design* (Safety Series No. 50-C-D (Rev. 1))


1993  *Safety Fundamentals: The Safety of Nuclear Installations* (Safety Series No. 110)
HOW TO ORDER IAEA PUBLICATIONS

☆ ☆ In the United States of America and Canada, the exclusive sales agent for IAEA publications, to whom all orders and inquiries should be addressed, is:
UNIPUB, 4611-F Assembly Drive, Lanham, MD 20706-4391, USA

☆ ☆ 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.

ARGENTINA Comisión Nacional de Energía Atómica, Avenida del Libertador 8250, RA-1429 Buenos Aires
AUSTRALIA Hunter Publications, 58A Gipps Street, Collingwood, Victoria 3066
BELGIUM Service Courtier UNESCO, 202, Avenue du Roi, B-1060 Brussels
CHILE Comisión Chilena de Energía Nuclear, Venta de Publicaciones, Amunategui 95, Casilla 188-D, Santiago
CHINA IAEA Publications in Chinese: China Nuclear Energy Industry Corporation, Translation Section, P.O. Box 2103, Beijing
IAEA Publications other than in Chinese: China National Publications Import & Export Corporation, Deutsche Abteilung, P.O. Box 88, Beijing
FRANCE Office International de Documentation et Librairie, 48, rue Gay-Lussac, F-75240 Paris Cedex 05
GERMANY UNO-Verlag, Vertriebs- und Verlags GmbH, Dag Hammarskjöld-Haus, Poppelsdorfer Allee 85, D-53115 Bonn
HUNGARY Librotrade Ltd., Book Import, P.O. Box 126, H-1656 Budapest
INDIA Oxford Book and Stationery Co., Scindia House, New Delhi-110 001
ISRAEL YOZMOT Literature Ltd., P.O. Box 56055, IL-61560 Tel Aviv
ITALY Libreria Scientifica Dott. Lucio di Biasio “AEIOU”, Via Coronelli 6, l-20146 Milan
JAPAN Maruzen Company, Ltd, P.O. Box 5050, 100-31 Tokyo International
NETHERLANDS Martinus Nijhoff International, P.O. Box 269, NL-2501 AX The Hague
SWEDEN Swets and Zeitlinger b.v., P.O. Box 830, NL-2610 SZ Lisse
PAKISTAN Mirza Book Agency, 65, Shahrah Quaid-e-Azam, P.O. Box 729, Lahore 3
POLAND Ars Polona, Foreign Trade Enterprise, Krakowskie Przedmiescie 7, PL-00-068 Warsaw
ROMANIA Ilexim, P.O. Box 136-137, Bucharest
RUSSIAN FEDERATION Mezdunarodnaya Kniga, Sovinskniaga-EA, Dimitrova 39, SU-113 095 Moscow
SLOVAK REPUBLIC Alfa Publishers, Hurbanovo námestie 3, SK-815 89 Bratislava
SOUTH AFRICA Van Schaik Bookstore (Pty) Ltd, P.O. Box 724, Pretoria 0001
SPAIN Díaz de Santos, Lagasca 95, E-28006 Madrid
SWEDEN Fritzes Information Centre, S-106 47 Stockholm
UNITED KINGDOM HMSO, Publications Centre, Agency Section, 51 Nine Elms Lane, London SW8 5DR
YUGOSLAVIA Jugoslovenska Knjiga, Terazije 27, P.O. Box 36, YU-11001 Belgrade

☆ ☆ Orders (except for customers in Canada and the USA) and requests for information may also be addressed directly to:

Sales and Promotion Unit
International Atomic Energy Agency
Wagramerstrasse 5, P.O. Box 100, A-1400 Vienna, Austria