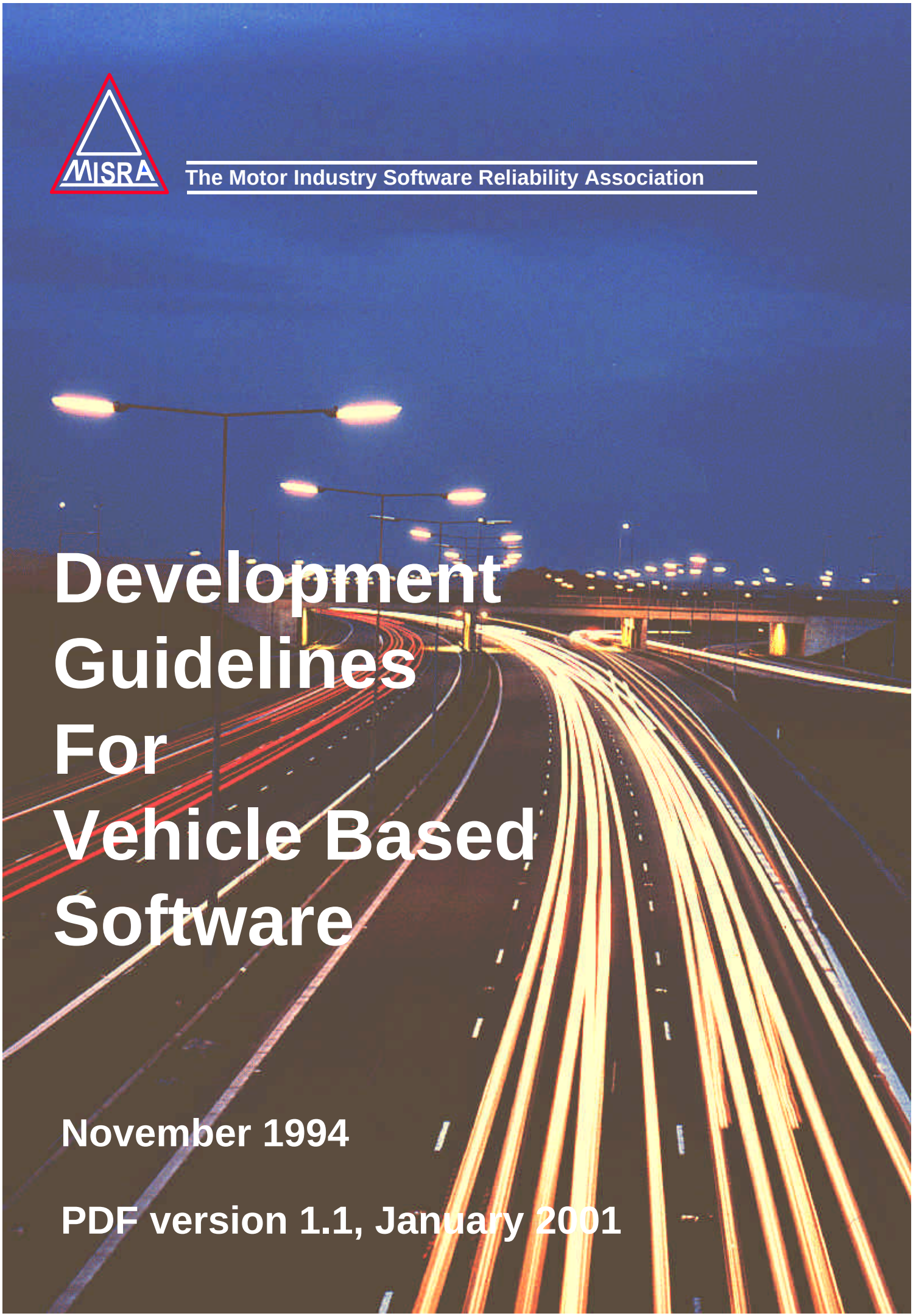

MISRA
The Motor Industry Software Reliability Association
Development Guidelines For Vehicle Based Software
November 1994
PDF version 1.1, January 2001

PDF version 1.1 **© MIRA, 2001**

This electronic version of the MISRA Guidelines is issued in accordance with the license conditions on the MISRA website. Its use is permitted by individuals only, and it may not be placed on company intranets or similar services without prior written permission.

MISRA gives no guarantees about the accuracy of the information contained in this PDF version of the Guidelines, and the published paper document should be taken as authoritative.

Information is available from the MISRA web site on how to obtain printed copies of the document.

First published November 1994
by The Motor Industry Research Association
Watling Street
Nuneaton
Warwickshire
CV10 0TU

http://www.misra.org.uk

ISBN 0 9524156 0 7

British Library Cataloguing in Publication Data.

A catalogue record for this book is available from the British Library.

The Motor Industry Software Reliability Association

Development Guidelines For Vehicle Based Software

November 1994

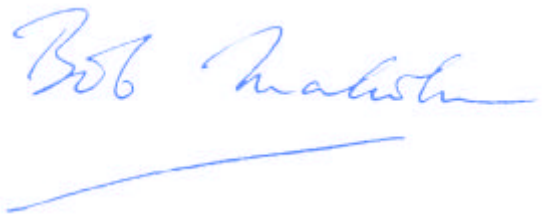
Foreword

As coordinator of the Safety Critical Systems Research Programme, supported by the Department of Trade and Industry and the Engineering and Physical Sciences Research Council, I am pleased to support the publication of these guidelines.

In this programme we have been concerned to ensure that the results of research should not be buried in learned papers, but promulgated in ways which will affect practice in industry. We have therefore sought the involvement of user organizations so that the work based on an understanding of their real industrial needs, and so that the results will be credible to their peers. The MISRA Study, with its combination of vehicle and equipment manufacturers, has admirably realized that ambition.

The voluntary nature of the guidelines is important. They were produced voluntarily, for the benefit of both the industry and the public. Their adoption will also be voluntary. This has encouraged the MISRA consortium to develop guidelines which offer genuine benefits for their users, rather than burdensome restrictions.

The renowned cost-consciousness of the automotive industry might be thought by some to diminish its contribution to technological development. On the contrary: while this industry is properly cautious in the interests of safety, the fiercely competitive market and the public exposure of any problems ensures that the commercial value of new technology is thoroughly evaluated. This sector is a hard proving-ground for technology. So, while many of the issues on which the MISRA consortium has concentrated are specific to vehicles, other sectors might do well to see what they might glean from the trends here.



Bob Malcolm
DTI-EPSRC Safety Critical systems

Foreword (continued)

The motorist gains many benefits, including enhanced safety features, from advances in vehicle electronics.

The development of software is a specialized and often complex area where much relies on an effective approach by those directly involved. I welcome MISRA's initiative and efforts in developing these safety related guidelines and advice under a joint DTI, EPSRC and industry funded programme. The guidelines reflect a responsible and serious industry attitude to safety issues.

Malcolm Fendick BSc CEng FIMechE
Chief Mechanical Engineer
Department of Transport

The use of electronic systems in vehicles has increased significantly over recent years and will continue to increase. Much time and resources are being committed by vehicle manufacturers to deal with the compatibility of such systems in particular the problem of interference from external sources.

The vehicle owner expects his vehicle to be reliable and safe, and this includes the electronic systems. Electronic systems are dependent on the software provided by the manufacturer. The greater complexity of such systems is increasing the need to maintain software quality and reliability. This has resulted in the need for a unified approach to software design. It is therefore pleasing that an independent group has produced these guidelines on vehicle-based software which will be of benefit to the Motor Industry and also more importantly the motorist.

K B Barnes
Head of Engineering
SMMT

Acknowledgements

The MISRA consortium would like to thank the following organizations for their support of the study that produced these Guidelines:

AB Automotive Electronics Ltd	(CM)
AP Borg & Beck	(CM)
Delco Electronics	(CM)
Department of Trade and Industry	
Department of Transport	
Ford Motor Company Ltd	(CM)
Jaguar Cars Ltd	(CM)
Lotus Engineering	(CM)
Lucas Electronics	(CM)
Rolls-Royce and Associates Ltd	(C)
Rover Group Ltd	(CM)
The Centre for Software Engineering Ltd	(C)
The Motor Industry Research Association	(PM, C)
The Society of Motor Manufacturers and Traders Ltd	
The University of Leeds	(C)

Key

CM	controlling member of the MISRA consortium
PM	project manager
C	consultant

The MISRA consortium would like to thank the following individuals for their contributions to these Guidelines:

Alex Abbot	Vivien Hamilton	Frank O'Neill
Klaus Allen	Nick Heatley	Dave Perry
Neil Andrewartha	Keith Hobley	Mike Radford
Nigel Barrett	Peter Jesty	Roger Rivett
Terry Beadman	Edward Jones	Steve de la Salle
Nick Bennett	Ian Kendall	Chris Shakespeare
Colin Bowles	Keith Longmore	Heather Storey
Tom Buckley	Paul Manford	Ian Stothers
John Comfort	Chris Marshall	Lloyd Thomas
Nick Dunn	Josh McCallin	David Ward
Paul Edwards	Tom Monk	Brian Whitfield
Simon Farrall	Tony Moon	Anthony Wood
John Fox	Dave Newman	Marco Zasiura

Contents

Contents (continued)

1. Introduction

1.1 Statement of mission and objectives

1.1.1 The purpose of these Guidelines is to provide assistance to the automotive industry in the creation and application within a vehicle system of safe, reliable software.

1.1.2 There has been much recent growth in the quantity and complexity of electronic controls on motor vehicles. The greater the complexity, the harder it is to maintain the software quality and reliability the customer has come to expect. A unified approach to software development, using agreed techniques, is desirable across the automotive industry. This will enable the driver or owner to have continued confidence in complex electronic systems.

1.2 Benefits to the end customer

1.2.1 The public exposure to software in applications with safety implications is increasing. It may be through vehicles that the majority of the public will encounter such software. Therefore, it is vital that such software is both correct and perceived to be correct.

1.2.2 The record of the automotive industry in regard to software is good. The application of these Guidelines will maintain the industry's confidence in the quality of the software used in its products as the complexity increases.

1.3 The MISRA consortium

1.3.1 The MISRA consortium was formed in response to the UK Safety Critical Systems Research Programme, supported by the Department of Trade and Industry and the Engineering and Physical Sciences Research Council, and consisted of eight controlling members, four consultants and a project manager (see Acknowledgements page).

1.3.2 The consortium initially conducted a survey of existing work, both in the automotive sector and in other industrial sectors.

1.3.3 The consortium then formed eight subgroups to research specific issues relating to automotive software:

- diagnostics and integrated vehicle systems
- integrity
- noise, EMC and real-time
- software in control systems
- software metrics
- verification and validation
- subcontracting of automotive software
- human factors in software development.

1.3.4 Each group produced a report [1]–[8] that provides more detailed supporting information, and recommendations that have been incorporated into these Guidelines.

1.3.5 The results of the survey were continually updated throughout the study [9].

1.3.6 The eight reports and the survey report are available separately [1]–[9]. These reports provide additional background, more detailed recommendations and additional references that will be invaluable to the specialists in the field.

1.3.7 For the sake of brevity, the Guidelines do not always justify recommendations. The eight reports [1]–[8] contain the background material and justifications where appropriate.

1.4 Background

1.4.1 There are important differences between software and other forms of automotive engineering and components:

(a) Software is primarily a design, with no manufacturing variation, wear, corrosion or ageing aspects.

(b) It has a much greater capacity to contain complexity.

(c) It is perceived to be easy to change.

(d) Software errors are systematic, not random.

(e) It is intangible.

1.4.2 There are differences between automotive applications and applications in other industrial sectors:

(a) Production volumes are high (leading to manufacturing variation), related mechanical components are subject to wear and maintenance levels are difficult to assure. Therefore automotive software has an emphasis on data driven algorithms, parameter optimization, adaptive control and on-board diagnostics.

(b) Passenger car drivers receive little or no training compared with other users of computer-based products and services. Therefore automotive software requires an emphasis on failure management techniques based on the controllability of the vehicle.

(c) Traditional automotive test environments use real vehicles and components, as well as simulations. These are available to test systems and software extensively and safely before they reach the customer.

1.5 Scope and uses of the Guidelines

1.5.1 Scope

1.5.1.1 The boundaries between software and system design are not precise and there are many interrelated aspects. These Guidelines take a broad view of software, recognizing that many concerns attributed to software reliability are as much systems issues as they are related to software engineering.

1.5.1.2 The Guidelines make recommendations on system issues that are considered to influence software development.

1.5.1.3 The Guidelines are forward looking, supporting both today's developments and the vehicle features that are envisaged over the next decade.

1.5.1.4 The discipline of software engineering is already well supported by academia, text books and standards. Further guidance in this field is not appropriate at this time.

1.5.1.5 The eight specific issues for which guidance is given can be seen as aspects that provide a link between the established automotive engineering disciplines and software engineering, as shown in Figure 1.

1.5.2 Uses

1.5.2.1 This document is intended to be used by software engineers, managers and others involved in the creation, procurement and maintenance of embedded vehicle software.

1.5.2.2 The following uses are intended:

- guidance for creating contracts and specifications for software procurement
- as an introduction to issues of automotive software reliability
- the basis for training requirements within the automotive industry
- guidance for company quality procedures
- guidance for management on resource requirements
- to provide the basis of assessment
- as a foundation for a standard.

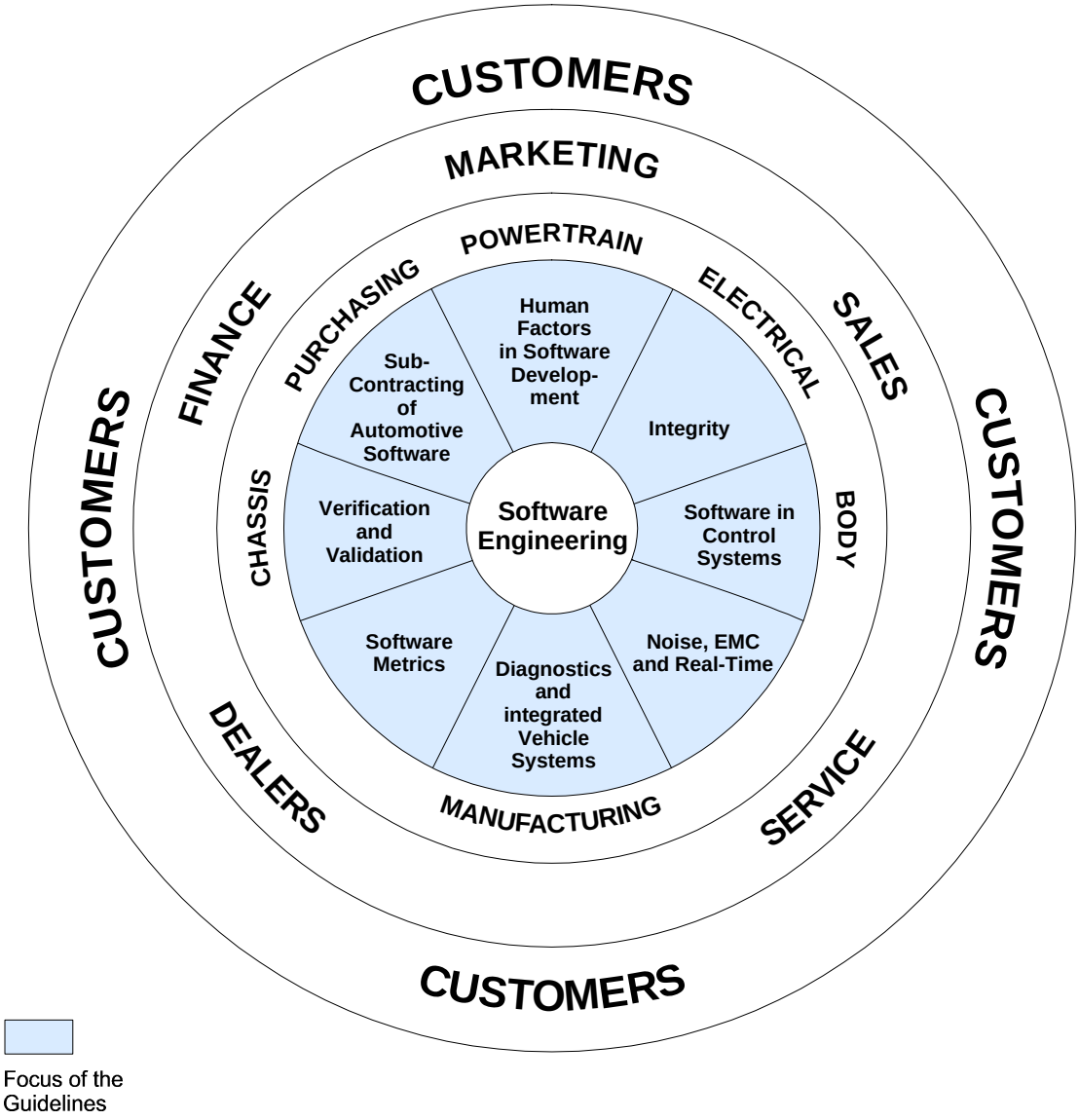


Figure 1. Scope of the MISRA Guidelines

1.5.2.3 Many of the recommendations are interrelated and overlap. In some instances, due to their generic nature, they may appear not to be in the optimum order within the document.

1.5.2.4 Not all the recommendations will apply in any one situation, so it is not anticipated that any project will, or can, conform to all the recommendations. Reasons for deviation from relevant parts of the Guidelines should be properly justified in the associated documentation. The Guidelines make recommendations on assessment of compliance.

1.6 Fundamental concepts

1.6.1 It is assumed and recommended that the users of this document operate a Quality Management System capable of meeting the requirements of ISO 9001 [10] as assessed under the guidelines in ISO 9000-3 [11] (e.g. TickIT [12]).

1.6.2 The users should have a working knowledge of the relevant software engineering practices.

1.6.3 Users of these Guidelines should be aware of present and emerging standards. Standards of particular importance include:

- the emerging generic standards in preparation by the International Electrotechnical Commission (IEC) [13, 14]
- the work of the ISO/TC22/SC3/WG11, Automotive Electronic Control Systems — Technical Documentation, working party [15]
- the IEEE software engineering standards collection [16]
- the IEE guidelines for the documentation of computer software [17].

1.6.4 Compliance with these Guidelines or with any standard does not of itself confer immunity from legal obligations.

1.6.5 These Guidelines are not intended to be prescriptive or proscriptive. As an element of flexibility it is important to allow users to adopt an approach that best matches their processes and organizational strengths. Hence the term “should” is normally used throughout in preference to “must” or “shall”. No particular significance should be attached to the use of alternative wording.

1.6.6 Techniques, activities and their rigour increase with integrity level. Equally, the weight of the relevant recommendations increases.

1.6.7 For the sake of readability there are no cross-references in this document.

1.6.8 Several basic principles associated with safety engineering and safety related systems design need to be understood:

(a) Safety, like justice and democracy, must be seen to be present.

(b) The greater the risk, the greater the need for information.

(c) Software robustness, reliability and safety, like quality, should be built in rather than added on.

(d) The requirements for human safety and security of property can be in conflict. Safety must take precedence.

(e) System design should consider both random and systematic faults.

(f) It is necessary to demonstrate robustness, not rely on the absence of failures.

(g) Safety considerations should apply across the design, manufacture, operation, servicing and disposal of products.

2. Definition of terms

2.1 Definitions

2.1.1 The terminology used in these Guidelines is taken from [13, 14, 18].

2.1.2 Terms specific to the automotive industry are defined in the text.

2.2 List of abbreviations

CAN	controller area network
CARB	California Air Resources Board (US)
CASE	computer aided software engineering
CV	*curriculum vitae*
DIS	draft international standard
EC	European Commission
ECU	electronic control unit
EMC	electromagnetic compatibility
EMI	electromagnetic interference
IEC	International Electrotechnical Commission
IEE	The Institution of Electrical Engineers
IEEE	The Institute of Electrical and Electronics Engineers (US)
I/O	input and output
ISO	International Organization for Standardization
KWP	keyword protocol
OBD	on-board diagnostics
OSI	open systems interconnection
QA	quality assurance
RAM	random-access memory
ROM	read-only memory
SAE	Society of Automotive Engineers (US)
VAN	vehicle area network
VDU	visual display unit

3. Software lifecycle

3.1 Project planning

3.1.1 Project definition

3.1.1.1 The use of software has many benefits in terms of cost, flexibility and functionality. However, because of the problems associated with complexity, it is important that systems containing software are only used where these benefits are required.

3.1.1.2 The recommendations in this section can be applied both at a vehicle system level and at a software system level [2, 6].

3.1.1.3 The project objectives should be clearly laid down before the start of a project. For example, the nature of a project will be different for a volume production system when compared to a research and development prototype.

3.1.1.4 The project definition should consist of a list of features and functions to be implemented. It should be agreed and documented by the appropriate authorities of the vehicle manufacturer and made available to the design and development teams before detailed work begins.

3.1.1.5 The project definition should also include any known legislative requirements, project assumptions and nonfunctional requirements (e.g. the use of an existing engine and/or ECU in a new vehicle).

3.1.1.6 It is particularly important to lay down and agree which features are fixed and which are customer options.

3.1.1.7 The project definition should be subject to strict change control.

3.1.1.8 Any changes made to the project definition will lead to increased risk from both safety and commercial points of view. The later changes are made, the greater the incremental risk.

3.1.1.9 Computer-based tools to be used during the project should be identified, and provision made for their procurement and resourcing at the project definition stage. Such tools are likely to include:

- configuration management tools
- calibration development aids
- modelling and simulation tools
- CASE tools
- rapid prototyping tools.

3.1.2 Lifecycle plans

3.1.2.1 An appropriate development approach should be defined and documented (e.g. by producing a Software Development Plan, a Quality Plan, a Safety Plan and a Configuration Management Plan). The design material should define:

(a) project organization.

(b) procedures to be followed.

(c) management processes to ensure compliance with the procedures in (b).

(d) planned management reviews of the effectiveness of the procedures in (b) and processes in (c).

(e) specific software deliverables together with their authors and readers.

3.1.2.2 The project team should progressively produce a Software Development Plan with clearly defined phases that, comprehensively and cost-effectively, cover the:

- specification
- design
- programming
- integration
- verification and validation (including testing)
- in-service support of the software.

3.1.2.3 The Safety Plan [14] should aim to ensure that verification and validation are performed throughout the project. These activities should be performed with a rigour appropriate to the integrity level.

3.1.2.4 Appropriate lifecycle plans should be defined before the start of a project for the system and the software. Figure 2 shows an example lifecycle.

3.1.2.5 Each individual system should have its own lifecycle that should be compatible with the project definition and the overall vehicle plan. An example is shown in Figure 3.

3.1.3 Planning for verification and validation

3.1.3.1 The verification and validation activities should be defined to establish a level of confidence in the software that corresponds the integrity requirements of the system. See Figure 4.

Software lifecycle (continued)

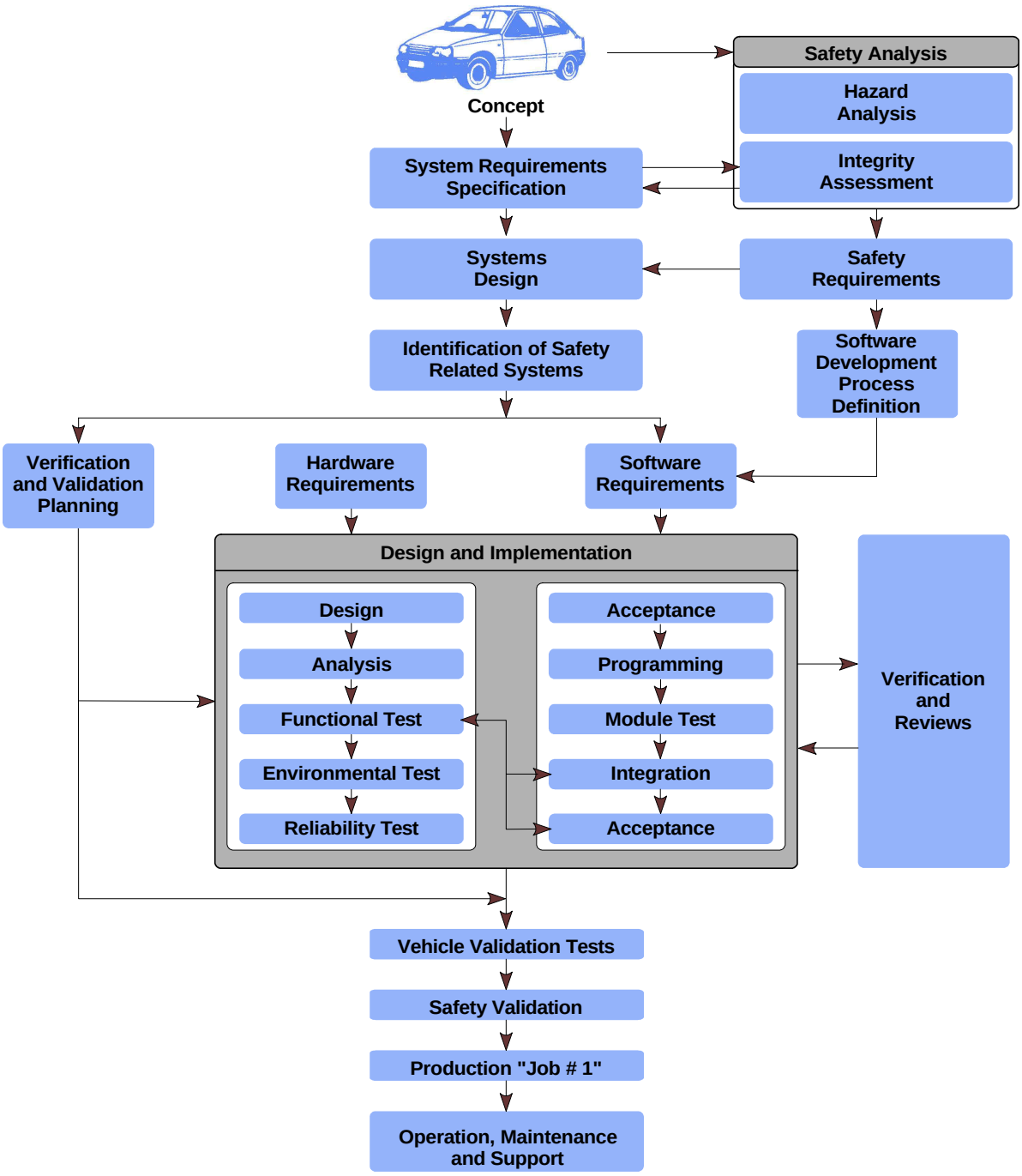


Figure 2. An example lifecycle

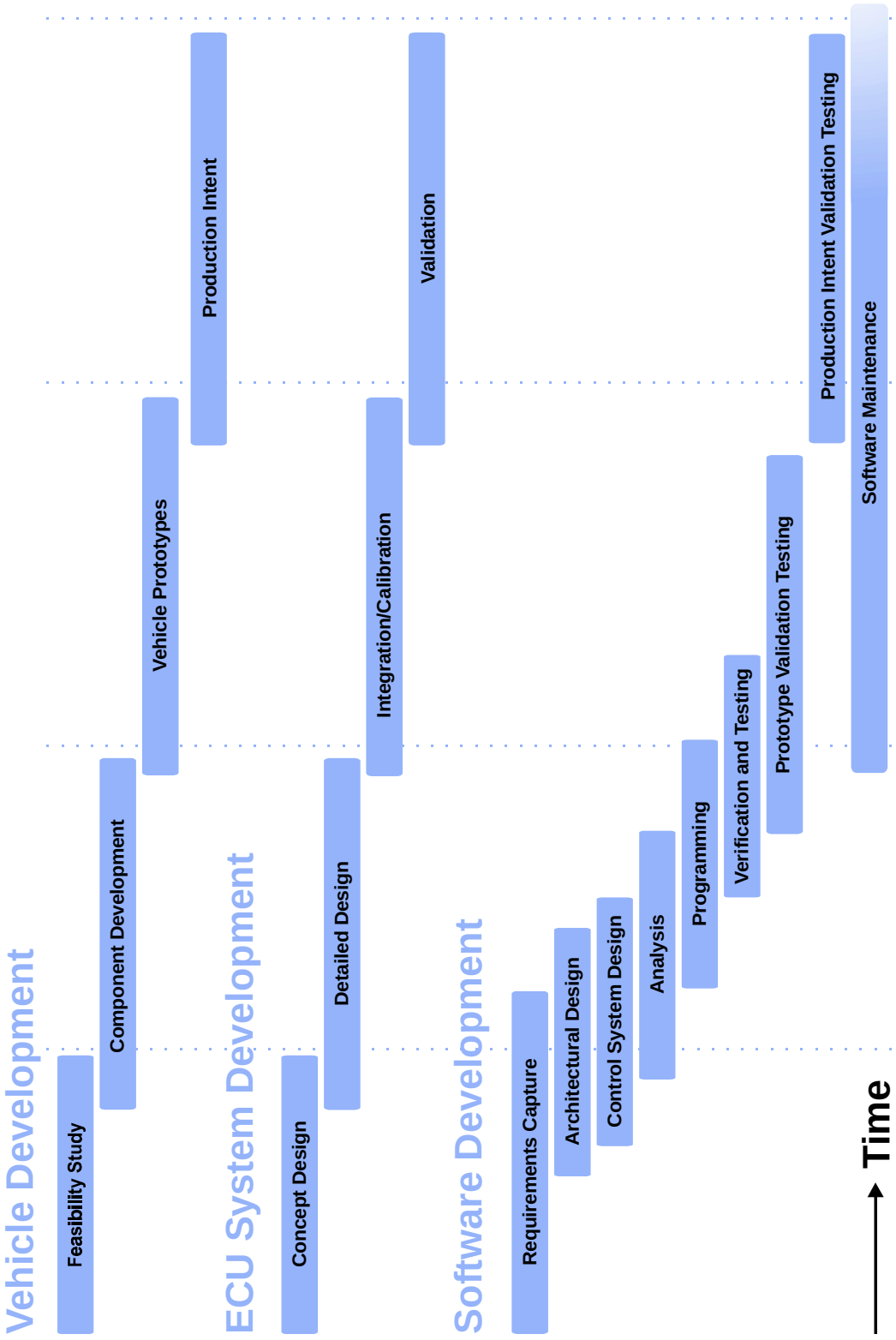


Figure 3. An example of vehicle, system and software lifecycle compatibility

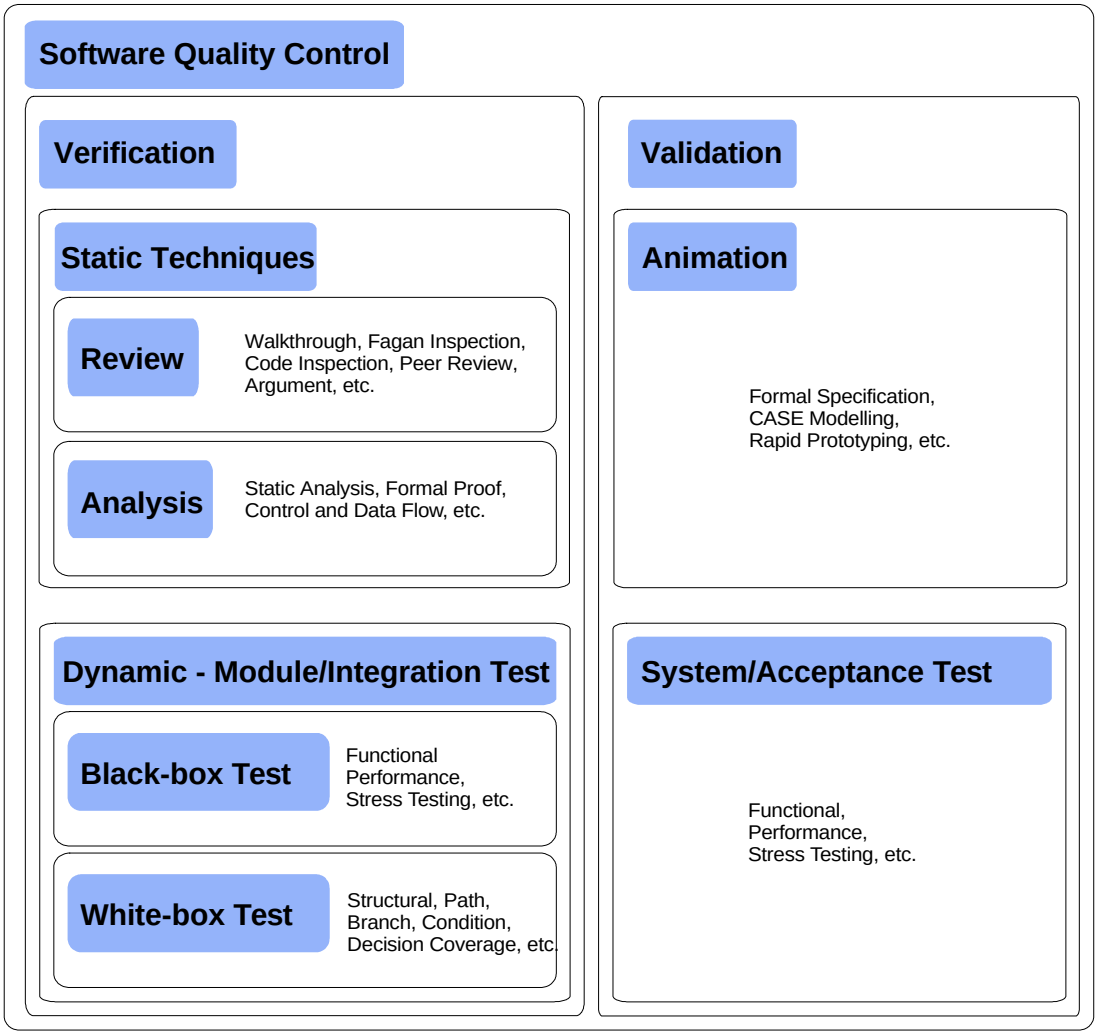


Figure 4. Software verification and validation

3.1.3.2 The purpose of software verification [6] is to ensure that each phase of the software development process maintains the attributes of the previous one without introducing errors. The outputs of each phase should be technically evaluated, by analysis and/or inspection with respect to its inputs, including standards and guidelines.

3.1.3.3 Safety verification is specifically concerned with the safety requirements and should proceed concurrently with the normal design and implementation process. Each phase of the development should also be verified with respect to safety before progressing on to the next development phase.

3.1.3.4 A verification plan should be established covering each phase of the software lifecycle.

3.1.3.5 The purpose of software validation [6] is to determine whether the software satisfies its intention and mainly consists of performing tests, but it should also involve analysis and mathematical techniques where appropriate. Full validation of the software cannot usually take place until it has been integrated with the rest of the system and often the vehicle.

3.1.3.6 Safety validation should be carried out to confirm specifically that all the safety requirements have been met.

3.1.3.7 A validation plan should be established for the software, system and vehicle.

3.1.3.8 To be effective, it is recommended that verification and validation are carried out by a person or team exhibiting a degree of independence from the design and implementation appropriate to the integrity requirements of the software and system.

3.1.4 Assessment

3.1.4.1 It is recommended that a company nominates a suitably qualified [19] and independent assessor and/or auditor to perform product assessment, in order to act as an advocate for the level of confidence in the safety delivered to the end customer [2].

3.1.4.2 An assessment process should be performed to demonstrate that the risks associated with the final system are at an acceptable level [14].

3.1.4.3 The higher levels of integrity should require greater independence of assessment in order to ensure that there is no bias from the development team, nor misplaced pressure from management.

3.1.4.4 The degree of independence [14] should be achieved by one or more of:

- different person
- different department or section
- different company or division.

3.1.4.5 The degree of confidence in a piece of software depends on the quality and the extent of the available information. This should cover the specification, design, implementation and testing of the software.

3.1.4.6 Preparing for assessment is a task that should begin at the start of a project; this also avoids the possibility of arriving at the point of assessment and finding that a vital piece of information has been omitted or overlooked. Early liaison with the assessor is recommended.

3.1.5 Reuse

3.1.5.1 "Off the shelf" or commercially available components containing software to be used in a project should be assessed to the required integrity level, and such reuse of existing software should be considered on a case by case basis.

3.1.5.2 The software that is intended for reuse should be fully compliant with the requirements of the new system that it is intended to be used in.

3.1.5.3 Software that is intended for reuse, as with all software, should have a standard of documentation such that its functions and interactions are easily understood by people who may not have been the original authors.

3.1.5.4 Software that has performed faultlessly for a number of years in the field may well offer a greater level of confidence in the final product than a wholly new program, provided it is used within the limitations of its design and is not modified.

3.2 Integrity

3.2.1 Introduction

3.2.1.1 Each system in a vehicle needs to have a level of integrity because there are requirements that it should not cause:

- harm to humans
- legislation to be broken
- undue traffic disruption
- damage to property or the environment (e.g. emissions).
- undue financial loss to either the manufacturer or the owner.

3.2.1.2 By achieving the required level of integrity, the risks associated with each of the requirements listed above should be reduced to an acceptable level [14]. Higher levels of integrity are achieved by the increased use of specialized methods and design requirements. This leads to an increasing cost, both of the development and of the end product.

3.2.1.3 The use of integrity levels permits a reasoned argument for assessing the degree of confidence that one should have in a system, which is commensurate with the inherent risk associated with the system function.

3.2.1.4 The recommendations and processes described in this section seek to enable a developer to:

(a) analyse a system to determine its integrity level.

(b) adopt a suitable development process in order to achieve the confidence level required in the software.

This ensures that the measures taken are both necessary and sufficient for the system being designed.

3.2.1.5 The system design may reduce the integrity level required for particular subsystems. For example, a failure mode set to a safe state by a hardware interlock of a suitable integrity level would remove that potential failure from the software hazard list. The hazard analysis therefore needs to be maintained throughout the development.

3.2.1.6 Once an integrity level for the software has been determined, an appropriate development approach should be defined, in order to gain the required confidence in the software.

3.2.1.7 The higher levels of integrity require more information and more rigorous application of software engineering techniques.

3.2.2 Safety analysis

3.2.2.1 A preliminary safety analysis [20] should be carried out as part of the requirements analysis phase, which becomes more detailed as the design progresses. Safety analysis is highly iterative and should be considered as a continuous process during requirements analysis and design [2].

3.2.2.2 A preliminary safety analysis should be performed early in the lifecycle to determine:

- the hazards associated with potential failures of the system
- the high level safety requirements necessary to reduce the risk associated with each hazard to an acceptable level [14]
- the initial integrity level.

3.2.2.3 The reasons for discounting potential hazards that are the result of extremely unlikely scenarios should be recorded during the preliminary safety analysis.

3.2.2.4 Once identified, each hazard should be analysed to determine its controllability category. The controllability approach was developed by the project DRIVE Safely [21].

3.2.2.5 The hazard with the highest controllability category will determine the integrity level of the system. This hazard analysis should be followed down the hierarchy to each system component (including software) to determine the integrity level required for each component.

3.2.2.6 The safety analysis is based on a hazard analysis using categories of controllability that are defined as follows:

Uncontrollable: This relates to failures whose effects are not controllable by the vehicle occupants, and which are most likely to lead to extremely severe outcomes. The outcome cannot be influenced by a human response.

Difficult to control: This relates to failures whose effects are not normally controllable by the vehicle occupants but could, under favourable circumstances, be influenced by a mature human response. They are likely to lead to very severe outcomes.

Debilitating: This relates to failures whose effects are usually controllable by a sensible human response and, whilst there is a reduction in safety margin, can usually be expected to lead to outcomes which are at worst severe.

Distracting: This relates to failures which produce operational limitations, but a normal human response will limit the outcome to no worse than minor.

Nuisance only: This relates to failures where safety is not normally considered to be affected, and where customer satisfaction is the main consideration.

3.2.2.7 Controllability applies to the ability of the vehicle occupants, not necessarily the driver, to control the safety of the situation following a failure. Systems for which the “occupants” may not necessarily be the driver include electric windows and electric seats.

3.2.2.8 The main steps involved in the determination of an initial integrity level are as follows (see Figure 5):

Software lifecycle (continued)

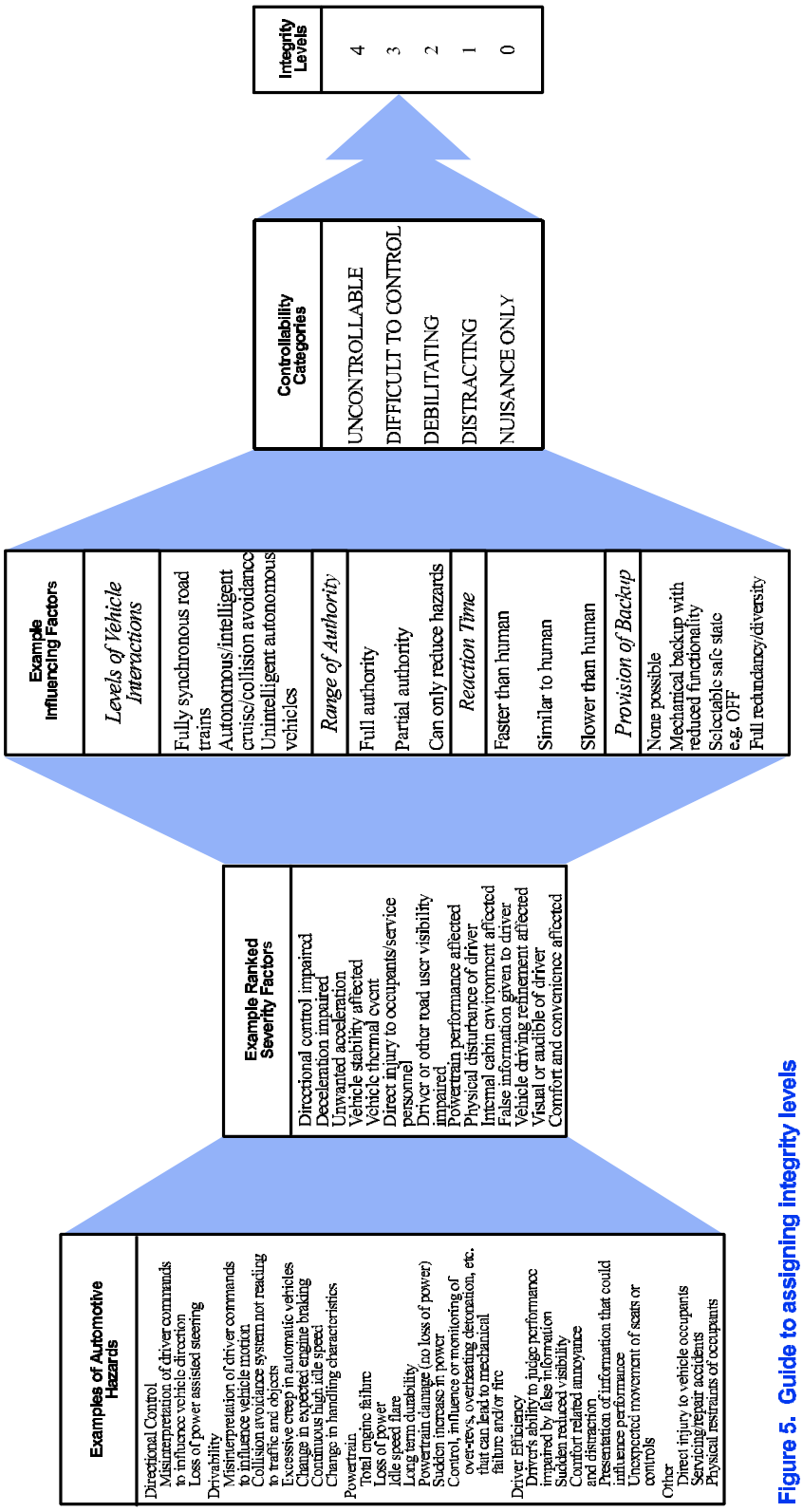


Figure 5. Guide to assigning integrity levels

(a) List all hazards that result from all the failures of the system.

(b) Assess each failure mode identified in step (a) to determine its controllability category.

(c) The failure mode with the highest associated controllability category determines the integrity level of the system (see Table 1).

Table 1 Integrity levels

Controllability Category	Acceptable Failure Rate	Integrity Level
Uncontrollable	Extremely improbable	4
Difficult to control	Very remote	3
Debilitating	Remote	2
Distracting	Unlikely	1
Nuisance only	Reasonably possible	0

3.2.2.9 Where the integrity level of the system relies upon the software, this should be identified specifically in the documentation.

3.2.3 Human factors in safety analysis

3.2.3.1 The driver and a vehicle can interact in many different ways, and the mechanism for these interactions should be considered carefully during safety analysis.

3.2.3.2 The following aspects may need to be considered when assessing hazards associated with the human interaction with a system:

- human reaction times
- ease of recognition of a situation
- attentiveness
- driver experience
- risk compensation (improved safety can lead to riskier behaviour)
- subversion or overriding of system functions
- smooth and readily perceived transfer of control from a system to the driver
- the workload of the driver, especially at the moment of failure.

3.2.3.3 It may also be necessary to consider the effects of the range of capabilities relating to:

- vision and hearing
- mental state (e.g. lack of sleep, jet lag)
- physical disability.

3.2.3.4 Although originating in a different industrial sector, Table 2 may be useful in assessing the significance of human error when interacting with a vehicle system (derived from [22]).

Table 2 Human error probability

Type of Human Behaviour	Human Error Probability
Extraordinary errors — those for which it is difficult to conceive how they could occur. Stress free, with powerful cues pointing to success.	10^{-5}
Errors in regularly performed, commonplace simple tasks with minimum stress.	10^{-4}
Errors of commission such as pressing the wrong button or reading the wrong display. Reasonably complex tasks, little time available, some cues necessary.	10^{-3}
Errors of omission where dependence is placed on situation and memory. Complex, unfamiliar task with little feedback and some distraction.	10^{-2}
Highly complex task, considerable stress, little time available.	10^{-1}
Process involving creative thinking, unfamiliar, complex operations where time is short and stress is high.	$1 – 10^{-1}$

3.2.4 Development approaches

3.2.4.1 The process of system design involves partitioning the system into subsystems, and assigning functionality to the various elements (hardware, software, etc.) The hazard analysis described above should be followed down to the components, in order to determine the integrity level of each component. In this way, it is possible to reduce the integrity level required for certain elements of the system. The integrity level of the software can be reduced by the use of hardware measures. These can be used to manage the more hazardous failure modes, thus removing the hazards or reducing their importance.

3.2.4.2 Once the integrity level of the software has been determined, an appropriate development approach should be defined. Table 3 gives guidance for each integrity level.

3.2.4.3 The following notes should be read in conjunction with Table 3.

(a) Specification and design.

Automated code generation from a formal specification is recommended at level 4 in order to remove human error from the process. It is recognized that this is not possible with current technology, and that validated tools will be required to make the approach usable.

(b) Languages and compilers.

Most, if not all, languages do not have precisely defined semantics. This means that not only may compilers contain faults, but different compilers for the same language may implement a given feature in different ways. In addition, some programming "features" such as pointers or recursion can cause unpredictable behaviour. This has led to the recommendation that restricted subsets are used at levels 2 and 3 and that certified compilers with proven formal semantics (not currently available) are used at level 4.

Also, since there are currently no formally proven compilers, it may be necessary to show that the machine code (object) does indeed reflect the high level language version (source) of the program. For this reason, or for reasons such as required speed of execution or restricted memory availability, assembly languages are still being used at all levels of integrity.

(c) Configuration management: products.

"Products" in this context means all documents generated or used during the development process (i.e. the set of information used for assessment). Additionally for level 2 and above, the tools used should also be under configuration management.

(d) Configuration management: process.

Confirmation process at level 2 implies a means of confirming that the software has been built from identifiable components. At levels 3 and 4, the automated confirmation process is to confirm that only intended changes are made, and to perform automated impact analysis of proposed changes.

Table 3 Summary of requirements

Development Process	Integrity Level				
	0	**1**	**2**	**3**	**4**
Specification and design	ISO 9001	Structured method.	Structured method supported by CASE tool.	Formal specification for those functions at this level.	Formal specification of complete system. Automated code generation (when available).
Languages and compilers		Standardized structured language.	A restricted subset of a standardized structured language. Validated or tested compilers (if available).	As for 2.	Independently certified compilers with proven formal syntax and semantics (when available).
Configuration management: products		All software products. Source code.	Relationships between all software products. All tools.	As for 2.	As for 2.
Configuration management: processes		Unique identification. Product matches documentation. Access control. Authorized changes.	Control and audit changes. Confirmation process.	Automated change and build control. Automated confirmation process.	As for 3.
Testing		Show fitness for purpose. Test all safety requirements. Repeatable test plan.	Black box testing.	White box module testing — defined coverage. Stress testing against deadlock. Syntactic static analysis.	100% white box module testing. 100% requirements testing. 100% integration testing. Semantic static analysis.
Verification and validation		Show tests: are suitable; have been performed; are acceptable; exercise safety features. Traceable correction.	Structured program review. Show no new faults after corrections.	Automated static analysis. Proof (argument) of safety properties. Analysis for lack of deadlock. Justify test coverage. Show tests have been suitable.	All tools to be formally validated (when available). Proof (argument) of code against specification. Proof (argument) for lack of deadlock. Show object code reflects source code.
Access for assessment		Requirements and acceptance criteria. QA and product plans. Training policy. System test results.	Design documents. Software test results. Training structure.	Techniques, processes, tools. Witness testing. Adequate training. Code.	Full access to all stages and processes.

See [2] for full details.

(e) Testing.

Test coverage at level 4 (100%) implies an assessment of the coverage against defined criteria. This should be defined in the project planning documentation. A coverage analysis approach, such as linear code sequence and jumps [23], should be used.

(f) Verification and validation.

All tools that could affect the integrity of the product should be formally validated at level 4. It is not necessary to formally validate tools such as word processors and project planning tools, and arguments may be made for not formally validating other tools (e.g. CASE tools) where the output is fully validated. Care should be taken to ensure that in-house utilities and "scaffold code" linking tools together are appropriately validated.

(g) Access for assessment.

The degree of independence should be justified in the planning documentation based on the level of integrity and organizational issues.

3.3 Requirements specification

3.3.1 Whole vehicle architecture

3.3.1.1 Before attempting to write requirements specifications for individual ECUs, particularly for integrated systems, an architecture of the complete vehicle electrical system should be defined in conceptual form. The objective is to ensure that the requirements specifications for subsystems and components are consistent and compatible with one another. In defining an architecture, teams of component engineers, system engineers and management should work together to give a broad view of the whole vehicle electrical system [1].

3.3.1.2 Be very clear about the objectives for a vehicle design before entering an architecture exercise and ensure that this is agreed with the area in the company responsible for product definition and strategy. Agree the documented deliverables for the architecture study. Figure 6 gives an example of these deliverables.

3.3.1.3 Results from the safety analysis should be fed into the design at an early stage. The architectural design is at least as important as the detailed design stages in minimizing the effort required to meet safety requirements. The development of an optimal architecture will involve several iterations.

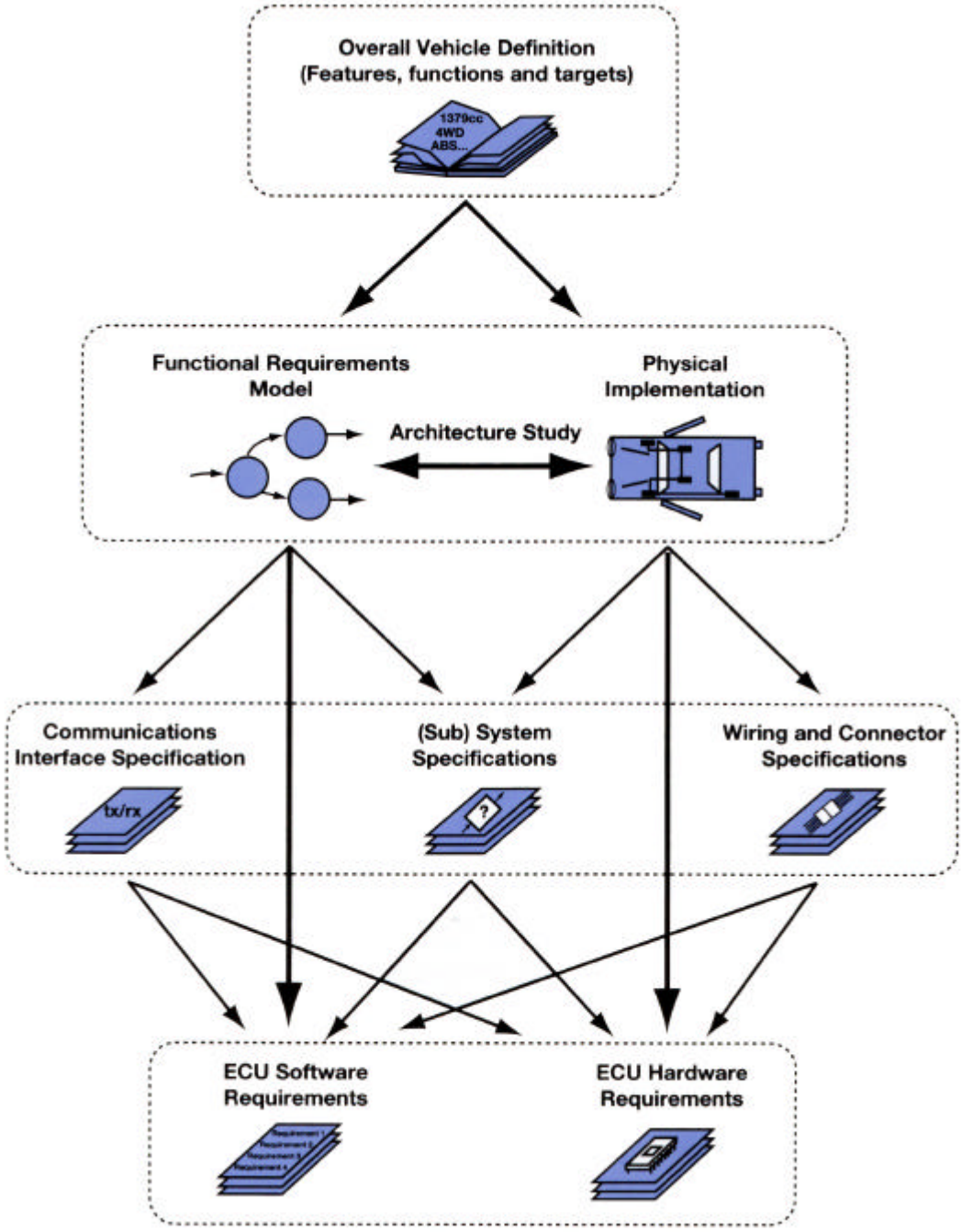


Figure 6. Example architecture study deliverables

3.3.1.4 Where appropriate, electrical isolation should be considered for functions needing a high level of integrity. This will ease the demonstration of compliance with the safety requirements.

3.3.1.5 To ensure all aspects are covered, split the work into functional and implementation aspects, sometimes called logical and physical partitioning, and address each with a team. Close cooperation and communication between these two architecture teams is essential.

3.3.1.6 For logical partitioning, create and maintain a model of the functional requirements of the complete vehicle electrical system. Split the functional requirements modelling into four phases:

- analysis of the high level requirements
- definition of the system boundary and the environment
- detailed top-down modelling of the functional requirements
- implementation of functions on the vehicle hardware.

3.3.1.7 For physical partitioning:

- consider implementation (e.g. sensors and actuators) at an early stage to allow the architecture to encompass other vehicle design aspects such as packaging and body design
- identify the interfaces required by each subsystem (I/O identification)
- agree the available technology options that may affect the architecture (e.g. communications protocols and switching)
- review these options as the architecture study progresses.

3.3.1.8 Keep up to date lists of all current assumptions and open or unresolved issues made during the architectural design stage. Keep these under version control.

3.3.1.9 The vehicle manufacturer should perform a proper functional analysis and maintain an overall system specification. If multiple suppliers are to be used for sourcing programmable components that are to be integrated, then this is essential to ensure that the component specifications given to each supplier are compatible.

3.3.1.10 Strict change control procedures should be enforced for software, hardware and other appropriate design material at all times. Changes should be subject to review by the architecture teams responsible.

3.3.2 Vehicle control systems

3.3.2.1 In order to create a comprehensive requirements specification for a control system, it is essential to have a thorough understanding of the complexities and mathematics of the relevant control theory [4].

3.3.2.2 Give special consideration to the required attributes of a control system at the requirements specification stage (see Figure 7). Performance attributes should be accurately quantified. Assumptions, uncertainties and unknowns should be identified and documented in the requirements specification.

3.3.2.3 It is essential to demonstrate that automotive control systems are stable under all operating conditions. This may be difficult to calculate mathematically with significantly nonlinear systems, such as those employed in vehicle applications.

3.3.2.4 Many diverse methods may be used to assess stability. No one method may be identified as preferred. It should not be assumed that similar results will be obtained from different methods for a given solution.

3.3.2.5 The selected method or methods for the calculation of control system performance, and especially stability, should be defined in the requirements specification. It is particularly important to be consistent where parts of a system are created by different teams.

3.3.2.6 Parameters critical to system stability should be identified in the requirements specification.

3.3.2.7 Several control techniques may be required in a complex control application. Selection should be on the basis of the following characteristics:

- predictability
- defined target hardware resources
- deviations between implementation and theory
- robustness (especially of input data)
- testability
- simplicity.

3.3.2.8 It is important to consider carefully the implications of:

- dynamic range
- linearity
- conversion time
- response time
- noise
- damping
- effects of arithmetic systems used (e.g. fixed, floating point)
- effect of accumulation of errors and rounding.

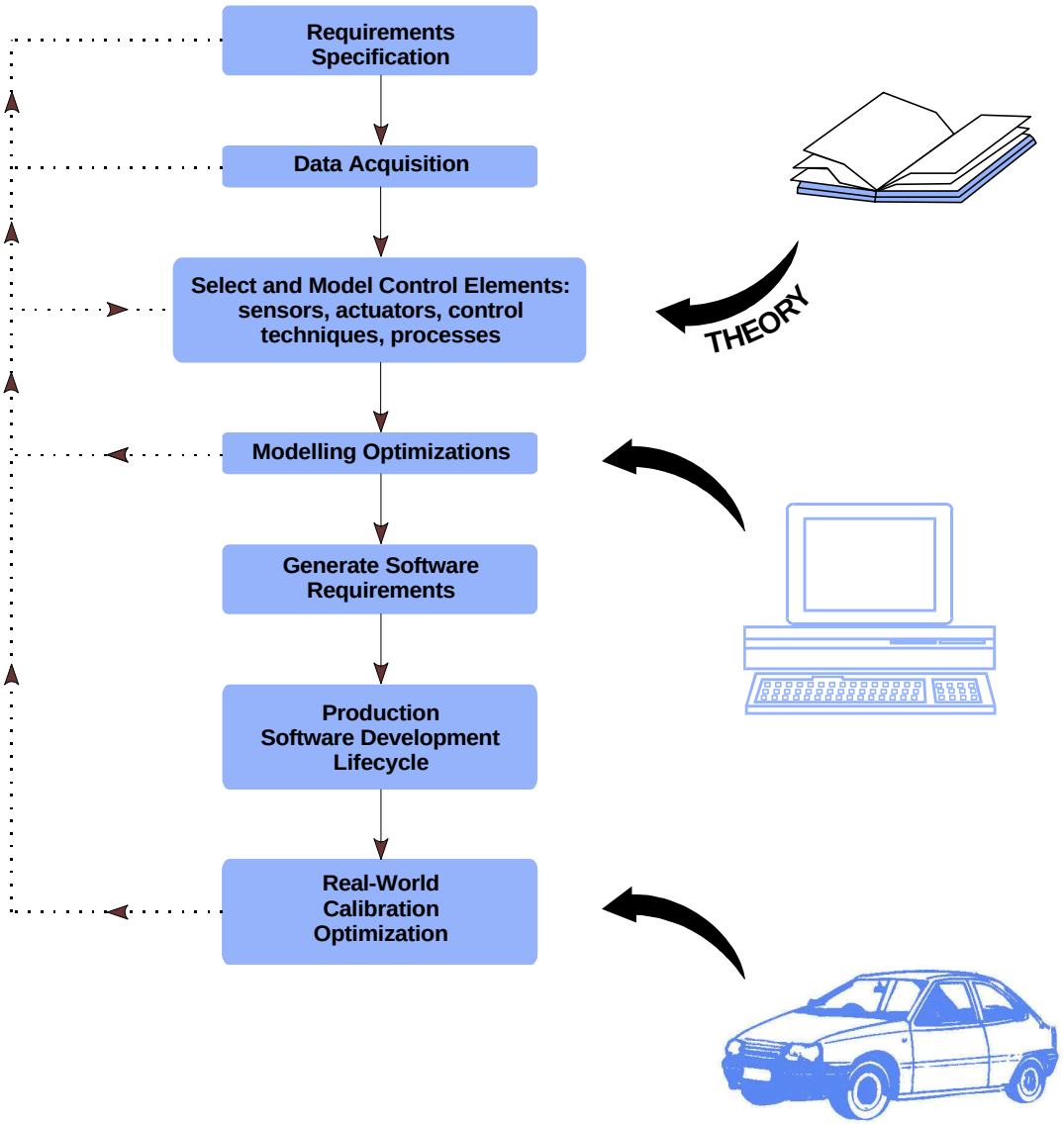


Figure 7. Control system design

3.3.2.9 In order to perform accurate control, it is important to acquire accurate knowledge regarding the state of the system being controlled. This can be difficult because of the invisibility of some data, lag in the system, or poor understanding of the control actions needed. This can result in poor accuracy, compensated for by the manual adjustment of parameters during development. However, there are techniques that may be employed to improve control accuracy:

- inferred variables
- statistical estimates
- artificial variables
- observers
- Kalman filters
- feedforward control
- predictive control.

3.3.2.10 Sampling is an inherent part of digital control techniques. Attention should be paid to the resulting issue of aliasing; in particular, the sampling rate should be high enough to minimize aliasing.

3.3.2.11 If the sampling rate is software controlled, software timing should not be based on loop execution time alone.

3.3.2.12 Both resolution and system response times should meet the system requirements.

3.3.2.13 Consider the effect of quantization. If dithering is used to reduce the quantization errors, consider its effect on system response, accuracy and linearity.

3.3.2.14 Take adequate design precautions to reduce the effects of noise signals (e.g. anti-aliasing filters).

3.3.2.15 Evaluate the effects on closed-loop control of:

- input filters
- output filters
- controlled object response.

3.3.2.16 The definition of transient conditions requires particular care.

3.3.2.17 Open-loop control has widespread application in vehicle control systems during:

- a system development phase, for quantifying parameters
- initialization, to aid in checking system component health
- a default or "limp home" condition
- transient system conditions, where the speed of response cannot be provided by closed-loop action.

3.3.2.18 Care should be taken in mapping parameters for open-loop control. In particular, the effects of sensor errors may be more pronounced than in a closed-loop system. Any development aids used in the calibration of open-loop systems should be designed for ease of use.

3.3.2.19 Calibration is a skilled activity. Experience is essential for the achievement of optimum performance.

3.3.2.20 Formal mathematical methods, especially for specification, may appear appropriate for this subject. However, control theory already has its own mathematical basis, therefore the benefits of formal mathematical methods may be outweighed by added complexity when attempting to apply them in this area. Many control systems incorporate concurrency, which is not yet widely supported by current industrial strength formal mathematical methods.

3.3.3 Noise and electromagnetic compatibility

3.3.3.1 Although radio frequency emissions are part of electromagnetic compatibility (EMC), only immunity is considered here [3]. Hardware measures to minimize electromagnetic interference (EMI) are the prime defence; however, this section describes only some additional measures that may be taken in software.

3.3.3.2 Information on hardware EMC measures may be obtained from documents such as papers from international conferences and seminars, or the *Guidelines for the Achievement of EMC in Motor Vehicles* [24] in which standards and sources are listed. Test methods and performance levels are given in ISO standards and the draft EC Directive amending 72/245/EEC.

3.3.3.3 EMC standards identify test levels and failure modes for hardware functions. These may not be related directly to software integrity levels.

3.3.3.4 EMC measures, in common with other aspects of system design, involve a trade-off between hardware and software solutions. Each has its own implications that should be assessed when making design decisions.

3.3.3.5 Specific techniques for defence against EMI, whether hardware or software, should not be attributed to an integrity level, since individual techniques may behave differently under diverse circumstances, and may not give consistent results.

3.3.3.6 EMI effects can only be minimized, not prevented, either in hardware or software. The most important role that software has in relation to EMC is in providing a means of recovery from the effects of interference.

3.3.3.7 Hardware filters may introduce effects in the control system response that may be impossible to compensate for fully in software.

3.3.3.8 Software may be used to:

- digitally filter data
- compare data with constant values, or values inferred from related signals, to identify errors
- perform error detection and correction
- dynamically adjust scaling to optimize signal-to-noise ratio.

3.3.3.9 The following considerations apply to processors:

(a) If the processor suffers EMI, software generally cannot provide any defence, and the system will require reinitialization.

(b) Single chip controllers may improve immunity to some types of EMI, but may be less immune to others, depending on the design of the controller. Their integrated nature may make diagnosis and correction by software of EMI effects difficult.

(c) It is essential to have an independent hardware watchdog designed to ensure reset and reinitialization in the event of EMI corruption. Do not rely on software watchdogs alone.

(d) When choosing the timeout interval for a hardware watchdog, consider the response time of the system in the event of a fault.

(e) Processors may be less immune to EMI when operating at the limit of their clock rates.

3.3.3.10 The following considerations apply to data held in memory:

(a) Corruption of data in memory can be identified and corrected by software. Routines to check for compatibility with expected results, against other data or by checksum comparisons, may be useful.

(b) Data in ROM is generally less likely to be corrupted than in other types of memory. Critical data stored in RAM should be periodically recalculated and should recover from corrupted values.

3.3.3.11 The following considerations apply to I/O devices:

(a) The dynamic range of analogue to digital converters should be chosen to avoid "clipping" and maximize signal-to-noise ratio. Software cannot necessarily detect and compensate for signals that are corrupted in this way.

(b) If multiplexed I/O is employed, it should be reinitialized by the software prior to use.

(c) Whilst noise outside the frequency response of the system may be minimized by the use of either hardware or software filtering, in-band noise can be hard to detect and correct, especially if it affects a pulse train. Use software to compare frequency-critical data with expected or related data, and use default values, or resample, if an unexpected result is observed.

(d) Registers in programmed I/O devices should be regularly reloaded, or compared with their initialization data in ROM and reinitialized if in error.

(e) For slowly changing signals, compare newly acquired values with previous values, and discard if the variation is excessive.

3.3.3.12 Communications data and processes may be corrupted by EMI. Software detection and correction techniques can be very effective, and should be used in addition to hardware measures.

3.3.4 Verification and validation of software requirements

3.3.4.1 Team oriented reviews and analyses should be carried out to confirm that the software requirements specification is unambiguous, accurate, complete and can be verified [6]. The degree with which this should be done depends on the integrity level [2].

3.3.4.2 The software requirements specification should include the software functional requirements, the software nonfunctional requirements and the software safety requirements. The latter should be specifically identified. The requirements hierarchy is shown in Figure 8.

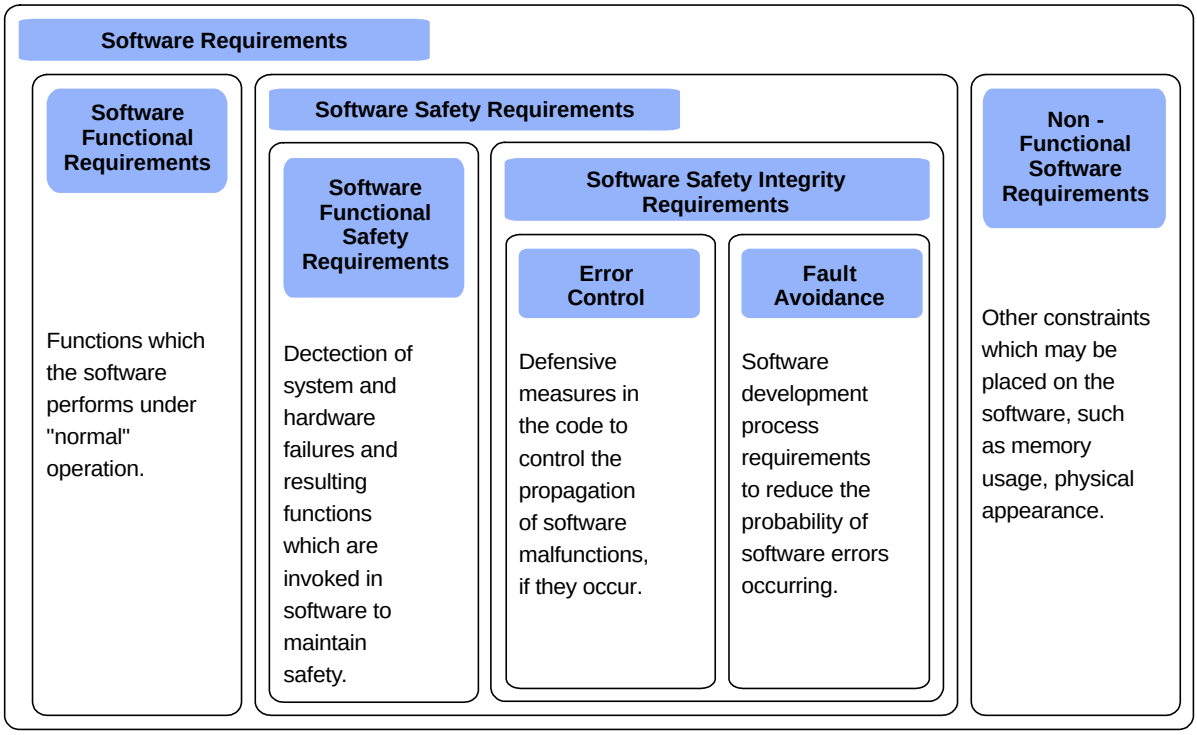


Figure 8. Requirements hierarchy

3.3.4.3 The software safety requirements should describe the potential system failure conditions together with the functions that the software should perform as a result. They should also highlight any safety integrity requirements and design constraints.

3.3.4.4 Safety integrity requirements should be divided into process measures to avoid, and design attribute measures to control systematic and random failures. These requirements include:

- logical and physical partitioning
- diversity
- redundancy
- condition monitoring
- self-test.

3.3.4.5 The software requirements specification should be verifiable. As a result the verification process may impose additional requirements or constraints on the software development process.

3.3.4.6 Design constraints should be analysed to see if they are essential, unnecessary, or necessary but in the wrong form.

3.3.4.7 The safety invariants (statements of the safety of the system that should apply in all circumstances) are part of the system requirements specification.

3.3.4.8 Each software high level requirement should be traceable to one or more system requirements, and *vice versa*.

3.3.4.9 Reviews of the preliminary safety analysis should be carried out as part of the project requirements review.

3.3.4.10 The hardware and software interfaces should be carefully reviewed to ensure compliance with the overall system safety requirements. It is essential that the hardware and software teams understand the global impact of any measures they take to comply with system, and especially safety, requirements.

3.3.5 Tools and techniques for requirements specification

3.3.5.1 The requirements specification structure should attempt to mimic the problem structure, and it should be easy to change if the problem changes.

3.3.5.2 The tools to be used on the project, including tools used on previous projects, should be reviewed during the requirements specification phase.

3.3.5.3 Use the appropriate tools to aid in the capture of requirements; in particular, consider CASE tools to support functional modelling during the definition of a vehicle architecture (see Table 3).

3.3.5.4 An animation and/or a prototype can help the procurer and the supplier to assess the adequacy and correctness of the specification, and should be considered to stabilize the requirements before the design begins.

3.3.5.5 If rapid prototyping is used there should be no attempt to use or modify the prototype code for production.

3.3.5.6 Formal mathematical methods can help to ensure that a specification is unambiguous and well structured, and can support functional consistency checks, as well as providing a basis for sound review. See Table 3.

3.3.5.7 A good formal specification should be accompanied by a natural language description to support the precise mathematics to help those not familiar with the notation, and should read clearly with the mathematics omitted. The mathematics, however, are the definitive specification.

3.3.5.8 If used, the application of formal mathematical methods should be supported by commercially available tools.

3.3.5.9 Fault recovery or fault tolerance schemes should be designed to ensure the maintenance of a safe and legal state at all times.

3.4 Design

3.4.1 Real-time implications

3.4.1.1 The use of CASE tools can assist in the assessment of the timing performance of the initial software design [3].

3.4.1.2 There are many interrelated aspects of the initial design that are best analysed and resolved using a powerful computer-based modelling tool, such as:

- types of input/output regimes
- estimates of the number of statements or instructions to be executed
- processor and memory resource knowledge and information
- average instruction execution times
- knowledge of any real-time kernel, executive or operating system
- software filtering requirements
- failure management requirements
- relationship between the high level language statements and the code generated.

3.4.1.3 Many of the aspects involved will be poorly defined at the outset and will have a large processing variability associated with them, so a large contingency for unknowns will be needed in the initial estimate of processing power and memory. Such aspects include:

- searching (e.g. table lookup)
- sorting
- iteration (e.g. statistical, curve fitting methods)
- stack, list, or queue oriented processes.

3.4.1.4 Queue-based systems pose particular problems associated with computer resource planning and allocation. An understanding of the consequences of queues and the techniques to handle them is essential. See Figure 9 for the example of the impact of queuing time.

3.4.1.5 Interrupts are useful in achieving good response times in a real-time control system. However, the quality and robustness of the software in this area can have a major impact on the overall system reliability. Interrupt routines require special attention to ensure robustness because:

(a) Spurious interrupts due to noise can saturate the processor through excessive activity inhibiting other parts of the software from initiating fall-back operations.

(b) Program structures using interrupts may create an unbounded number of paths through the software, increasing the difficulty of integration and system testing.

(c) Asynchronous interrupts make it difficult for the software design to cater for all scenarios, which may result in data corruption.

(d) Several concurrent processes initiated by interrupts that compete for resources may reach deadlock.

(e) There can be conflicts of priority between one task's real-time response requirements and other tasks' integrity requirements.

3.4.1.6 For robust interrupt handling, a selection of these or compatible techniques are recommended:

(a) Use hardware filtering and buffering as the first line of defence against noise and excessive rates of data.

(b) Use sufficiently independent and diverse mechanisms that can be provided by hardware and software to back up and protect interrupt handling, such as masking, prioritization and supervisor modes.

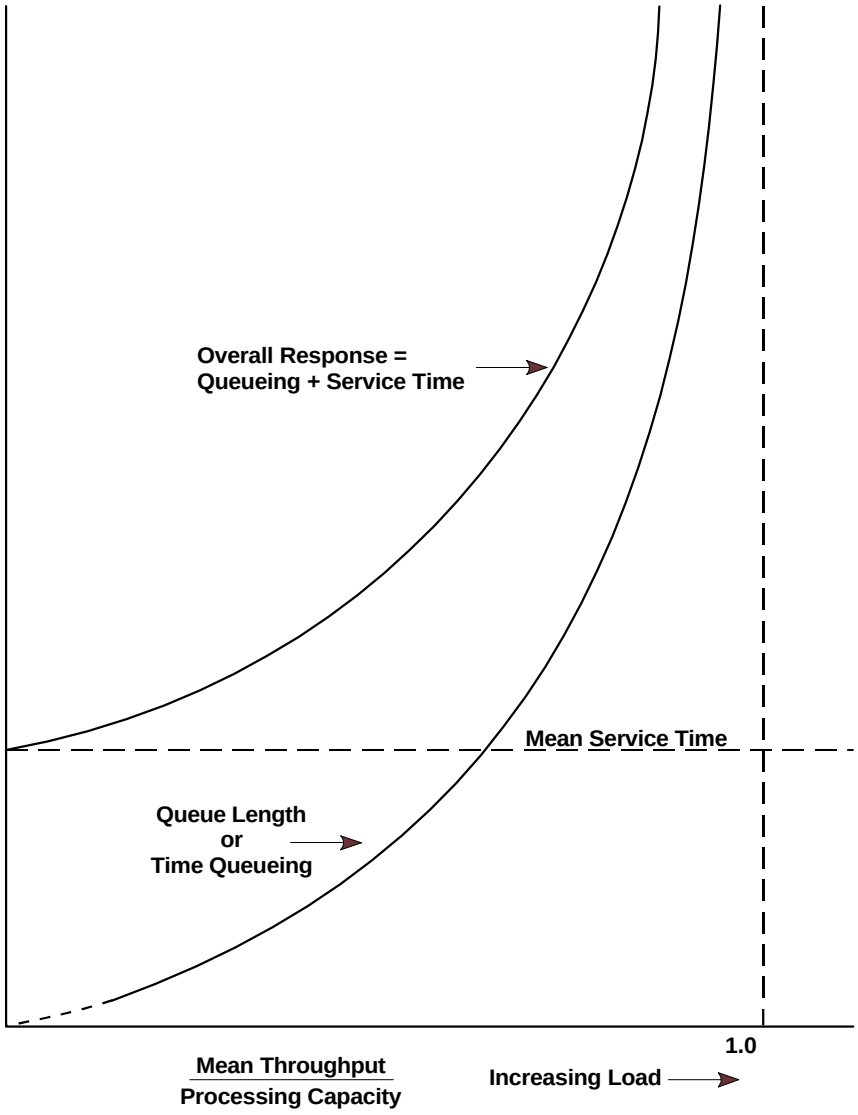


Figure 9. The impact of queueing time

(c) Maintain strong separation and modularity between the interrupt routines and other non-interrupt routines.

(d) Use an existing kernel where possible.

(e) Avoid using the program control stack for data. Set stack lengths to the theoretical maximum.

(f) Follow suppliers' recommendations for interrupt design.

(g) Employ personnel with experience of designing interrupt handlers for their implementation and modification.

(h) For the higher integrity levels, consider using an independent monitor processor to initiate a safe state.

3.4.1.7 The diverse nature of many of the proposed techniques means that complementary mechanisms can be overlaid to provide increasing robustness with rising integrity levels [3].

3.4.2 Floating point arithmetic

3.4.2.1 Floating point arithmetic has several advantages related to quality and reliability:

- numerical range is increased
- the potential for numerical overflow is reduced
- manual scaling by the programmer is unnecessary, thus one source of software error is eliminated
- numerical accuracy is increased due to greater word length.

3.4.2.2 The use of floating point, however, does not solve all numerical problems and may introduce some new ones [3]. An awareness and understanding of general numerical methods remains essential to resolve issues relating to:

- conversions to and from real and integer numbers and rounding problems
- handling of divide-by-zero conditions
- handling "Not-A-Number" situations
- propagation of numerical errors through filters and integrators
- build-up of rounding errors
- the effect of differences in small numbers within ill-conditioned equations.

3.4.2.3 In many real-time applications, the use of hardware to implement floating point may be required to meet response time requirements. This imposes different and special requirements on the system and software development processes:

(a) Floating point co-processors can be corrupted by EMI and fail independently from the main processor. Therefore, the additional EMC constraints and on-board diagnostics requirements for floating point units are similar to those required for off-chip RAM and other separate devices.

(b) It may be necessary to save and restore floating point data on receipt of an external interrupt. There can be a considerable processing overhead associated with combining floating point hardware with a requirement to handle a high input data rate.

(c) Validation of floating point hardware units is a major and time consuming exercise and evidence should be sought from the semiconductor supplier.

3.4.2.4 The use of hardware floating point arithmetic is not the ideal solution in every situation. An alternative approach is to use partial floating point implemented in software, only where it is required [3].

3.4.3 Modelling

3.4.3.1 The use of a modelling tool is recommended to aid the design of control systems. There are a significant number of commercial modelling packages available, as well as custom designed and maintained packages [4].

3.4.3.2 Give careful consideration to the choice between commercially available and custom modelling packages. Commercial packages offer great flexibility and ease of use; custom packages may be much more powerful for the given application for which they are designed.

3.4.3.3 Modelling packages may be used as a validation tool provided adequate confidence in the package itself can be justified.

3.4.3.4 Do not use the simplicity of good graphical user interfaces in commercial modelling tools as a substitute for experienced staff.

3.4.3.5 Ensure that the data acquired for design and modelling purposes is sufficiently representative of the functions for which it is being considered.

3.4.3.6 Simulation and emulation can be beneficial to the development process.

3.4.3.7 In order to have sufficient confidence in the results of simulation and emulation, software quality management should be applied to them.

3.4.3.8 If emulation techniques are to be used as part of the prototype stage, manufacturers should satisfy themselves of the safety of the emulation in advance. It should allow inspection, and not modification, of control algorithms.

3.4.4 Optimization and adaptive control

3.4.4.1 Optimization of control [4] is the process of developing a control system to be as efficient as possible within defined constraints (see Figure 7). In relatively linear systems, good optimization may be achievable by the application of theory or modelling. In the case of significantly nonlinear systems, however, optimization is often not straightforward, and it is important to consider the following factors:

(a) In significantly nonlinear systems it is possible for the input fundamental frequency not to be present in the output.

(b) In nonlinear systems there may be local nonlinearities, which are impossible to analyse and quantify; thus optimization may have to be done by empirical methods.

(c) The most significant problem of nonlinear control systems is that the dynamic behaviour becomes a function of amplitude.

3.4.4.2 Adaptive control is effectively continuous on-line optimization of parameters in a closed-loop control system, but its use should be assessed in relation to the possible failure modes and range of authority.

3.4.4.3 As few variables as possible should be used in the adaptation algorithm, in order to restrict the number of degrees of freedom.

3.4.4.4 Even where adaptation is used, it may still be necessary to use manually derived look-up tables in the adaptation algorithm to compensate for significant nonlinearities.

3.4.5 Communications and multiplexing

3.4.5.1 If the analysis of requirements highlights the potential for using a bus based architecture [1], where possible an internationally recognized protocol standard should be used for network communications, for example ISO DIS 11898 [25] (CAN High Speed), ISO DIS 11519 [26] (VAN and CAN Low Speed), SAE J1850 [27].

3.4.5.2 Non-standard proprietary protocols are designed with a specific application in mind, where some special attribute such as cost, speed or integrity dominates. Vehicle manufacturers should assure themselves that proprietary protocols are adequate for their intended use.

3.4.5.3 As an alternative to dedicated hardware for lower speed communications systems, it is possible to support some protocols in software with a suitable microcontroller.

3.4.5.4 Where communication is necessary for systems requiring a high level of integrity, it is recommended that a dedicated bus is used. The use of a high integrity protocol should be considered (e.g. CAN, VAN, J1850).

3.4.5.5 Because of the differing requirements of some subsystems, for example those for real-time control and driver/passenger services, multiple networks can be used. If a gateway is required, it should be designed in as part of each system. The data transfer requirements for the gateway can be determined during the final stage of the functional requirements modelling process, and should aim to minimize the transfer traffic.

3.4.5.6 Responsibility for the system, including the definition and integration of the communications network, should be assigned to a defined person or organization, usually the vehicle manufacturer. This person or organization should also maintain close control over, and ensure good communications between, the different design teams that use the multiplex bus. Component suppliers should work closely together and accept constraints on their software design.

3.4.5.7 Communications and multiplex systems should be specified and designed using a layered model of the network services, for example the ISO-OSI seven layers. For most automotive systems a three layer model consisting of physical, transfer and application layers is adequate (see Figure 10).

3.4.5.8 The physical layer specification should ensure that interfaces between ECUs are defined and adhered to. Where possible use an existing or a common standard.

3.4.5.9 The transfer layer should carry out packaging of data into message frames, message transmission scheduling and checks on received message timeouts and corruption (e.g. cyclic redundancy check).

3.4.5.10 The application layer should be designed to perform range and/or rate-of-change checks on data associated with high integrity functions (plausibility checks). These are normally carried out at the point of reception, but can also be applied before transmission.

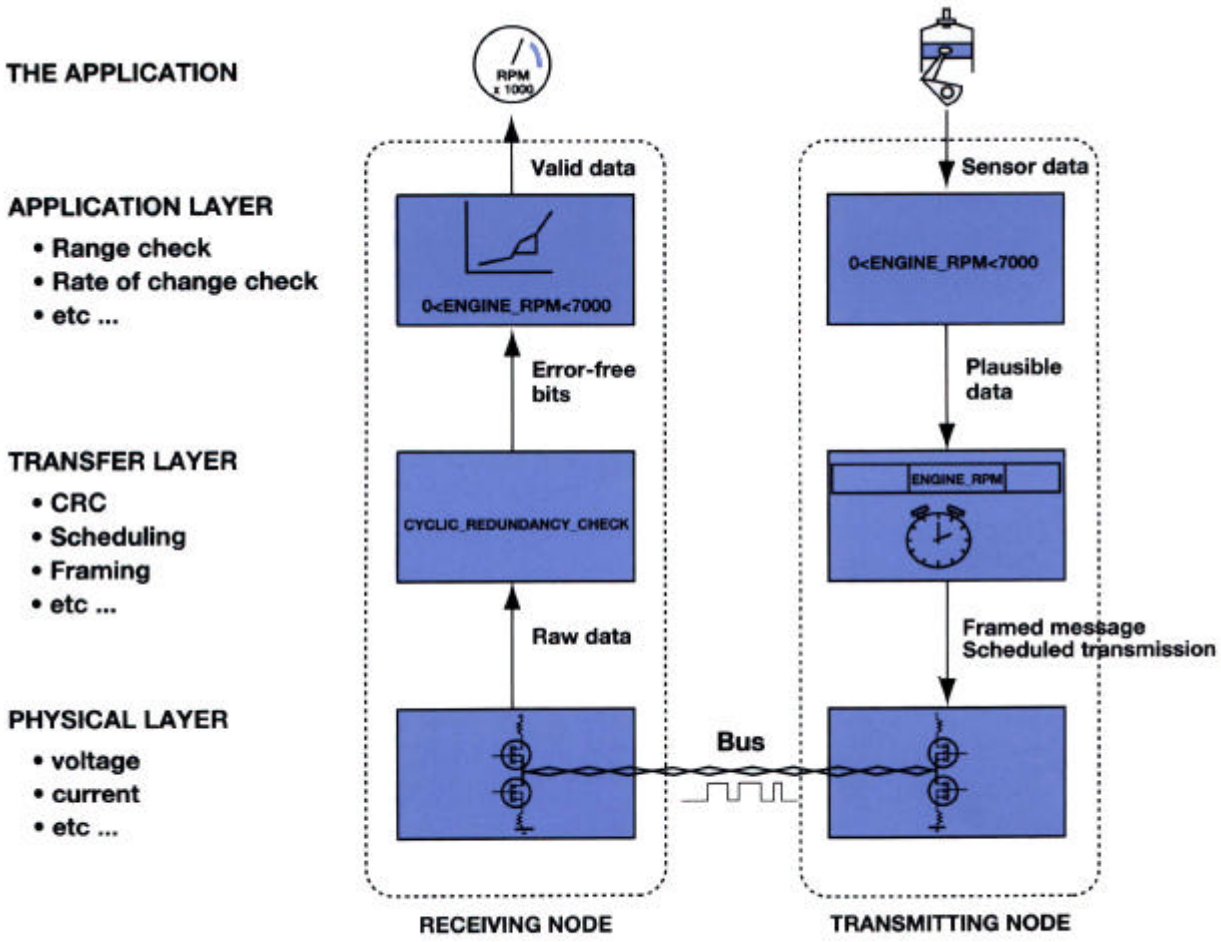


Figure 10. Three layer communications

3.4.5.11 Scheduling of message transmissions is important to ensure that the correct data rates or cycle times can be achieved. It is recommended that the transmission software drivers are not fragmented within the application code. It is better to have a centralized scheduling algorithm, based on one timing source in each ECU, which can be analysed for predictability.

3.4.5.12 Ensure that the relationships between the communications layers in each ECU are defined and adhered to.

3.4.5.13 It is recommended that the loading on the bus, message priorities, delay (latency) times and target processes are carefully analysed to ensure that the system operates reliably. The network should be robust against situations such as overloading and excessive delay, and avoid deadlocking.

3.4.5.14 To achieve the higher integrity levels, it may be necessary to augment the basic protocol with additional features such as:

- performing repeated transmissions of data to enable voting before an action is taken
- providing extra redundancy in the data field to offer error detection and/or correction codes
- time tagging each message.

3.4.6 On-board diagnostics

3.4.6.1 On-board diagnostics [1] should seek to:

- ensure that the vehicle is maintained in as safe and as legal a state as possible
- advise the driver of significant failures
- provide information to authorized personnel for prompt fault identification
- keep the customer mobile for as long as possible.

3.4.6.2 Diagnostic software should not be added as an afterthought: it should be included as part of the overall system requirements and design strategy. The objective of the diagnostic scheme should be clear.

3.4.6.3 It is recommended that on-board diagnostics and tools are available at the first release of software to assist the development process and calibration, and not just viewed as post-production support.

3.4.6.4 In principle, on-board diagnostic software can be used to detect three kinds of fault:

- nonexistent or incorrect sensor signals
- actuators not performing as intended
- processes within the host system not functioning as specified, where the malfunctions are not due to the processor itself.

3.4.6.5 If a detection strategy is to be effective it should:

- detect only “genuine” faults
- invoke “limp home” states in a safe manner
- provide a warning to the driver in a reasonable fashion
- store the fault information and make it accessible to repair personnel.

For these requirements to be met, it is essential to lay down clearly and understand the criteria for each part of the detection and diagnosis process.

3.4.6.6 In systems where the performance degrades as components age and wear, on-board condition monitoring can be employed to raise system integrity by identifying potential failures before they occur. The extent of monitoring will depend on the severity of the potential hazard, namely, the higher the integrity requirements the more comprehensive the condition monitoring that needs to be applied.

3.4.6.7 Detected faults should result in one or more of a range of actions:

- warn the driver
- store fault data till reset
- set “limp home” mode until repaired
- set local default states.

3.4.6.8 Consideration should be given to the mechanism of warnings, so that the driver is not unduly alarmed, or unnecessarily warned for minor faults. Such over-warning can lead to confusion or a nonchalant attitude towards heeding warnings.

3.4.6.9 The software for supporting off-board diagnostic tools that communicate with on-board systems should perform the following:

- transmit sensor data on demand
- transmit fault codes on demand
- allow adjustment of authorized on-board parameters by the diagnostic tool
- provide a sufficient level of protection against unauthorized adjustment.

3.4.6.10 An actuator may be controlling devices that are not directly visible to the system and whose states have to be inferred from other parameters. This may be compensated for by building a model of “expected” behaviour into the control algorithm.

3.4.6.11 Long term historical characteristics may be used in "expected" behaviour models to detect gradual deterioration.

3.4.6.12 Actuators may have parameters that can be used to aid health checking in a static mode (e.g. solenoid inductance).

3.4.6.13 All aspects of sensor and actuator installations (including connectors and wiring) may suffer from degradation and should be considered by a diagnostic strategy.

3.4.6.14 Consider the physics of sensors, how they interact with their environment and the ECU, and how they are influenced by ageing, wear and contamination.

3.4.6.15 Failure mode and effects analysis (FMEA) and further safety analysis studies should be undertaken following the full development of the diagnostic scheme.

3.4.6.16 Thoroughly understand the provisions of legislation, in the respective market areas, to enforce diagnostic requirements related to exhaust emissions (e.g. [28]).

3.4.7 System security

3.4.7.1 There should be protection against unauthorized access to software. Various methods are available [1, 4].

3.4.7.2 An alternative is to provide for the detection of tampering, for example as in US CARB OBD II legislation [28]. As a minimum, ensure that the evidence of tampering is clearly apparent.

3.4.7.3 It is important to ensure that inter-ECU communications cannot be modified by unauthorized means.

3.4.7.4 Be aware of the risks associated with "chipping" (reprogramming of electronic control systems, often by replacing the ROM storing the program and/or data, and most prevalent in engine management systems for performance gains), and take all possible steps to ensure that it is as difficult as possible for its perpetrators to gain access to and modify the software.

3.4.7.5 Be aware that "chipping" may cause a manufacturer to be legally liable in some world markets.

3.4.8 Fault management

3.4.8.1 A control system should be specified such that it degrades in a graceful manner in accordance with its integrity and availability requirements [4].

3.4.8.2 The criticality of components or functions of a control system often dictate the approach to fault management relevant to those components or functions. For a given function it may be necessary to adopt a fault tolerant approach; for another function, a fault recovery scheme may be appropriate. Give careful consideration to the fault management approach for every function and component of the system.

3.4.8.3 Fault management routines should be designed and demonstrated to respond within timing requirements for critical routines.

3.4.8.4 Fault migration between functions should be minimized by partitioning and recovery block schemes. Recovery from a fault management routine should be demonstrated, not assumed.

3.4.8.5 In multiprocessor systems, fault recovery action should be synchronized and prioritized, especially where communications failure is detected.

3.4.8.6 It is essential that recovery action is not inhibited by interrupts.

3.4.8.7 Many microcontrollers have on-chip watchdogs, which can provide powerful protection against some software errors, but will not protect against failure of all hardware components. If this is an unacceptable risk, then an off-chip watchdog or a separate monitor processor with separate timing circuits can be used.

3.4.8.8 Depending on the seriousness of a fault, a “limp home” state may be invoked to allow the vehicle to continue its journey in a safe manner but with some form of performance or functional restriction. It is advisable to maintain as much functionality as possible and not degrade the drivability unnecessarily.

3.4.8.9 Once the driver has been warned of a failure, and it can be determined that no action has been taken to rectify the fault within a reasonable time period, then designers could consider a degradation of facilities or performance that is effected at the next vehicle start up. This is particularly important where critical redundant components can fail transparently to the driver.

3.4.8.10 “Limp home” should only be invoked when there is no other alternative, and if it is going to significantly impact on the driver’s control of the vehicle, adequate notice of the event should be given, as far as this is possible.

3.4.8.11 Failure management should offer alternative sensor information or mechanical back up wherever possible, and all default definitions should be supported by a well-reasoned diagnostic strategy and objectives.

3.4.8.12 Safe states should be derived with reference to the system hazard analysis. The default action taken should be appropriate for the controllability category of the hazards involved.

3.4.8.13 Safety analysis of default states should consider potential driving situations, and how the default states, or combinations of default states, interact with those situations. Safety analysis should also consider the effects of system reset, so as to maintain a safe state.

3.4.8.14 Some possibilities for handling predicted component failure are:

- a policy of service replacement
- standard component performance test, comparing with original data
- multiple channel, redundant systems or components
- inference from other sensors
- inference from start up data derived from setting components to known states
- monitoring trends.

3.4.8.15 In contrast to open-loop systems, and in common with all digital systems, closed-loop control systems work very reliably, but tend to fail dramatically. Therefore, it is important to emphasize the design and validation of the failure management mechanisms.

3.4.8.16 There is a high risk of "fault masking" in microprocessor based closed-loop control systems. This occurs when the failure mode management is so effective that the driver fails to recognize the presence of a fault. The use of condition monitoring and appropriate warnings are recommended.

3.4.8.17 It is recommended that a combination of failure management techniques are used for handling failures in closed-loop systems, including switching to open-loop operation.

3.4.9 Design for verification and validation

3.4.9.1 Design for verification and validation [6], bearing in mind the capabilities and limitations of the available software tools, test rigs and in-vehicle aids.

3.4.9.2 For a design to be comprehensible and maintainable, it should be constructed in a modular and structured manner.

3.4.9.3 The software may have to be organized into sufficiently small modules to achieve the necessary test coverage.

3.4.9.4 The definition of test coverage should be considered carefully.

3.4.9.5 It should be possible to trace a module throughout a hierarchically described design; and to be able to relate the design to its specification.

3.4.9.6 Design reviews should be carried out to establish that the design process is producing the required output at each relevant stage.

3.4.9.7 Reviews of the detailed safety analysis should be carried out as part of the project design review.

3.4.9.8 The design of an integrated system, with a range of integrity levels, should aim to segregate functions of different integrity levels. Otherwise, the highest level of integrity should prevail for the whole system.

3.4.10 Tools and techniques for design

3.4.10.1 Consider the use of structured analysis techniques, supported by appropriate CASE tools, to help construct a suitable design from the software requirements.

3.4.10.2 The networking of CASE tools can be a significant advantage for project teams, and should be considered to aid the configuration management of design material.

3.4.10.3 A CASE tool should be capable of good graphical support for the preferred methodology and be well supported by a good data dictionary.

3.4.10.4 Carefully assess CASE tool suppliers for stability, long term commitment and support services.

3.4.10.5 Assess new tools to ensure that they support and integrate into the existing development process rather than hinder it. Cumbersome tools and processes tend to be circumvented.

3.4.10.6 CASE tools cost more than the purchase price alone. Allow for maintenance, training and integration of the tool into the development process.

3.4.10.7 Do not take the decision to switch design methodologies lightly. If possible assess alternative methodologies in parallel with existing methods on a current project.

3.4.10.8 Before beginning a design, the rules and naming conventions for the use of the CASE tool should be defined, agreed and documented.

3.4.10.9 The true power of a toolset can only be realized when these tools are integrated into a common environment.

3.4.10.10 It is currently recognized that a high degree of assurance in a design may be obtained by using formal mathematical methods, and by providing all the supporting proofs that have been checked with a proof checker.

3.4.10.11 Use simulation tools to predict communications network performance during the analysis and design phases. Compare actual performance against the prediction so that future simulations can be made more accurate.

3.4.10.12 To meet integrity requirements defensive techniques should be used where appropriate.

3.4.10.13 All system resources used by a high integrity level function should be provided up to the same standard as that function. Note that redundancy and diversity may be applicable to reduce the integrity requirements of individual parts.

3.5 Programming

3.5.1 Codes of practice

3.5.1.1 The choice of a programming language itself should be strongly influenced by the way that language is used and by the availability of tools that exist to support it [1, 6].

3.5.1.2 It is recommended that the use of programming directly in machine code should be discouraged. Machine code is difficult to follow and the potential for unidentified mistakes remaining unresolved is higher than when a high level language is used. Alterations and changes to the program are also more difficult to verify at the machine code level.

3.5.1.3 Documented company and project standards should be used to give guidance on:

- source code layout
- documentation style
- language usage
- compiler usage
- design best practice
- naming conventions.

3.5.1.4 Code commenting is an integral part of any well prepared program and is vital for code verification. Care should be taken to ensure that comments consistently match code, particularly when changes are made.

3.5.1.5 Consideration should be given to using a restricted subset of a programming language to aid clarity, assist verification and facilitate static analysis where appropriate.

3.5.1.6 Changes to the software should be approved, fully documented and kept under a change control system.

3.5.1.7 Due to the intangibility of the final program code, the production of supporting documentation is essential.

3.5.2 Verification and validation of code

3.5.2.1 The documentation for the software modules should support the planned verification activities.

3.5.2.2 Before dynamic testing begins the code should be reviewed in accordance with the software verification plan to ensure that it does conform to the design specification.

3.5.2.3 Code reviews and/or walkthroughs should be used to identify any inconsistencies with the specifications [6].

3.5.2.4 All production and verification tools should be justified in relation to the integrity level of the software they are being used to analyse.

3.5.2.5 Compilers should conform to an international standard for the language definition (where one exists), and if possible be validated for the target computer.

3.5.2.6 Static analysis is effective in demonstrating that a program is well structured with respect to its control, data and information flow. It can also assist in assessing its functional consistency with its specification.

3.5.2.7 Product metrics [5] are a measure of some attribute of an output of the software development process (e.g. code). The attributes that can be measured include:

- complexity
- maintainability
- modularity
- reliability
- structure
- testability.

3.5.2.8 Product metrics can be used to assess the quality of the output. For example, a highly complex code item with poor structure is more likely to contain errors than a simple item with good structure.

3.5.3 Programming tools and techniques

3.5.3.1 If possible use automated tools to check that company and project programming standards are being adhered to.

3.5.3.2 A checklist is useful in examining a program for common errors.

3.6 Testing

3.6.1 General

3.6.1.1 The limitations of testing software based systems should be properly appreciated. The number of possible paths through a complete program means that even in the strictest test regime only a proportion of the paths will be executed.

3.6.1.2 Testing forms one part of the overall verification and validation activities (see Figure 11). It should be remembered that quality can never be tested into a product.

3.6.1.3 The purpose of testing is to discover errors, not to prove correctness.

3.6.1.4 Plan and document a test strategy, showing an intelligently selected test data set complete with justifications for the tests [6].

3.6.1.5 A software test plan should, for each functional requirement state the:

- input conditions
- output conditions
- acceptance criteria
- limits of operation
- likely failure modes.

3.6.2 Dynamic test

3.6.2.1 The dynamic characteristics of the unit should be tested for compliance with the timing and response requirements, as well as the functional requirements.

3.6.2.2 Consider both in-range and out-of-range values when selecting test cases.

3.6.2.3 Test data should be derived from the software specification of the appropriate level, for example, requirements, design or module. It should remain consistent and traceable to that specification throughout.

3.6.3 Integration test

3.6.3.1 Plan integration testing activities in advance. Do not test in an *ad hoc* fashion, ensure each test has a purpose defined in the plan. It should remain the responsibility of the vehicle manufacturer to plan, perform and approve system integration tests [1].

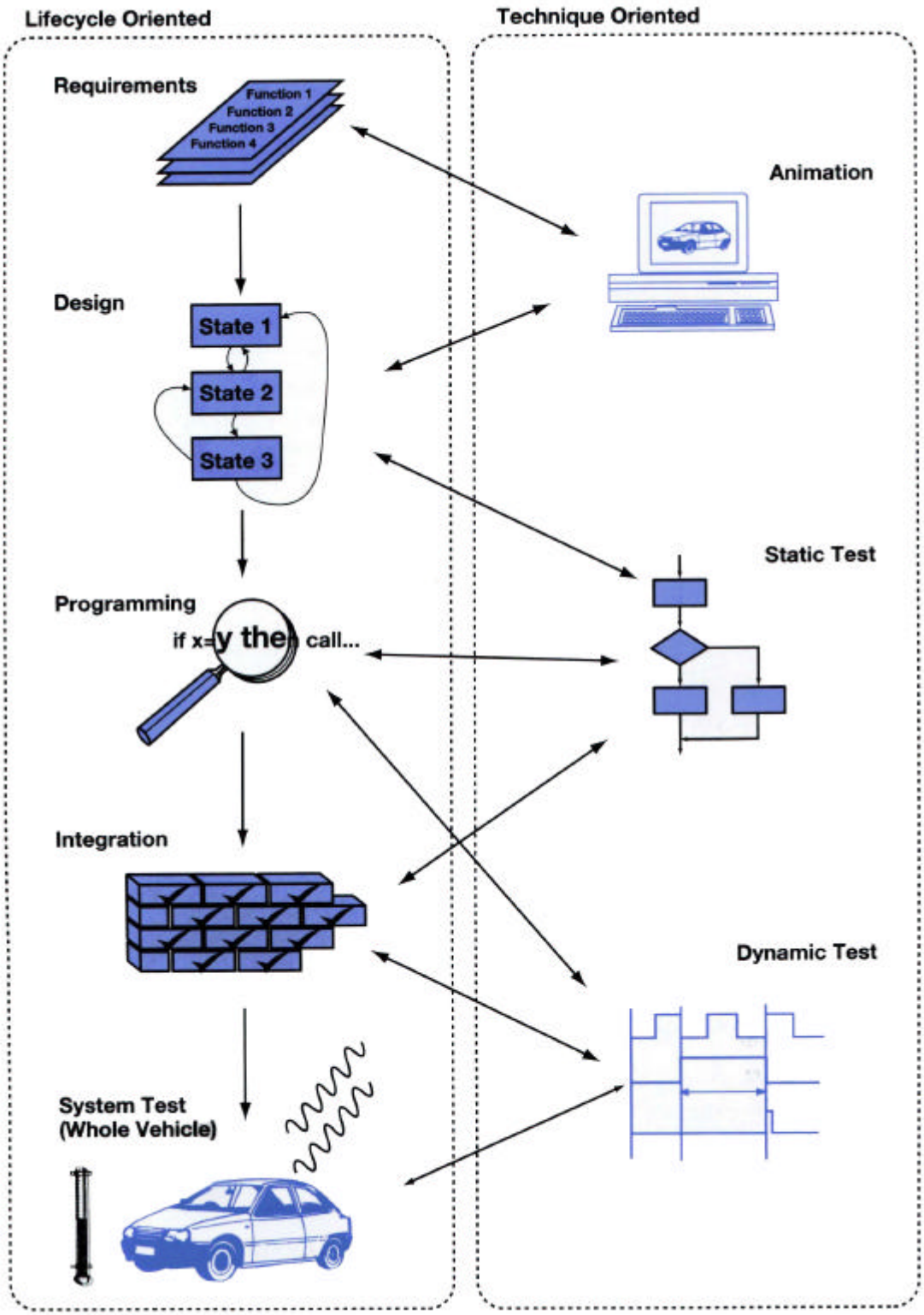


Figure 11. Relationship between testing methods

3.6.3.2 The integration test plan should define the levels of testing to be performed, for example:

- software modules integrating into complete software
- complete software and target hardware integrating into an ECU
- individual ECUs integrating into a vehicle system.

3.6.3.3 Assess individual ECUs for compliance with their interface specifications before attempting to perform system integration tests.

3.6.3.4 Testing the entire software aims to ensure that all modules interact correctly. Coverage analysis should be used to ensure that the testing adequately exercises the code structurally.

3.6.4 System test

3.6.4.1 Each module should be tested separately and then, after integration, tests should be made on the entire system.

3.6.4.2 Where possible, system tests should be performed using fully integrated, production intent software and hardware, operating in the real environment under realistic workloads. Certain tests may require simulated environments.

3.6.4.3 Confirm the effect on the system when operated under abnormal workloads (stress testing).

3.6.4.4 The system testing process should be based on the requirements. Coverage analysis should be used to show that the test cases are adequate to demonstrate compliance with each requirement.

3.6.4.5 Test cases should be selected with the objective of uncovering errors. Previous experience should be used where possible (error guessing).

3.6.4.6 Specific testing should be performed to determine whether or not the safety requirements are fulfilled.

3.6.5 Tools and techniques for testing

3.6.5.1 Black box testing can be used to compare the high level design against the requirements. It ignores the implementation details of the software itself and test cases are simply chosen to demonstrate that each software module complies with its requirements. Typically requirements coverage analysis is carried out with black box testing.

3.6.5.2 White box testing is similar to black box testing except that it uses knowledge of the internal program structure of modules. Test cases are chosen to ensure that branches, statements or conditions in the code are tested. The type and extent of coverage should be defined prior to white box testing. See Table 3.

3.6.5.3 Error seeding may be used to demonstrate the effectiveness of error detection and recovery routines. This can give an indication of the detection rate of the testing process adopted.

3.6.5.4 Equivalence partitioning [6] can be used to reduce the test space by reasoning logically that not all possible combinations of test data are necessary.

3.6.5.5 Boundary value analysis is a useful technique for the efficient testing of software, by selecting test inputs just above, just below and on threshold values for key parameters.

3.6.5.6 It may be useful to compare the results of testing the full implementation against the results provided by animating a formal specification (if possible) with the same test data.

3.7 Product support

3.7.1 Off-board diagnostics

3.7.1.1 Off-board diagnostic systems [1] should have the ability to:

- retrieve data from on-board systems
- apply on-board tests
- diagnose relevant faults that cannot be detected by on-board systems.

3.7.1.2 If a test routine invoked by a service bay tool can affect the normal operation of the system, then it is essential that the unit is configured back to normal operation before ending the diagnostic session. Exiting the diagnostic session, either by controlled exit, power down or communications failure, has to return the unit to normal operation.

3.7.1.3 Service diagnostics should be designed to assist both expert and inexperienced technicians. For inexperienced technicians it may be appropriate to lead them down a fault tree until the failed component is identified (lowest replaceable unit).

3.7.1.4 If using the communications network as a medium for performing diagnosis, it is also necessary to consider failure of the communications network itself.

3.7.1.5 Access to appropriate information may have to be provided for non-franchised repairers and for roadside repair throughout the life of the vehicle.

3.7.1.6 Technical information proprietary to the manufacturer requires support in terms of training and backup services.

3.7.1.7 Elaborate off-board diagnostic systems are not a substitute for engineering reliability and robustness into the vehicle itself.

3.7.1.8 Diagnostic tools should be able to identify the lowest replaceable unit (LRU). The tool can also be used to remind the technician not to attempt to repair an LRU.

3.7.1.9 Information on warranty claims should be available at all levels in both the vehicle manufacturers' and the component suppliers' organizations, providing a feedback mechanism for design decisions.

3.7.1.10 For bus based diagnostic systems, a common protocol and common message definition should apply to all systems connected to the bus. Where possible use an international standard such as the draft ISO 14230, *Keyword Protocol 2000* (KWP 2000) [29].

3.7.1.11 Consideration should be given to the use of event recorders that can be fitted to vehicles to store dynamic parameters to assist in locating intermittent faults.

3.7.2 Software maintenance

3.7.2.1 Post-production modification should be performed with the same level of expertise, automated tools, planning and management as the initial development of the system. It is recommended that the procedures used for the product development are applied to post-production modifications [6].

3.7.2.2 The replacement of a safety-related system by a new version should require the same levels of authorization and assessment, and follow the same procedures, as applied to the original system.

3.7.2.3 Ensure that user documentation contains an appropriate warning regarding unauthorized modification of vehicle systems, especially software, and if appropriate, a suitable disclaimer for liability in the event of unauthorized modification.

3.7.2.4 Manufacturers should provide the service organizations with the means for assuring customers that the software in their vehicles conforms to the intended specification.

3.7.2.5 Management should have a policy on how long a product has to be supported. Each product will have an associated development environment, computer hardware and software, which was used to develop it. When changing or upgrading a development environment, consideration should be given to the effect of this action on the products developed using it.

4. Software quality planning

4.1 Management responsibilities

4.1.1 There should be a single individual with overall and ultimate responsibility for the project.

4.1.2 Define and implement a Quality Management System based on a software development lifecycle (see Figure 2) and ensure that it is effective. See Table 3: note that the requirements of ISO 9001 [10] are appropriate for integrity level 0, and further steps are needed for integrity levels above zero.

4.1.3 Provide adequate resources for the scale of project planned in view of the recommendations in these Guidelines.

4.1.4 Organize the resources necessary to support a degree of independence appropriate to the integrity level for both verification and validation, and assessment activities.

4.1.5 Ensure that staff, suppliers and subcontractors are competent to perform the tasks assigned to them.

4.1.6 Ensure that there is an appropriate balance of expertise to undertake the nature of the project, covering engineering disciplines and specializations such as:

- control systems
- software engineering
- electronic engineering
- communications
- mechanical engineering
- human factors engineering.

4.1.7 Ensure that there is an appropriate balance of staff, so that:

(a) Opportunities exist for trainees to acquire experience from mature members of the team.

(b) Continuity of expertise is maintained in anticipation of staff attrition.

4.1.8 Managers should be aware of individual and team limitations and should not make unreasonable demands in terms of technical capability or timing.

4.1.9 Managers should actively participate with engineers to resolve conflicts of commercial interest and safety.

4.1.10 Provide structured training.

4.1.11 Management is heavily dependent on documentation to monitor accurately the progress of the software development within a project. Management should give emphasis to its generation, maintenance and review.

4.1.12 Software is a major contributor to quality and reliability performance and should not be dismissed by the vehicle manufacturer as a supplier's problem.

4.2 Education and experience

4.2.1 The staff employed should:

- have the relevant theoretical knowledge
- have the practical experience appropriate to their role
- be aware of current practice and available technologies
- be familiar with applicable legislation, regulations and standards, including product liability issues
- have an awareness of a professional code of practice for engineers and managers, such as in the IEE *Safety Related Systems* brief [18] or in the Engineering Council's *Guidelines on Risk Issues* [19].

4.2.2 The Quality Assurance function of both supplier and vehicle manufacturer should include personnel from a software background, or with the necessary training to be able to create and monitor the implementation of a Software Quality Plan.

4.3 Human factors in software development

4.3.1 Introduction

4.3.1.1 This section relates to the human factors that arise when people are used to design and develop software, particularly for safety-related applications [8]. The aim is to ensure that products, and the processes used to create those products, match the capabilities, limitations and needs of the people involved.

4.3.1.2 It is common for projects to make use of teams of people. The social psychological considerations in the workplace that result from human behaviour should be regarded as important contributors to project success.

4.3.1.3 The issues identified in this section should be considered at the project planning stage, and the decisions that result should be documented where appropriate.

4.3.2 Teams and organizational structure

4.3.2.1 The organizational structure of a project team should match the design and development as far as is practicable, and allow for appropriate independence of verification and validation, and assessment.

4.3.2.2 There should be a continuous reassessment of tasks with frequent interaction between personnel.

4.3.2.3 Individual team members should be encouraged not only to take responsibility for their own tasks, but also to share responsibility for the overall project.

4.3.2.4 The setting of project goals should involve some degree of group participation.

4.3.2.5 Decentralized communication structures should be advocated such that information is freely distributed between all team members. However, document control has to be maintained to prevent the use of out-of-date information.

4.3.2.6 The product of a team's efforts should be seamless in that it should appear to be the work of one individual rather than several.

4.3.2.7 Project standards should aim for uniformity of style.

4.3.3 Individual differences and job design

4.3.3.1 There should not be an overreliance on the results of psychometric tests in the selection of software engineers, as there is no evidence that they successfully predict engineering competence.

4.3.3.2 The selection process for software engineers should address behavioural factors and personality effects, rather than relying on intelligence and qualifications alone. The ability to work together in groups is a key factor.

4.3.3.3 Software development environments should allow for individuals' opinions to be heard and foster the realization of personal and collective goals.

4.3.3.4 Reward systems should be tailored to individuals. Money should not be interpreted as the primary motivator of individuals who function in highly technical and/or creative environments.

4.3.3.5 Reward structures should be effective so that people are rewarded in a way that matches their individual expectations and aspirations.

4.3.3.6 Career opportunities should exist to promote competent individuals within the technical area. This helps to maintain key technical skills.

4.3.4 Human error management

4.3.4.1 It is virtually impossible to prevent human errors from occurring, therefore provision should be made in the development process for effective error detection and correction; for example, reviews by individuals other than the authors.

4.3.4.2 It is recommended that a fear free but responsible culture is engendered for the reporting of issues and errors.

4.3.4.3 The communication of information regarding errors to design and development personnel should be as clear as possible. For example, errors found during reviews should be fully recorded at the point of detection.

4.3.4.4 Project goals should be clear, resulting in easier understanding of the goals by all team members. This can aid in the formulation of subgoals at lower levels.

4.3.4.5 Design and development tasks that rely on the recall memory of individuals should be minimized in favour of recognition dependent tasks such as consistency checks, rules and patterns. For example, checklists can help to ensure completeness, and menus are less prone to errors than command lines.

4.3.4.6 Structured methods and modularization will aid the development of internal semantics [30]. This will improve understanding of the problem and help reduce errors.

4.3.5 The physical environment

4.3.5.1 Consider the effects of the working environment on the ability of the project team to avoid design errors.

4.3.5.2 The conditions that are most likely to form an effective working environment include:

- room temperature: 20 – 21 ^{o}C in winter, 20 – 24 ^{o}C in summer
- humidity: > 30% in winter, 40 – 60% in summer
- drafts: < 0.5 ms^{-1}
- ventilation: 30 m^{3}/hour per person of fresh air and/or air conditioning
- ambient noise: < 40 – 45 dBA.

4.3.5.3 To combat the problems of glare, VDUs should be placed at right angles to windows, with light fixtures being positioned parallel to, and on either side of, the operator-screen axis.

4.3.5.4 The recommendations of the European Directive 90/270/EEC 29 May 1990 (Display Screen Equipment) that came into force on 1 January 1993 [31] should be adhered to.

4.4 Quality assurance

4.4.1 Standards and accreditation

4.4.1.1 The basic requirement is for organizations involved in the development and procurement of software to operate a Quality Management System capable of meeting the requirements of ISO 9001 [10] as assessed under the guidelines in ISO 9000-3 [11] (e.g. TickIT [12]).

4.4.1.2 The Quality System operated by organizations from whom software is procured should be considered when selecting suppliers.

4.4.2 Checklists

4.4.2.1 Checklists are a simple, concise but effective method of establishing and verifying completeness. They should be applied at defined milestones in the development lifecycle.

4.4.2.2 Checklists take many forms. The essence is the establishment of a list of predefined topics and questions against which an objective assessment or measurement can be undertaken.

4.4.2.3 It is recommended that the completed and dated checklists are preserved as part of the project documentation.

4.4.3 Assessment of compliance

4.4.3.1 The recommendations given in both these Guidelines and the supporting reports can used as the basis of a checklist. All of these recommendations can be transformed into a question. For example, this can be accomplished by encapsulating the recommendations with a phrase such as:

- Have the following factors been considered … ?
- Does the design meet the requirements for … ?
- Has … been planned/undertaken/provided/completed/considered/… ?

4.4.3.2 A record of the compliance against each recommendation may simply be by selection of a token or keyword from a list, such as:

Yes	Refer to …	No
Completed	Not applicable	No evidence
Provided	Not assessed	Not provided.

4.4.3.3 Below is an example of a *pro forma* that could be used to assess compliance and record the results:

Guideline Reference	Topic Title	Compliance	Justification and/or Comment

4.4.3.4 A measure of the degree of compliance to the Guidelines can be obtained from the completed *pro forma*. Note that not all recommendations carry the same weight. Justifications should be recorded where individual recommendations are not adopted.

4.4.4 Changes during production

4.4.4.1 Production runs in the automotive industry are typically much longer than those in the semiconductor industry; consequently service support is a major issue [3].

4.4.4.2 Masked microcontrollers are partially custom devices that are tested generically. It is not unknown for a device to undergo a process change and pass generic testing, but fail in the application domain.

4.4.4.3 Users of microcontrollers should select manufacturers who notify of process changes as they are likely to control the change well. They should test advance samples at high and low temperature, under temperature change, high and low supply voltage and for EMC. They should also check restart of oscillation and any affected inputs and outputs for transient immunity.

4.4.4.4 Problem containment depends on batch traceability. Semiconductor manufacturers should provide batch traceability of all products. Distributors of semiconductors and suppliers of subassemblies should not impede traceability, as root cause analysis depends on it.

4.4.5 Software process metrics

4.4.5.1 Software process metrics are measures that provide information about the performance of the development process itself [5]. There is a dual purpose to the collection of such data:

(a) To provide an indicator as to the ultimate quality of the software being produced.

(b) To assist the organization in continuously improving its development process by highlighting inefficient or error-prone areas of the process.

4.4.5.2 It is recommended that at least core process metrics are collected, analysed and reviewed. The basic metrics recommended are:

- estimated duration of each task
- actual effort expended on tasks
- number of defects detected.

4.4.5.3 When measuring data on defects, it can also be extremely effective to record the source of the defect (lifecycle phase) and the phase in which the defect was detected. Using this data, it is possible to assess the effectiveness of the development process at different phases. If coupled with data from item reviews, areas of concern in the development process can be quickly identified and corrective action taken.

4.4.5.4 The analysis and reviews should be performed in a timely manner in order to facilitate effective feedback. A number of reference books [32, 33] are available that give a more detailed description of process metrics and how to collect the information.

4.4.5.5 Other metrics may be found useful, but this will depend on the organization. There are no "correct" metrics. Initiating a metrics programme can be problematic, so for an organization not currently collecting software metrics:

(a) Understand the objectives of implementing a metrics programme.

(b) Identify those metrics that aim to achieve those objectives within the organization.

(c) Do not be too ambitious initially — start with a number of simple metrics (such as those listed above).

(d) Do not concentrate on a single metric — this can distort the collected data.

(e) Analyse data in a timely manner so that feedback can be effective.

4.4.5.6 Other aspects that should be borne in mind:

(a) It is not advisable to allow metrics to be used to measure performance of individuals.

(b) Involve staff in the choice of metrics and the setting of goals.

(c) Review the metrics information at regular intervals.

(d) Provide regular feedback to the team and discuss problems and performance issues.

(e) Data collection methods should be as simple as possible, preferably automated, to minimize the effort involved in collection.

(f) Data should be presented graphically if possible for ease of interpretation, particularly since trends will often be more relevant than absolute values.

4.5 Documentation requirements

4.5.1 There are many recommendations within these Guidelines associated with the generation and use of documentation.

4.5.2 Due to the intangibility of the final program code, the requirements to produce supporting documentation are inseparable from the principles of software engineering. Many generic standards and guidelines exist that define documentation requirements for all aspects of the software development lifecycle [16, 17, 34] and some of the larger automotive companies have developed and are using their own standards.

4.5.3 In view of the activities of the ISO/TC22/SC3/WG11, "Automotive Electronic Control Systems — Technical Documentation" working party [15] it is inappropriate to define a new or alternative documentation regime at this time. The adoption of the new ISO documentation standard for automotive electronic control systems should be considered when available.

4.5.4 There is an accepted principle associated with all forms of safety related engineering that the greater the risks, the greater is the need to supply information and give evidence of robustness. The integrity level of the system will dictate the amount and the form of the information to be provided. See Table 3.

4.5.5 A Software Quality Plan should be established describing the overall approach to software development including a detailed lifecycle definition, the development environment, assurance controls, tools, standards, guidelines and the configuration management procedures used.

4.5.6 As certain aspects of verification and validation will depend almost entirely on documentation, it is vital that it is both concise and complete [6].

4.5.7 The format and nature of the documentation produced will depend upon the size of the project, the integrity level and company policy. The information, however, should always be clearly identifiable.

4.5.8 The handling of documentation should be properly regulated, both to control access to it, and to control changes.

4.5.9 As the material involved may have been produced at different times, reside at different locations and on different media, it is recommended that an overall index is maintained that identifies and links it to a single point.

4.5.10 The period of time for which documents are to be retained should be laid down, procedures defined for their control during that time, and plans made for their final disposal.

4.6 Subcontracting

4.6.1 Introduction

4.6.1.1 As well as placing a contract for the purchase of systems or components, it is important that for projects containing software to be developed by another party, a formal project-specific contract is agreed before the development begins.

4.6.1.2 These recommendations are intended to help engineers, managers, and purchasing departments to understand the issues involved in subcontracting products for vehicle use that contain embedded software [7] (see Figure 12). They are not intended as a definitive legal treatise, and anyone who wishes to compile a software contract should seek appropriate legal advice.

4.6.2 Definitions

4.6.2.1 **Purchaser** The legal term for the party placing a contract.

4.6.2.2 **Vendor** The legal term for the party on whom the contract is placed.

Software quality planning (continued)

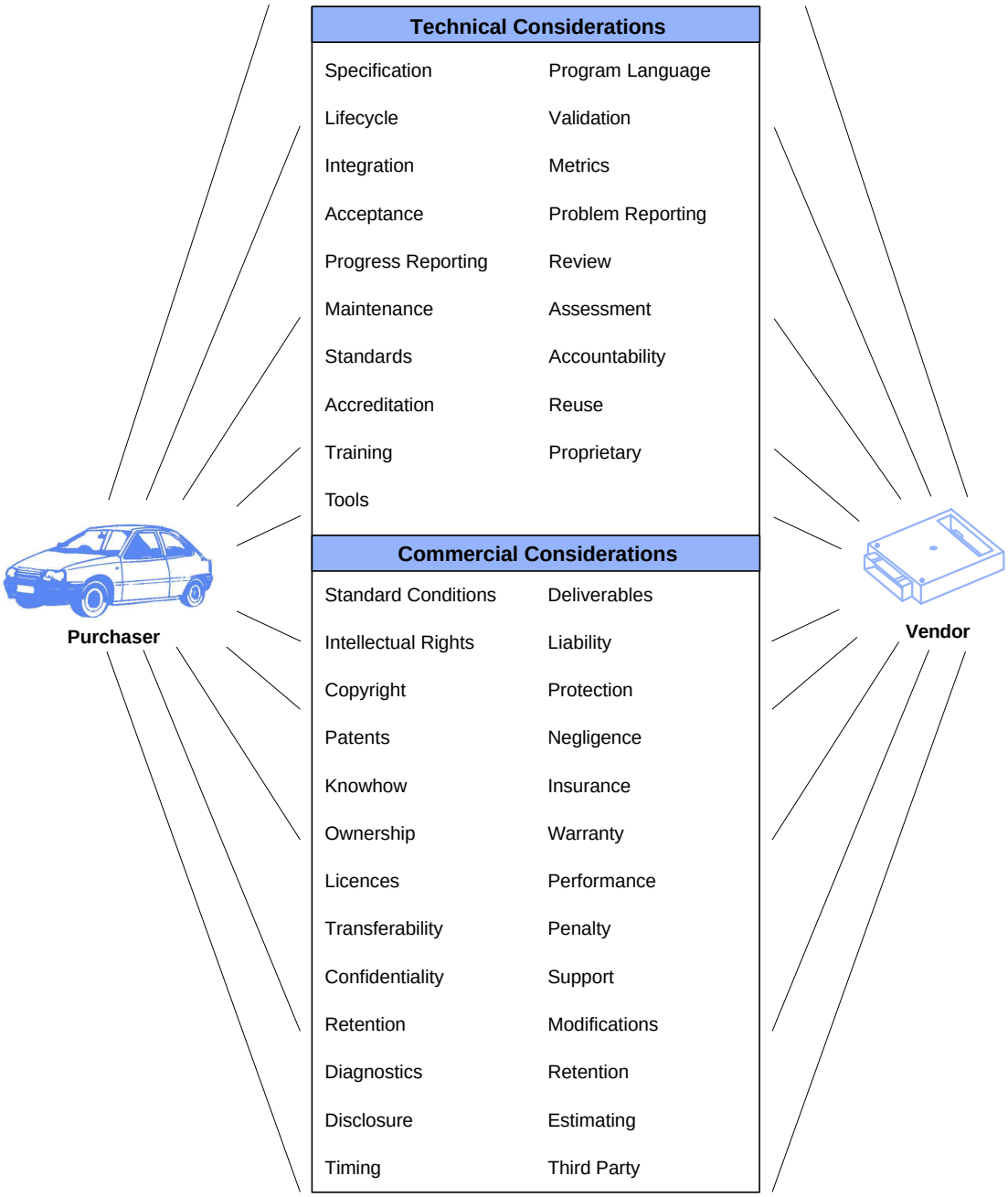


Figure 12. Software subcontracting topics

4.6.3 Technical considerations

4.6.3.1 Good communications between the contracting parties is vital for the creation of an effective software contract. Oral communications alone, even if minuted, are inadvisable (even though minutes can be a legal document). A formal Invitation to Tender, incorporating a functional specification, defining special constraints or requirements, should be compiled by the Purchaser and issued to the Vendor.

4.6.3.2 Contracts, like requirements specifications, should be unambiguous. Establish a glossary of terms used, with their definitions as they are applied in the contract.

4.6.3.3 A contract to develop a product involving software should not be seen as a punitive document aimed at extracting penalties when the project goes wrong. Instead, it should seek to ensure that the project does not go wrong. Deadlines and penalties are irrelevant if the product is not fit for launch.

4.6.3.4 A software contract should clearly state what the product of the project is expected to do, and what it is expected not to do. It should cover topics such as:

- required features with performance constraints
- any legislative constraints on the product
- parameters that are to be visible to diagnostics, including priority structure if possible
- interfaces to other on-board systems and off-board tools
- failure management policy, including default states and warnings
- reliability requirements: definition, criteria and targets.

4.6.3.5 The software contract should lay down procedures for agreeing a detailed requirements specification, which should be given greater weight of authority than any descriptions in the contract. It should also include procedures for handling ambiguities, inconsistencies, query rectification and disputes. The Purchaser should consider defining requirements capture and/or analysis tools.

4.6.3.6 Adoption of an agreed product development and quality lifecycle plan is beneficial to both parties to the contract. It should not be treated as a punitive device.

4.6.3.7 The development lifecycle should lay down agreed milestones; simply enforcing an end of project deadline, possibly with penalties for late delivery, may result in problems being hidden from the Purchaser.

4.6.3.8 It is in both parties' interest to agree the correctness of the requirements specification at the point in the lifecycle that offers least risk to them both.

4.6.3.9 In order for the Vendor to understand his responsibility for quality and safety, risk assessment procedures should be defined in the contract. A preliminary safety analysis should be performed by the Purchaser and the integrity level specified in the contract. The contract should also make provision for a detailed safety analysis.

4.6.3.10 Both parties should be fully cognisant of the required development tools for the product (e.g. which product, version, add-ons). Agree a mechanism to ensure that if accredited tools are required, only those tools are used.

4.6.3.11 It is essential where proprietary software is specified that commercial, timing and technical constraints are understood, agreed, and documented by both parties, at both the Invitation to Tender and contract stages.

4.6.3.12 The contract should address verification, validation and timing issues, particularly where reuse of software is proposed.

4.6.3.13 Ideally, one route of communication for software issues and authorization should be identified in the contract.

4.6.3.14 Any specific competence requirements for key project staff should be identified in the contract.

4.6.3.15 It is in the interest of both parties to define problem reporting and corrective action procedures in the contract. Human factors are the biggest constraint to resolution of problems in a proper manner in software contracts.

4.6.3.16 Progress reporting criteria and metrics, as well as procedures, should be laid down in the contract, so that concerns are identified at the earliest point.

4.6.3.17 It is very important that acceptance criteria, based on joint reviews, are contractually defined for each milestone.

4.6.3.18 Maintenance and support should be treated separately from warranty provisions. They should be planned for, and be contractually defined, including the level and type of staff, period of support, and man-hours to be provided for. These issues relate to changes arising out of experience in the field, not warranty-type "fixes".

4.6.3.19 A quality culture such as Total Quality Management is not a substitute for a properly constituted contract.

4.6.3.20 Agree issues relating to second and third party audit, support and modification at the earliest stage possible, including confidentiality matters, and ensure that any such requirements are defined in the contract.

4.6.3.21 Ensure that requirements for retention of records are fully laid down in the contract, including stipulated time period and media. There is often a requirement under legislative provisions such as automotive type approval for retention of records for at least ten years after the end of production.

4.6.4 Commercial considerations

4.6.4.1 In general, English law tries to adopt a balanced stance: not only are the terms of the contract considered, but force is also given to what is fair and reasonable. More importantly, perhaps, is that force is not given to what the Courts do not deem fair and reasonable. Overseas law may apply different criteria or interpretations. For example, Scottish law differs from English law in many respects, and many English lawyers do not purport to be able to offer advice regarding any law other than English.

4.6.4.2 Ownership

(a) Be clear who owns what items when material is free-issued to a Vendor, or purchased by the Vendor for the purpose of carrying out the project.

(b) It is in the Purchaser's best interest to ensure that purchasing terms for proprietary software and protocols are complied with by all contractual parties; and, should a Vendor be prosecuted for licence infringement, a protection or contingency mechanism may be extremely important for the Purchaser. The contract should not place any liability on the Purchaser for the Vendor's noncompliance.

(c) Terms and conditions for transfer of licence agreements should be fully understood by all parties concerned, and if the Purchaser's software is involved, the terms should be covered in the contract.

(d) Advice should be taken from a patent lawyer or legal consultant regarding copyright, right to replicate and patents in respect of software [35, 36]. It is important to be aware that under English law only functions that the system is carrying out can be patented, not the actual code and algorithms. However, they can be copyrighted. It is also vital to be aware that a significantly greater risk regarding patent infringement exists elsewhere, especially in the US, where there is a tripled penalty in the courts for knowingly infringing a patent.

(e) Define in the contract the responsibility for negotiation with a copyright or patent holder in the event that the contract involves using a third party's copyrighted or patented material.

(f) Advice should be taken from a legally qualified person regarding intellectual property rights [35, 36].

(g) If knowhow is sensitive, use modularization of the project and multiple Vendors, to ensure that a single Vendor cannot obtain a complete view of the project. Ensure that the contract imposes a procedure for controlling the passing of all information, in writing to the Vendors.

(h) It may be desirable for critical or sensitive software to be subject to a copy being lodged with a suitable escrow agent. This may be relevant if there is a perceived risk of a Vendor's change of ownership, business failure or future legal action.

4.6.4.3 Warranties and assurances between Purchaser and Vendor

(a) Fitness for purpose is most likely to be solely the responsibility of the Purchaser; this should be a convincing argument for proper specification and quality procedures to be practised by the Purchaser, and imposed on the Vendor.

(b) Performance bonds and penalty clauses are counterproductive for software contracts, and generally cannot be recommended practice. If proper procedures are contractually agreed, there should be no need of such punitive measures.

(c) Some Purchasers may require CVs of staff assigned to the project as an assurance of competence. This is especially the case where high integrity software is involved. This should be specified in the contract if it is required, along with a mechanism to ensure confidentiality.

4.6.4.4 Liabilities

(a) Liabilities should be defined as far as possible in the contract, especially where proprietary software is concerned. However, the ultimate liability is considered to lie with the Purchaser for ensuring that the product is fit for the marketplace, and for ensuring that contracts require adherence to proper practices.

(b) It is the responsibility of the Purchaser to ensure that both he and the Vendor take all the steps recommended in these Guidelines, and comply with standards such as ISO 9001 [10], as appropriate.

(c) Liability for breach of confidence can be incorporated into the contract, but the best protection is via close control of the passage of information and its authorization.

(d) The Purchaser may need the right to pass access rights to third parties, for example, diagnostic data to roadside repair organizations.

4.6.4.5 Deliverables

(a) Software cost estimates are a well recognized area of difficulty. Staged contracts are a significant help in obviating problems, by identifying them at an early stage before the project has gone seriously wrong.

(b) The use of a lifecycle plan aids in understanding where resources are expended in software projects, especially if each stage has to be approved before the start of the next stage. This is only effective if the deliverables from each stage are clearly defined in the contract.

(c) All requirements for transfer of data and documentation, magnetic, electronic and paper should be defined in the contract, along with the number of copies, personnel receiving copies, and any requirements for retention or destruction of residual data.

4.6.4.6 Standard contracts should be treated with caution where software is involved. There are model contracts [37, 38] that may be suitable for software projects, but be aware that model contracts often have adopted such a generalized approach that they may not be suitable for a specific situation without significant modification. Advice should be sought from an expert in software contracts.

4.6.4.7 When placing contracts for software internationally, seek advice from legally qualified persons with special expertise in the country of placement.

4.6.4.8 It is recommended that advice from software engineers is taken into account in creating software contracts.

4.6.4.9 Legal issues regarding software are currently less subject to precedent than many other legal matters. If unsure do not rely on experience of non-software contracts only.

4.6.4.10 If in any doubt, seek legal advice from a source with experience of software contracts.

5. Emerging technologies

5.1 General

5.1.1 New techniques, such as neural networks, object orientation, fuzzy logic and formal mathematical methods, will have to be judged individually against the confidence that can be placed on current techniques. The implications of using them in safety-related applications should therefore be considered carefully in relation to the integrity level [6].

5.1.2 The recommendations of these Guidelines should be applied in full to any emerging technologies.

5.2 Neural networks

5.2.1 Neural networks consist of a number of simple processors or neurons, linked together by connections or synapses. The neurons combine their inputs according to a set of weightings and subsequently produce an output that is passed to other neurons.

5.2.2 The weights in the network are "trained" by applying sets of inputs and expected outputs to the network that iteratively adjusts the weighting factors of each connection.

5.2.3 The selection of the data used to train the network is crucial, since any bias will be trained into the network and may result in unexpected behaviour when real data is used.

5.2.4 Once "training" is completed, the network should be switched to a "non-learning" mode. Leaving the network in its "learning" mode, for example to attempt to compensate for mechanical wear in a system, could permit unpredictable behaviour.

5.2.5 Neural networks are a special case of adaptive control in which the network, by a process of feedforward prediction, "learns" about the device that it is to control using a "learning" algorithm and real recorded data.

5.2.6 Neural networks are effective at handling nonlinearities, and especially systems that have poor visibility for some, possibly critical, parameters.

5.2.7 Neural networks may be very difficult indeed to validate. It may also be difficult to demonstrate the stability of control systems using neural networks under all operating conditions [4].

5.3 Object orientation

5.3.1 Object oriented techniques can be applied to any or all of the analysis, design and programming lifecycle phases [6]. Object orientation views the software in a system as a set of interacting objects with their own private states, rather than as a collection of functions. It is based on the idea of information hiding or abstraction.

5.3.2 An advantage is the ability of object oriented systems to match the problem domain and to segregate architectural issues from functional aspects.

5.3.3 Once an object has been defined the interfaces should never be altered. This leads to easier maintenance and reuse.

5.3.4 By identifying the main object classes in an application, object oriented approaches lend themselves to the creation of an object library in which commonly used object classes are maintained. This leads to an efficient development process and to opportunities for reuse.

5.3.5 It is worth noting that while there has been a great deal of interest expressed in object oriented techniques, they are much less mature than other equivalent techniques, and it still remains to be seen how effective they are in practice.

5.3.6 If object oriented programming makes extensive use of dynamic memory allocation for objects, it cannot be shown to be deterministic and is not recommended for safety-related systems.

5.3.7 When using an object oriented programming language a large run-time library of code is added to the program to assist the execution. This code should have been produced to the same integrity level as that required for the user program.

5.3.8 Object oriented analysis and design can however be used with conventional programming languages.

5.3.9 There may be situations where object oriented programming is useful, for example, rapid prototyping and modelling.

5.4 Fuzzy logic

5.4.1 When the rules of the fuzzy logic are fixed, fuzzy logic is an entirely deterministic method of making decisions or controlling systems. In this respect, it is no different from any other software based control or decision making process.

5.4.2 Fuzzy logic is, in effect, a process of interpolation between data sets. Software that operates by means of interpolation is very common in automotive applications.

5.4.3 Fuzzy logic may be implemented in a number of different ways.

(a) Leaving the controller form relatively unconstrained may lead to software that is cumbersome to program, operates slowly and requires a relatively large amount of space. However, the controller may then more closely capture the ideas of the engineer who was responsible for its design.

(b) Having a relatively constrained controller is likely to lead to software that is more easily developed and robust in use, but which may capture the ideas of the control system designer rather less.

5.4.4 Generally, the fuzzy controller will be more robust in its development and operation if:

- the data sets have a relatively simple shape
- the controller has only a few inputs and outputs (preferably two or less inputs; the number of outputs is less critical)
- the actions of the controller are defined over all the values that all the inputs can take.

5.5 Formal mathematical methods

5.5.1 A formal mathematical method is a description, based on a self-consistent mathematical theory of axioms and rules of inference, which is amenable to objective analysis, manipulation and proof. Formal mathematical methods involve the use of formal logic for the specification and verification of software.

5.5.2 Formal mathematical methods attract much attention for their "proof" potential in critical applications [39]. Extensive research is being undertaken, and a large volume of published literature has resulted. This research has indicated that there may be significant benefits in using formal mathematical methods, and as a result some specific recommendations associated with formal mathematical methods and formal requirements specification have been included within these Guidelines. Generally, however, its classification as an "emerging technology" is still appropriate, together with the general guidance given in this section.

6. References

[1] *Diagnostics and Integrated Vehicle Systems*, MISRA Report 1, 1994.

[2] *Integrity*, MISRA Report 2, 1994.

[3] *Noise, EMC and Real-Time*, MISRA Report 3, 1994.

[4] *Software in Control Systems*, MISRA Report 4, 1994.

[5] *Software Metrics*, MISRA Report 5, 1994.

[6] *Verification and Validation*, MISRA Report 6, 1994.

[7] *Subcontracting of Automotive Software*, MISRA Report 7, 1994.

[8] *Human Factors in Software Development*, MISRA Report 8, 1994.

[9] *Sources of Information*, MISRA Phase 1 Report, 1994.

[10] ISO 9001, *Quality systems — Model for quality assurance in design/development, production, installation and servicing*, 1987.

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

[12] TickIT, *A Guide to Software Quality Management System Construction and Certification using ISO 9001/EN 29001/BS 5750 Part 1*, TickIT, Issue 2.0, 28 February 1992.

[13] IEC 65A (Secretariat) 122, *Software for computers in the application of industrial safety-related systems*, draft, IEC 65A WG 9, version 1.0, 26 September 1991.

[14] IEC 65A (Secretariat) 123, *Functional safety of electrical/electronic/programmable electronic systems: Generic Aspects. Part 1: General requirements*, draft, IEC 65A WG 10.

References (continued)

[15] ISO/TC 22/SC 3/WG 11 N 39 draft, *Road Vehicles Electronic Control Systems Technical Documentation*, revision 0.0, 1 October 1993.

[16] *IEEE Software Engineering Standards Collection*, The Institute of Electrical and Electronics Engineers, Inc., Spring 1991 Edition.

[17] *Guidelines for the documentation of computer software for real time and interactive systems*, The Institution of Electrical Engineers, London, 2nd edition, 1990.

[18] *Safety Related Systems*, an IEE professional brief for the engineer, issue 1, January 1992.

[19] *Guidelines on Risk Issues, The Engineering Council, February 1993; incorporating Code of Professional Practice for Engineers and Risk Issues*, The Engineering Council, October 1992.

[20] *Framework for Prospective System Safety Analysis*, Draft 1, PASSPORT I (DRIVE II Project V2057), January 1994.

[21] *Towards a European Standard: The Development of Safe Road Transport Informatic Systems*, Draft 2, DRIVE Safely (DRIVE I Project V1051), March 1992.

[22] P.J. Comer and B.J. Kirwan, "A Reliability Study of a Platform Blowdown System", in *Automation for Safety in Shipping and Offshore Petroleum Operations*, Elsevier, 1986.

[23] M.A. Hennell, D. Hedley and I.J. Riddell, "Program analysis and systematic testing" in *High Integrity Software*, edited by C. Sennett, Pitman Publishing, 1989.

[24] SMMT, *Guidelines for the Achievement of EMC in Motor Vehicles*, 2nd edition, August 1992.

[25] ISO DIS 11898, *Road vehicles – Interchange of digital information – Controller area network (CAN) for high-speed communication*, to be published.

[26] ISO DIS 11519, *Road vehicles – Low-speed serial data communication*, to be published.

[27] SAE Recommended Practice J1850, *Class B Data Communications Network Interface*, August 1991.

[28] Californian Environmental Protection Agency — Air Resources Board, 1968.1, *Malfunction and Diagnostic System Requirements — 1994 and Subsequent Model-Year Passenger Cars, Light-Duty Trucks and Medium-Duty Vehicles and Engines (OBD II)*, The State of California, 1988.

[29] ISO CD 14230, *Keyword Protocol 2000*, to be published.

[30] J.B. Best, *Cognitive Psychology*, West Publishing Company, LA, 1986.

[31] HMSO, *HSE Display Screen Equipment Regulations*, Statutory Instrument number SI 2792, 1992.

[32] N.E. Fenton, *Software Metrics: A Rigorous Approach*, Chapman and Hall, 1991.

[33] R.B. Grady and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, NJ, 1987.

[34] BS 5515, *Code of practice for documentation of computer-based systems*, 1984 (1991).

[35] D. Bainbridge, *Intellectual Property*, Pitman Publishing, April 1992.

[36] D. Bainbridge, *Software and Copyright Law*, Pitman Publishing, November 1993.

[37] MF/1, *Model Form of General Conditions of Contract*, IEE, 1992.

[38] MF/2, *Model Form of General Conditions of Contract*, IEE, 1991 .

[39] *Formal Methods in Safety Critical Systems*, IEE Public Affairs Report number 9, June 1991.

7. Index

Index (continued)