SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 00-19

Safe Architectural Design Principles

Andrew Hussey
Brenton Atchison

June 2000

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Note: Most SVRC technical reports are available via anonymous FTP, from svrc.it.uq.edu.au in the directory /pub/SVRC/techreports. Abstracts and compressed postscript files are available via http://svrc.it.uq.edu.au.

Safe Architectural Design Principles

Andrew Hussey
Brenton Atchison

Software Verification Research Centre,
School of Information Technology
University of Queensland, Australia
email: {ahussey,brenton}@svrc.uq.edu.au

Abstract

This report discusses architectures for safety-critical sysems. The report summarises the existing literature in the area as well as the guidance provided by existing safety-critical system development standards. We discuss the three constituent functions of fault tolerant architectures: error detection, damage assessment and confinement and error recovery. We also consider methods for fault prevention.

Keywords: safety-critical, architecture, fault tolerance

1 Introduction

1.1 Scope and Purpose

This report discusses principles of safe architectural design and provides design guidelines describing generic design solutions for the prevention and tolerance of faults. Most of the principles and guidelines discussed can be applied at multiple levels of design granularity, from high-level system architecture to detailed component design.

A computer system is safety-critical when its operation (or lack of operation) could result in death, injury or environmental damage. Unsafe behaviour may arise from erroneous requirements or failure of the system to meet the requirements. In this report, our focus is on the design of the implemented system to minimise the occurrence of system failures.

System failures are caused by faults in the system. When the faulty part of the system is exercised, the fault gives rise to errors in the system state that may lead to eventual system failure [Lee94]. The first priority of safe design is to eliminate faults during the design process. Where faults cannot be eliminated, the system design must manage safety in the presence of system faults through fault tolerance.

Our approach for describing safe design principles is described in more detail in section 2. Section 3 gives detailed guidance on architectural principles for safe design using design guidelines. In section 4 we summarise the advice provided by existing safety standards.

1.2 Acronyms and Definitions

Erroneous state	An internal state of a component that could lead to a failure by a sequence of valid transitions

Error	A defective value in an erroneous state of a system
Failure	A deviation of the system behaviour from its specification
Fault	A defect in part of a component or in the design of a system
Hazard	A state of a system with the potential for harm
System	a set of interacting components together with a design that prescribes and controls the pattern of interaction

2 Approach

The safe design guidelines presented in this report are categorised in accordance with principles and strategies drawn from the literature. Guideline names are indicated by **boldface** type. For each guideline, the following information is described:

1. The problem addressed by the guideline and any special conditions that constrain the guideline's applicability.
2. The solution provided by the guideline and why the guideline is likely to prove useful to designers.
3. The applicable principle that the guideline satisfies and the decisions that may arise in implementing the solution.
4. Other related or potentially applicable guidelines (optional).

The guidelines presented are not detailed design solutions and their application is not strictly algorithmic. Instead, they describe design patterns which designers should alter according to their needs. The advantages of this approach include:

1. Communication: a common language for discussing design issues and for communicating principles of design;
2. Documentation: a common basis for understanding a system; and
3. Reuse: a storehouse of available solutions for common design scenarios.

3 Safe Design Guidelines

3.1 Framework

The framework for safe design guidelines is presented in Figure 1. The framework is constructed from a hierarchy of safe design principles. Guidelines are indicated in **boldface** at the right of the applicable principle that the guideline satisfies. There are two complementary top-level principles to achieving safe systems: fault prevention and fault tolerance.

Fault prevention is concerned with ensuring faults are not present in the operational system. Safety-related design faults can be eliminated, in theory, by developing design criteria and requirements based on the identified system hazards, and tracing such requirements and constraints to the resolution in the system implementation [Leveson95].

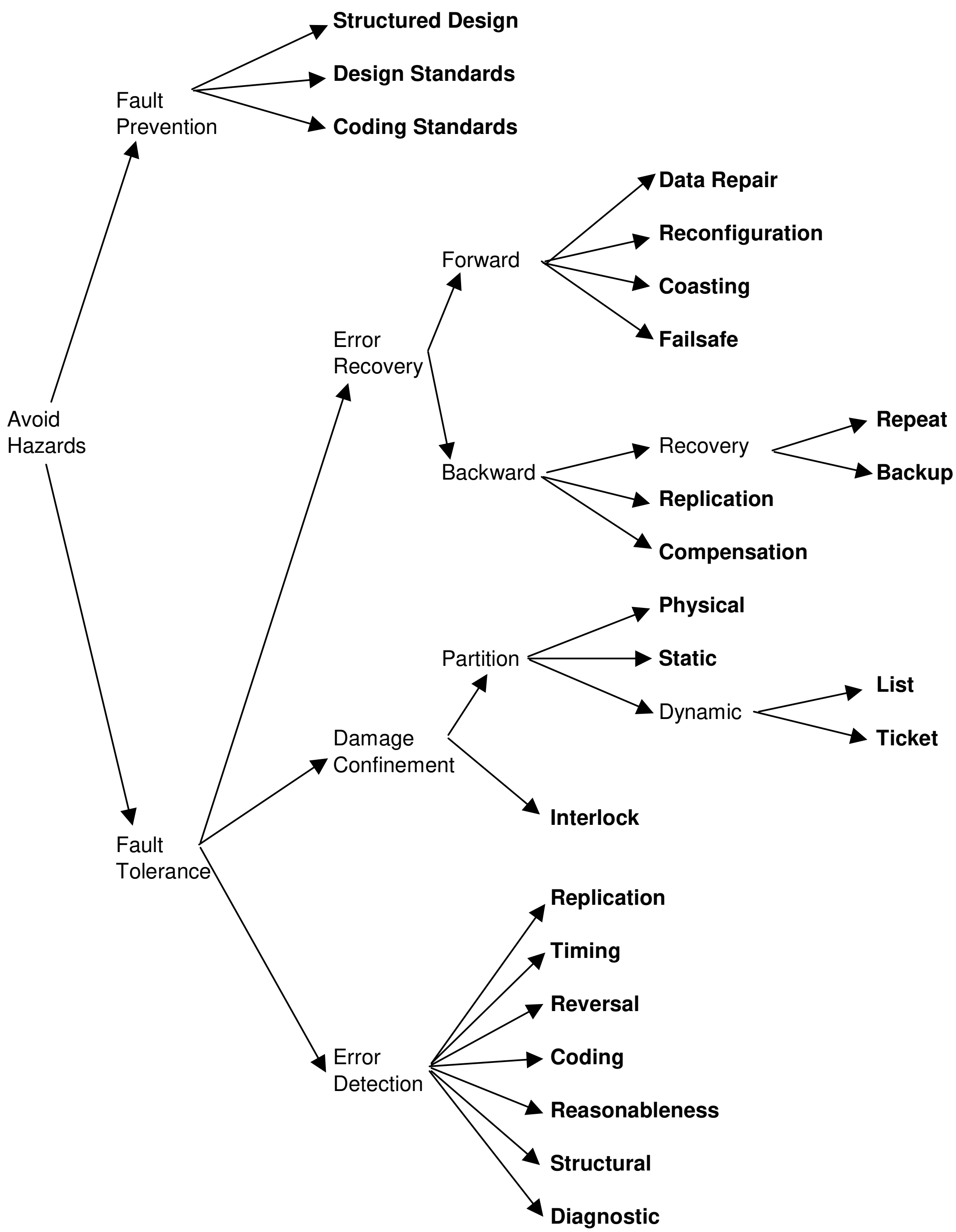

Figure 1 - Safe Design Guideline Framework

Fault prevention cannot be applied to uncontrollable fault sources such as hardware degradation or human error. Furthermore, it is often impractical to eliminate all design faults from a complex system. Fault tolerance techniques provide dependable behaviour in the presence of faults by detecting the presence of errors in a system and providing error recovery mechanisms [Lee94].

There are three constituent functions of fault tolerant architectures [Lee90]:

1. Error detection
2. Damage assessment and confinement
3. Error recovery

A fault cannot be detected by a system until the manifestation of the fault generates errors in the system function. Once an error is detected, it may be necessary to assess the extent to which the system state has been damaged. The system should be designed to confine possible damage to a minimum area. Based on the damage assessment, error recovery techniques aim to transform the current erroneous state into a well-defined error free state from which normal system operation can continue.

All techniques for achieving fault tolerance depend on the effective deployment and utilisation of redundancy [Lee90]. There are many factors that influence where redundancy should be deployed in a system and how much is actually required. The principle factor will be the reliability requirements of the particular application. Other factors will involve limitations on costs or constraints on the system within which the software is to operate (e.g., power, weight, and space).

3.2 Fault Prevention Guidelines

One of the most important principles of safe design is simplicity. A simple design tends to minimise the number of components, functional modes and interfaces between components and it has a small number of unknowns in terms of interactions with the system and with system operations [Leveson95]. According to Kletz [Kletz84], some of the reasons for complexity in system designs are:

1. The need to add complicated equipment to control hazards.
2. A desire for flexibility.
3. The use of redundancy.

A simple software system exhibits several features [Leveson95]:

1. The design should be testable (both observability of test results and controllability of test parameters).
2. The design should be portable.
3. The design should be easily understood and readable.
4. Interactions between components should be limited and straightforward.
5. Worst-case timing should be determinable from the code.
6. The code should contain only the minimum features required by the system specification.
7. The design should be deterministic including predictable real-time process scheduling, avoidance of interrupts and predictable use of resources, such as the processor, memory and network.

Simplicity in design is promoted by the use of

- **Structured design techniques** to enforce particular representations and style of design.
- **Design standards** to enforce characteristics of the design.
- **Coding standards** that avoid language constructs that are not well defined e.g., compiler

specific) or that inhibit analysis of program behaviour.

Although not strictly design guidelines (no specific guidance is provided for structuring systems), structured design techniques constrain potential design solutions. Structured design techniques are "thought tools" for systematically perceiving and partitioning a problem. They are characterised by:

1. A defined notation for description of behaviour.

2. Identification of the system boundary and description of the system environment.

3. Capacity to provide logical means of problem decomposition.

4. Suitability for review for consistency/completeness/correctness.

Structured design techniques are recommended by most safety standards, e.g., [Std5679], [Std4404], [Std00-56] and [Std61508]. Structured design techniques support methodical construction of test cases. Example structured design techniques include Booch [Booch94], Jackson System Development [Jackson83] and MASCOT [Simpson86]. Software design guidelines such as those described in [Gamma94] provide detailed guidance on how structured design techniques can be used to promote goals such as testability, understandability and reusability. For a given design problem, several guidelines may be applicable, depending on which principles are prioritised.

Semi-formal methods extend structured methods with precise notations for description of behaviour. Tool support is typically required but automated review for consistency/completeness and mechanical analysis for correctness and test design is possible. Example semi-formal methods include [Sommerville92] logic diagrams, data flow diagrams, finite state machines, petri nets and decision tables.

Formal methods apply mathematical notations to describe and analyse system specifications; such notations enable unambiguous description of behaviour, mechanised checks for consistency and completeness, mechanised processes for proof of correctness, the ability to prototype, or animate, specifications and automatic generation of test cases. Examples include Z [Spivey92], CSP [Hoare85] and Sum [Johnston99].

Design standards provide guidance on quality criteria and metrics for measurement against such criteria. Such metrics include measures for complexity (e.g., cyclomatic complexity which quantifies control flow complexity, i.e., the number of linearly independent paths through a section of code) and modularity (the extent to which a system is divided into modules with high cohesion and low coupling) [Card90]. Software quality metrics typically provide a means of estimating factors such as correctness, reliability, efficiency, maintainability, testability, flexibility, portability, reusability and interoperability.

Coding standards may be either external (e.g., STANAG 4404 [Std4404] and Def Stan 00-55 [Std00-55]) or internal to an organisation. Standards may be tailored to specific languages. For example, coding standards for the C language provide guidance for avoiding constructs that produce unspecified behaviour, undefined behaviour and implementation-dependent behaviour [Hatton95]. For example, the order in which expressions are evaluated in C is unspecified, and should not be relied on by the programmer. To provide built-in support for good programming style, SPARK Ada [Barnes97] enforces use of a "safe subset" of the Ada language constructs and provides support for verification of code using mathematical proof.

3.3 Fault Tolerance Guidelines

3.3.1 Error Detection

Error detection occurs by intercepting outputs produced by a system and checking whether those outputs would conform to the specification of the system [Lee90]. Leveson [Leveson95] argues that errors should be detected as soon as possible after they arise and at a level enough low to ensure that effective action can be taken before hazardous states are reached. However, while early checks enable action to be taken sooner to reverse the error, last-moment checks (just before results leave the system) ensure that none of the activity of the system remains unchecked.

Where they are used, care should be taken to ensure that checkers does not introduce other faults. For example, the checker slows overall processing of the system data, timing of which may be safety-critical. The efficiency of the checker algorithm may be crucial to whether use of a checker is feasible. Checkers may also add complexity to the system and mask the existence of faults that would be removed during test activities.

There are three criteria for ideal checks [Lee90]:

- The check should be derived solely from the system specification.
- The check should be complete.
- The system and its check should be independent.

In practice, it may be infeasible to provide a complete check of correctness due to faults in the checker and the performance and cost overheads involved. However, it may be possible to focus on partial checks that detect violations of safety-related properties.

Similarly, it is difficult to ensure complete independence between the system and checker [Leveson95]. The check can only be guaranteed to succeed if there is no possibility of there being a common point of failure between the system and its check. Independence could be compromised if the same person designed the system and its checker. The checker needs to access the same information as the system and may corrupt that information, and its design may be based on the same assumptions as the system.

Lee and Anderson [Lee90] list several types of checks. Each of these checks is a guideline for error detection.

1. **Replication** checks. Duplicate system function through redundancy or repeated operation. This is most applicable to random faults since duplicated systematic faults will not be independent. In some cases, it may be possible to replicate systems with different designs. However, care should be taken to ensure that the designs are truly independent.

   A form of replication for software fault tolerance is N-version programming where the function of a system is replicated by diverse implementations. The most common approach to N-version programming is to use separate teams to generate versions of the system [Leveson95]. However, it has been suggested that systems produced by separate teams do not fail in a statistically independent way. Popov et al. [Popov99] discuss methods for producing design diversity, referred to as “Diversity-Seeking Decisions”. Such “DSDs” produce differences in the design process. Forcing diversity, by adopting a DSD, is beneficial if the DSD creates two process variants that offer the same guarantees of reliability. When this is not true, a tradeoff is necessary between the degree of diversity achieved and the reduction in reliability.

2. **Timing** checks. Where timing conditions are known, introduce timeouts or watchdogs. Timing checks ensure that timing constraints are satisfied.
3. **Reversal** checks. Where the inverse of a computation or function is unique, apply the inverse to check the result. Reversal checks take the output from a system and calculate what the inputs should have been to produce that output. This is most effective where the function inverse is much simpler than the function itself, e.g., calculation of a square root.
4. **Coding** checks. Use redundant information in stored or communicated data to detect forms of corruption. Common examples include parity checks and checksums.
5. **Reasonableness** checks. Reasonableness checks determine the acceptability of outputs based on knowledge of the domain. The results of computations are examined for satisfaction of some condition derived from the specification. A common example is data range checks but it is possible to apply more sophisticated checks on safety-critical data.

   For example, it may be possible to provide an algorithm that checks the correctness of the data output without duplicating the process of generating that data. Such checks may be simpler to perform than the process of generating the data.
6. **Structural** checks. Examine the structural integrity of data. For example, corruption of two-way linked lists can be detected if the link structure is impaired.
7. **Diagnostic** checks. Apply tests to system components for compliance with predicted results. Components often are supplied with self-test facilities for this purpose. Diagnostic tests can be performed on-line or off-line.

A common type of checker is the monitor, which functions as a data checking entity in the data flow path for the system after generation of data but prior to its use. The module implementing the check is inserted into the data-flow for the system after the generation of the data, but before its use. If the data is found to be defective, the checker module blocks the use of the data and the data is regenerated and rechecked.

A different type of checker is the safety kernel, which centralises the monitoring mechanisms in a high integrity system component. All safety-critical functions are conducted through the request to the kernel, allowing the detection of unsafe behaviour [Leveson95]. A safety kernel allows centralisation and encapsulation of safety mechanisms with the associated advantages of reusability and verification of the kernel operations.

3.3.2 Damage Confinement

Damage confinement is concerned with structuring the system to minimise the spread of errors. Leveson [Leveson95] advocates decoupling of systems wherever possible. Partitioning to isolate safety-critical components is also recommended by standards such as [Std4754].

A tightly coupled system is highly interdependent, with each component linked to many others so that component failure can rapidly affect the status of the wider system. Tightly coupled systems tend to be rigid with an overall design that includes time-dependent processes, rigid sequencing of events and rigid, efficient use of resources.

The principles of decoupling can be applied to software by using modularisation to control complexity and promote software-engineering goals such as reusability, simplicity and testability.

Enforcing separation between design elements is commonly called **partitioning**. We distinguish between different means of providing partitioning as follows:

1. **Physical** partitioning. The decoupled systems are implemented using physically distinct hardware. Physical partitioning is the best means of ensuring separation but is not always practical for highly integrated systems or for systems using common resources.

2. **Static** logical partitioning. Logical separation of entities is achieved through the static system design structure. Such partitioning needs to be verified to ensure that the implemented data and control flows cannot violate the intended structure. It is also important to eliminate unintended component interaction through the use of common resources, for example, common memory, processor or external devices.

3. **Dynamic** logical partitioning. Access between decoupled components is explicitly controlled by run-time software mechanisms. Such partitioning is typically achieved by authorisation-based protection mechanisms [Lee90], which actively constrain access to objects controlled by a system. Authorisation mechanisms are commonly used in security applications to ensure appropriate access or information flow between system entities. Two examples of authorisation mechanisms are:

   - **List-based** schemes, which maintain a list of authorised users associated with each system object; and

   - **Ticket-based** schemes, which associate the right to access an object with the process rather than the object. The set of access rights for a process defines the protection domain for that process. Information may pass directly between processes if their protection domains overlap and hence allow objects to be shared. Tickets can be with respect to certain capabilities only. A similar mechanism to constrain access to objects is to use a "lock". Only objects that have the key for that object can access the object. To implement an atomic action it is necessary to lock all of the objects to be accessed within that action and to give the keys only to the processes concerned.

Errors can also be confined by erecting barriers between incompatible system states or events [Leveson95] using **interlocks** to enforce a sequence of events. Interlocks differ from partitioning mechanisms since they allow interaction between system components but restrict when that interaction can occur. Inhibitor interlocks ensure that an event cannot occur inadvertently or while a condition exists. A sequencer interlock ensures that an event cannot occur before another event.

3.3.3 Error Recovery

Error recovery is dependent on a prior assessment of the extent to which error has propagated through the system. When an accurate prediction of the extent of damage can be made, the damage is anticipated and when an accurate assessment cannot be made the damage is unanticipated [Lee90].

We distinguish between forward and backward error recovery mechanisms.

Forward error recovery corrects errors arising from faults and continues processing. The erroneous part of the state is repaired and processing continues without rolling back the state of the machine. Anticipated errors arising from anticipated faults can be dealt with by forward error recovery.

Forward error recovery measures are:

- Dependent on damage assessment and prediction.
- Inappropriate for recovery from unanticipated faults.
- Designed specifically for a particular system.
- Impossible to implement as generic mechanisms that deal with unanticipated faults.

Leveson [Leveson95] suggests that forward recovery is needed when there is insufficient time to repeat a computation or the control actions depend on the incremental state of the system and cannot be recovered.

Examples of forward error recovery guidelines include:

1. **Data repair**. This approach is possible if there is sufficient redundancy in data or data structure to repair a fault, once it is detected. Care needs to be taken that the repair moves the system to a correct state.

2. **Reconfiguration**. This is a common technique where a partial shutdown or reconfiguration is performed to remove a faulty item from the system. An example of reconfiguration is to move to a backup system, possibly with degraded performance. The backup may be performed by a separate computer system, or by reverting to operator control.

3. **Coasting.** Where errors may be transient, for example communications corruptions, it may be possible to ignore the occurrence of errors and continue operation with no change until the error is corrected. This is particularly appropriate for some real-time systems where data is refreshed at regular intervals. However, care should be taken that new failure modes are not introduced as a result.

4. **Failsafe.** If a safe state is known, it may be possible to recover from an error by forcing the system permanently to that safe state. Of course, this strategy is only appropriate where halt of operation is possible. Care should also be taken to ensure that the safe states are appropriate for all circumstances. Failsafe differs from data repair in that a failsafe strategy does not attempt to continue processing after a safe state has been identified.

Dealing with unanticipated errors is more difficult and a more general error recovery technique is needed. **Backward error recovery** reverses the state of the system to a prior state that is regarded as error free and which preceded invocation of the fault. In contrast to forward error recovery, backward error recovery is:

- Independent of damage assessment
- Capable of providing recovery from arbitrary faults
- Generally applicable to all systems
- Easily provided as a mechanism

Backward error recovery is a powerful error recovery technique, however forward error recovery will normally be more economical than measures to restore a prior state [Lee90]. Backward error recovery incurs penalties both in speed of operation of the system and expenditure of resources for storing information. In addition, some systems cannot be restored to their prior state because they produce physical effects in real time.

Dealing with a fault by applying backward error recovery requires that there be some redundancy in the system; the form of the redundancy depends on whether the fault is transient or permanent. Specific backward recovery guidelines include:

1. **Recovery and retry**. This approach uses redundancy in time or between design components to enable an erroneous computation to be retried. Some form of state restoration may be necessary to recover the system to a previously known safe state. The restoration can be achieved by checkpointing, or storing states that can be reverted to, or by recording an audit trail of events that can be reversed. If errors are detected before the system state is altered, no state restoration is necessary. The simplest of state restoration is reset, or restoration to the initial system state. Reset differs from a failsafe strategy in that processing continues after the reset.

   Once a safe state is recovered, retry can occur in two ways:

   - **Repeat**. Where faults are transient, redundancy in time by repeating operation with the same component is sufficient. Some design faults, especially software faults, may appear transient if they are triggered by rare operational circumstances. Repeating operation using the same component is based on the premise that many of the errors caused by software failures are based on the premise that the same error is unlikely to occur.

   - **Backup**. Where faults are permanent, the faulty component must be removed from the system and replaced by another component. Storey [Storey96] distinguishes between the "hot" standby where the alternative component is running continuously in parallel and "cold" standby where the alternative component is started only when needed.

     A form of recover and retry for software is the recovery block. On initial entry to a recovery block, a recovery point is implicitly established and the primary software module is executed. On completion of the module an acceptance test for the block is evaluated. If the test or the execution of the module does not raise any exception then the results of the module are assumed to be acceptable and the recovery block is exited. If an exception is raised, the recovery point is restored and an alternative module is executed [Leveson95].

2. **Replication** is a special case of backward recovery where multiple systems are run in parallel so that state restoration is not necessary [Leveson95]. Once a discrepancy between duplicate systems is detected, the systems may vote to determine the outcome of a processing activity.

3. **Compensation** provides the appearance of backward recovery by attempting to undo the effects of unrecoverable operations. For example, aborting an operation after it has erroneously been executed can restore the system to a previous safe state, even though there is no actual state restoration.

3.4 Applying the Guidelines

We suggest the following process for application of the presented guidelines.

1. As first priority, design for fault prevention. This is particularly important for design faults that can be eliminated, in theory.

2. Design for fault isolation (damage confinement), particularly to minimise the safety critical design space.

3. Where sufficient assurance cannot be gained in reduction of faults, apply fault tolerant design strategies to prevent the occurrence of hazardous failures. Each fault tolerant design should embody an error detection, damage assessment and error recovery mechanism.

4 International Safety Standards

Various approaches to safe architectural design are included in international safety standards. The standards recognise that the system architecture can influence the required integrity of system software components and offer advice on how this might be achieved.

Def (Aust) 5679 [Std5679] highlights independence as the key requirement for safe design. Two approaches for safe design are suggested: redundancy and safety kernels. Redundancy includes functional, data and design. However, design redundancy is prohibited as sufficient mitigation of software faults. Safety kernels are used to monitor the state of a system and to maintain safe states. It is noted that software control categories are a form of safety kernel where errors may be detected and controlled outside of software influence. However, the trust placed in control categories is restricted. The standard also advocates good engineering practices, including use of structured design techniques.

MIL-STD-882C [Std882C] does not directly address safe design but it uses software control categories as a means of software risk assessment. Software control categories are a means of measuring the degree of influence of software over the environment, taking into account other systems and operational procedures.

The NATO Standardization Agreements 4404 [Std4404] provides detailed guidance on safe design. The standard includes recommendations for the design and development process, system design, self-checking, design requirements, protection of safety-critical computing system functions, component interface design, software design, coding, and software analysis and testing. System design guidelines include isolation of safety-critical functions on a stand-alone computer and use of fallback and recovery to a designed safe state in the event of component failure. The use of self-checks for timing of operations, memory integrity and the correct function of safety-critical components is advocated, as well as the use of reasonableness checks on component inputs. Software design guidelines include use of modularity, limitation of access to safety-critical functions and use of timers to monitor execution of uninterruptable sections of code.

UK Def Stan 00-56 [Std00-56] does not discuss design for safety explicitly. However, implicit design principles underly the integrity allocation process. In particular, the standard allows integrity requirements to be reduced where safety functions are implemented by independent components. The guidance for DefStan 00-56 (Part 2 of the standard) also advocates use of structured design techniques and good engineering practices such as design reviews and use of software coding standards.

The SEA ARP 4754 standard [Std4754] recognises several examples of safe architectural design principles. Partitioning is advocated to isolate or contain faults, mostly for the purpose of separating software items of different integrity. Use of dissimilar designs is also recommended by providing multiple implementations of a function, either through monitoring or backup systems.

IEC61508 [Std61508] was developed primarily for systems where Equipment Under Control is supplemented by a separate Safety Related Control System. The standard provides detailed guidance on hardware design, including duplication of architecture components and diagnostic facilities. Software design techniques are also recommended for different target integrity levels. Structured design methods and programming restrictions are enforced to encourage fault prevention.

5 Discussion

5.1 Summary

This report has examined high-level principles for safe architectural design, such as use of partitioning, and low-level guidelines that enable designers to satisfy such principles, for example, use of list-based component access mechanisms. The principles and guidelines have been derived from the literature and current safety-critical systems development standards.

The safest architecture is fault-free and fault prevention techniques have been discussed that promoted the elimination of faults. However faults are almost inevitable in any complex system and fault tolerance mechanisms are necessary to enable safe system function in the presence of faults. Three properties of fault tolerant systems have been identified: error detection; damage confinement; and error recovery. Principles and guidelines have been discussed that enable a system to be designed such that the resulting design exhibits these essential fault tolerance properties.

5.2 Future Work

The guidelines presented are not project-specific, and are abstract patterns for general application. This report will be supplemented with project-specific examples of safe design practice in future versions of this document.

An important use of safe design principles is to guide the allocation of integrity levels to design components. A report describing how this may be achieved is forthcoming.

6 Acknowledgements

We gratefully acknowledge the support of the Department of Defence which funded this work under the Defence Software Acquisition Reform project which is investigating enhancements to the Australian Defence Standard Def(Aust) 5679 as part of an initiative to improve techniques for safety-critical system development in the Australian defence industry.

7 References

7.1 Standards

[Std5679] Australian Department of Defence, Def (Aust) 5679, The Procurement of Computer-Based Safety Critical Systems, 1998

[Std4404] NATO Standardization Agreement (STANAG): Safety Design Requirements and Guidelines for Munition Related Safety Critical Computing Systems, 199?

[Std4754] Society of Automotive Engineers. Aerospace Recommended Practice 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems, 1996.

[Std882C] US Department of Defense, MIL-STD-882C: Standard Practice for System Safety Program Requirements, 1996.

[Std61508] International Electrotechnical Commission, IEC 61508, Functional Safety of electrical/electronic/programmable electronic safety-related systems, 1998.

[Std00-55] UK Ministry of Defence, Defence Standard 00-55: Requirements for Safety Related Software in Defence Equipment, 1997.

[Std00-56] UK Ministry of Defence, Defence Standard 00-56: Safety Management Requirements for Defence Systems, 1991.

7.2 Research Papers and Books

[Barnes97] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[Booch94] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin Cummings, 2nd edition, 1994.

[Card90] D. N. Card and R. L. Glass. *Measuring Software Design Quality*. Prentice-Hall, 1990.

[Gamma94] E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Guidelines: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Hatton95] L. Hatton. *Safer C*. McGraw-Hill, 1995.

[Herrmann99] D. S. Herrmann. *Software Safety and Reliability*. IEEE Computer Society, 1999.

[Hoare85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Jackson83] M. A. Jackson. *System Development*. Prentice-Hall, 1983.

[Johnston99] W. Johnston and L. Wildman. *The Sum Reference Manual*, Technical Report 99-21, Software Verification Research Centre, 1999.

[Kletz84] T. Kletz. *Myths of the Chemical Industry*. The Institution of Chemical Engineers, 1984.

[Lee90] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 1990.

[Lee94] P. A. Lee. Software Faults: The Remaining Problem in Fault Tolerant Systems?. In M. Banatre and P. A. Lee, editors, *Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives*, pages 171-181. Springer-Verlag, 1994.

[Leveson95] N.G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, Chapter 16, 1995.

[Popov99] P. Popov, L. Strigini and A. Romanovsky. Choosing Effective

Methods for Design Diversity – How to Progress from Intuition to Science. In M. Felici, K. Kanoun and A. Pasquini, editors, *Computer Safety, Reliability and Security: 18th International Conference, SAFECOMP'99*, pages 272-285. Springer, 1999.

[Simpson86] H. Simpson. The MASCOT method. *BCS/IEE Software Engineering Journal*, 1(3): 103-120, 1986.

[Sommerville92] I. Sommerville. Software Engineering. Addison-Wesley, 4th edition, 1992.

[Spivey92] J. M. Spivey, *The Z Notation: a Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[Storey96] N. Storey, *Safety-Critical Computer Systems*, Addison Wesley, 1996.