

CONFERENCES IN RESEARCH AND PRACTICE IN
INFORMATION TECHNOLOGY

VOLUME 69

SAFETY CRITICAL SYSTEMS AND
SOFTWARE 2006



AUSTRALIAN
COMPUTER
SOCIETY

SAFETY CRITICAL SYSTEMS AND SOFTWARE 2006

Proceedings of the
**Eleventh Australian Workshop on Safety Critical
Systems and Software (SCS2006)**, Melbourne,
Australia, August/September, 2006

Tony Cant, Ed.

Volume 69 in the Conferences in Research and Practice in Information Technology Series.
Published by the Australian Computer Society Inc.



Published in association with the ACM Digital Library.

Safety Critical Systems and Software 2006. Safety Critical Systems and Software 2006. Proceedings of the Eleventh Australian Workshop on Safety Critical Systems and Software (SCS2006), Melbourne, Australia, August/September, 2006

Conferences in Research and Practice in Information Technology, Volume 69.

Copyright ©2006, Australian Computer Society. Reproduction for academic, not-for profit purposes permitted provided the copyright text at the foot of the first page of each paper is included.

Editors: **Tony Cant**
Trusted Computer Systems Group
Information Networks Division
Defence Science and Technology Organisation
PO Box 1500
EDINBURGH SA 5111
Australia
`tony.cant@dsto.defence.gov.au`

Series Editors:
Vladimir Estivill-Castro, Griffith University, Queensland
John F. Roddick, Flinders University, South Australia
Simeon Simoff, University of Technology, Sydney, NSW
`crpit@infoeng.flinders.edu.au`

Publisher: Australian Computer Society Inc.
PO Box Q534, QVB Post Office
Sydney 1230
New South Wales
Australia.

Conferences in Research and Practice in Information Technology, Volume 69.
ISSN 1445-1336.
ISBN 1-920-68250-3.

Printed, May 2007 by Flinders Press, PO Box 2100, Bedford Park, SA 5042, South Australia.
Cover Design by Modern Planet Design, (08) 8340 1361.

The *Conferences in Research and Practice in Information Technology* series aims to disseminate the results of peer-reviewed research in all areas of Information Technology. Further details can be found at <http://crpit.com/>.

Table of Contents

Safety Critical Systems and Software 2006. Proceedings of the Eleventh Australian Workshop on Safety Critical Systems and Software (SCS2006), Melbourne, Australia, August/September, 2006

Foreword.....	vii
The Australian Safety Critical Systems Association	viii
Sponsors	ix

Full Papers

Architecture-driven Modelling and Analysis	3
<i>David Garlan and Bradley Schmerl</i>	
Certified Software Factory: Open Software Toolsuites, Safe Methodologies and System Architectures .	19
<i>J. U. Gärtner</i>	
On proof-test intervals for safety functions implemented in software	23
<i>Alena Griffiths</i>	
Dynamic Design and Evaluation of Software Architecture in Critical Systems Development	35
<i>Klaus Marius Hansen and Lisa Wells</i>	
Assuring Separation of Safety and Non-safety Related Systems.....	45
<i>Bruce Hunter</i>	
Using Software Architecture Techniques to Support the Modular Certification of Safety-Critical Systems	53
<i>Tim Kelly</i>	
Formal Modelling and Analysis of Mission-Critical Software in Military Avionics Systems	67
<i>Zahid H. Qureshi</i>	
Certification Criteria for Emulation Technology in the Australian Defence Force Military Avionics Context	79
<i>Derek Reinhardt</i>	
Safety, Software Architecture and MIL-STD-1760	93
<i>Matthew John Squair</i>	
Implementation of a Triple Modular Redundant FPGA based Safety Critical Systems for Reliable Software Execution.....	113
<i>Venkatesh Vasudevan, Peter Waldeck, Hardik Mehta and Neil Bergmann</i>	
Author Index	121

Foreword

The *2006 Australian Workshop on Safety-Related Programmable Systems* was held in Melbourne on 31 August and 1 September, 2006. The workshop, sponsored by the Australian Safety Critical Systems Association, had the theme: “Safe Software Architectures” and was attended by 45 participants. Roughly half of the workshop papers addressed safety standards, while the other half covered the use of tools and techniques for safety assurance.

Once again, four international keynote speakers presented talks at the workshop:

- Klaus Marius Hansen, Associate Professor at the Computer Science Department, University of Aarhus, Denmark.
- David Garlan, Professor in the School of Computer Science at Carnegie Mellon University, USA.
- Tim Kelly, Lecturer in Department of Computer Science at the University of York; also Deputy Director of the Rolls-Royce Systems and Software Engineering University Technology Centre, UK.
- Jakob Gärtner, Technical Director of Esterel Technologies, Germany.

Full program details are available from <http://www.safety-club.org.au>.

The organizing committee is very grateful to the authors for the trouble they have taken in preparing their work to be included in these workshop proceedings. The papers were peer-reviewed for relevance and quality by the Association Committee and their colleagues. Note, however, that the views expressed in the papers are the authors’ own, and in no way represent the views of the editor, the Association Committee, or the ACS generally. The fact that the papers have been accepted for publication should not be interpreted as an endorsement of the views or methods they describe, and no responsibility or liability is accepted for the contents of the articles or their use.

The committee also wishes to thank the workshop sponsors for their support: Hyder Consulting, Airservices Australia, the Centre of Excellence in Defence and Industry Systems Capability (CEDISC) and the Defence Materiel Organisation in the Australian Government Department of Defence. These organisations have all helped to make the workshop a success.

I wish to thank the other members of the organising committee: Chris Edwards (Treasurer), Kevin Anderson (Secretary and Workshop Chair), and George Nikandros (Association Chair). Thanks are also due to the paper reviewers for their constructive comments. Finally, thanks to Karl Reed and the Computer Systems and Software Engineering Board of the ACS for their ongoing support of the Association.

Tony Cant
Workshop Chair
January 2007

The Australian Safety Critical Systems Association

Computer systems and embedded computers pervade all aspects of modern daily life, and many implement functions that have the potential to cause death or injury if they do not operate correctly. Some of these systems include emergency service dispatch, car braking, aircraft flight controls, railway control, and telecommunications systems. These systems are not safe by accident but require safety to be designed into them.

The Australian Computer Society launched the Australian Safety Critical Systems Club (as it was then named) on 17 October 2002, in conjunction with its annual SCS workshop in Adelaide. Chapter launches have been held in Perth, Melbourne, Sydney, Canberra and Brisbane.

At the AGM held in Sydney on 25 August 2005, it was agreed to change the name Australian Safety Critical Systems Club to Australian Safety Critical Systems Association.

The Association aims to foster discussion on the design and development of safety critical systems, as well as debate on more philosophical issues of safety standards, including questions such as How safe is safe enough?

Specifically, the Association's purpose is to:

- Provide a national focus and forum for its members who have an interest in safety-related systems, particularly those systems containing software.
- Provide professional association services for all categories of its membership.
- Stimulate the active contribution and participation of its members in the development and dissemination of safety-related systems knowledge and to support the activities of the Society.
- Foster and support education and training associated with all aspects of safety-related systems.
- To provide learned society functions for individuals and industry groups and to provide practice based opinion and advice for the Society.

Membership of the Australian Safety Critical Systems Association is open to anyone involved in design and development of safety critical systems, or with an interest in system safety issues. The Association is also expected to be relevant to people interested in the assurance of systems dependability, including Reliability, Availability, Maintainability and Safety (RAMS) of systems.

The SCS Association Committee is as follows:

- Chairman: George Nikandros, Queensland Rail
- Secretary: Kevin Anderson, Function Leader - Risk and Reliability, Hyder Consulting
- Treasurer: Chris Edwards, AMW Pty Ltd
- Members: Tony Cant, Robert Worthington, Peter Hartfield, Allan Coxson, Clive Boughton

For more information about the Australian Safety Critical Systems Association, visit our web site at <http://www.safety-club.org.au>.



Tony Cant
Defence Science and Technology Organisation, Australia
SCS2006 Program Chair
August/September, 2006

Sponsors

We wish to thank the following for their kind contributions towards this workshop.



Centre of Excellence in Defence and Industry Systems Capability



Australian Government

Department of Defence

Defence Materiel Organisation

FULL PAPERS

Architecture-driven Modelling and Analysis^{*}

David Garlan and Bradley Schmerl

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213 USA
{garlan, schmerl}@cs.cmu.edu

Abstract

Over the past 15 years there has been increasing recognition that careful attention to the design of a system's software architecture is critical to satisfying its requirements for quality attributes such as performance, security, and dependability. As a consequence, during this period the field of software architecture has matured significantly. However, current practices of software architecture rely on relatively informal methods, limiting the potential for fully exploiting architectural designs to gain insight and improve the quality of the resulting system. In this paper we draw from a variety of research results to illustrate how formal approaches to software architecture can lead to enhancements in software quality, including improved clarity of design, support for analysis, and assurance that implementations conform to their intended architecture.

Keywords: Software Architecture, Architecture Analysis

1 Introduction

Software architecture is concerned with the high-level structures of a software system, the relationships among them, and their properties of interest. These high-level structures represent the loci of computation, communication, and implementation. Typical properties include emergent behaviour, such as the performance, reliability, security, maintainability, and so on (Shaw and Garlan 1996, Perry and Wolf, 1992).

Well designed architectures typically allow one to reason about satisfaction of key requirements and to make principled engineering tradeoffs. They can provide clear rationale of assignment of function to components, establish principles of conceptual integrity, and lead to considerable reduction in rework over the lifespan of a system (Brookes 1975, Boehm and Turner 1993). They can also permit reuse of architectural design idioms and patterns, reduction of development costs through product line approaches, and guidance to future maintainers of those systems.

Given the potential benefits of software architecture, over the past decade and a half the field has received increasing attention and consequent progress. There are now numerous textbooks (Garlan and Shaw 1996, Bass,

Clements, and Kazman 2003, Rosanski and Woods 2005), review methods (Clements, Kazman, and Klein 2001), conferences (e.g., the Working IEEE/IFIP Conferences on Software Architecture (WICSA) and the European Workshops on Software Architecture (EWSA)), documentation standards (Clements et al. 2002, IEEE 2000), handbooks (Buschmann et al. 1996), and courses covering the topic. Success stories detailing the economic benefits and practice of product lines abound (Bosch 2000, Clements and Northrop 2001). Software development practices typically now incorporate architecture reviews, and software architects have formal titles and well-defined roles in many organizations.

Coupled with heightened awareness, and increasing maturity of practice, a number of standards bodies are now promoting notations and standards for software architecture. UML 2.0 from the Object Management Group, for example, now has improved capabilities to represent general component and connector architectures. The IEEE prescribes a meta-framework for architectural views (IEEE 2000). Some standards aim at more specific domains, such as resource constrained systems (e.g., AADL by SAE International, 2004, or SysML by the Object Management Group, 2006). Other standards-based approaches, like "model driven architecture" (MDA) from the Object Management Group (2003), attempt to provide ways to move from architectural models to architecturally-consistent implementations. Finally, the presence of middleware and their corresponding architectural frameworks have led to considerable standardization and reuse within certain application domains, (e.g., J2EE, Eclipse, ADO.NET).

However, despite notable progress and concern for ways to represent and use software architecture, specification of architectural designs remains relatively informal, relying on graphical notations with weak or non-existent semantics that are often limited to expressing only the basic of structural properties. As a consequence, it is almost impossible using today's common practices to (a) express architectural descriptions precisely and unambiguously; (b) provide soundness criteria and tools to check consistency of architectural designs; (c) analyse those designs to determine implied system properties; (d) exploit patterns and styles, and check whether a given architecture conforms to a given pattern; and (e) guarantee that the implementation of a system is consistent with its architectural design.

Copyright © 2006 Australian Computer Society, Inc. This paper appeared in the *11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, Melbourne. Conferences in Research and Practice in Information Technology, Vol. 69. Tony Cant, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

^{*} This paper accompanies the keynote talk "Software Architecture for Highly Dependable Systems" by David Garlan.

Luckily, however, research has developed techniques to address many of these shortcomings by providing more-formal approaches to architectural design. While these techniques may not be completely ready for full-scale adoption by industry, many of them are close to that level of maturity.

In this paper we outline several such techniques and their associated tools, drawing particularly from research carried out at Carnegie Mellon University in the ABLE Project. While not a comprehensive survey of existing work on formal approaches to software architecture, this paper will give a flavour for the kinds of techniques being investigated by the research community, and the kinds of potential benefits that they can bring to the field.

The remainder of this paper is organized as follows: Section 2 summarizes how to specify architectural structure; in Section 3 we introduce architectural properties and illustrate how a flexible property mechanism can facilitate architectural analysis; Section 4 shows how architectural behaviour can be specified; Section 5 introduces the concepts of architectural style, and shows how they can be used to provide domain-specific architectural models and the ability to check for conformance to a style; Section 6 presents a summary of our approaches to addressing the problem of establishing implementation conformance to an architecture; finally, Sections 7 and 8 present related work and conclusions.

2 Modelling architectural structure

The starting point for any formal treatment of software architecture is the representation of architectural structure. However, this raises the question: what kinds of structure? Any complex software system may have many structures of interest: modules, run-time entities, development teams, physical devices and networks. Today we understand that the preferred way of addressing this complexity is to recognize that an architectural design must be described in terms of a number of distinct (but related) views. Each view represents an architectural perspective on the system, exposing certain system structures and their properties, to address a particular set of concerns.

Following the approach of Clements et. al. (2002), one can categorize the kinds of structures into three general categories. First, there are *coding structures*, such as modules, packages, and classes, with relationships like uses, depends-on, inherits, etc. Second, there are *run-time structures*: databases, clients, servers, and connectors indicating communication pathways. Third, there are *allocation structures*, which map elements of the first two views into non-software entities, such as the physical setting (networks, CPUs, etc.) or development teams. These mappings lead to allocation views, such as deployment views or work breakdown structures.

In this paper we will focus on modelling and analysis of run-time structures, or **component and connector (C&C)** views. This is because such structures are the ones that most directly convey critical properties related to dependability, such as reliability, security, and performance. These are also the class of views that are least well supported by existing notations and tools.

2.1 Components, connectors, and systems

We model a run-time C&C view of software architecture as a graph of components and connectors. Specifically, basic elements and relations of a C&C view are:

- **Components** model the principle computational elements of a system's run-time structure. They include things like databases, clients, servers, GUI's, etc. Each component has a set of **ports**, which model the run-time interfaces of that component, through which it interacts with other components (via connectors). For example, a server might have a number of service invocation ports, each port representing a run-time interactions with an individual client.
- **Connectors** model the pathways of communication between components. They include things like pipes and client-server communication links. Connectors may be binary, such as pipes and client-server interactions, or N-ary, such as a publish-subscribe connector, which allows publisher component to interact with zero, one, or many subscribing components. Each connector has a set of **roles**, which model the specifications of behaviour required of the components that use a given connector. For example, a pipe might have a single reading and writing role, while a publish-subscribe connector would have multiple publish and subscribe roles.
- **Systems** model a graph of components and connectors in which the ports of a component fill the roles of a set of connectors to determine the interconnection topology.

Figure 1 illustrates these concepts. In addition, a component or a connector may have substructure (not illustrated here), called a **representation** that further elaborates its internal structure.

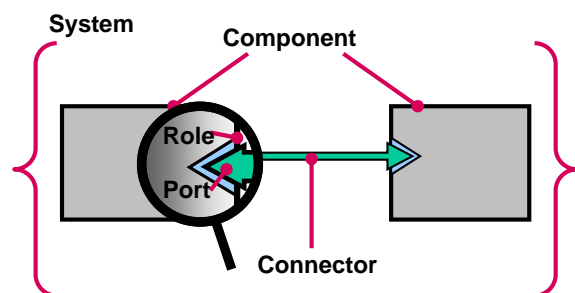


Figure 1. Component and Connector View.

This vocabulary allows one to model the box-and-and-line diagrams common to architectural descriptions, and generally corresponds to the primitive conceptual building blocks in most architectural description languages (ADLs). It is important to note, however, that unlike many informal diagrammatic depictions of architecture, the above model explicitly identifies component interfaces, and represents connectors as first-class model elements of the software architecture.

2.2 Acme

In order to support analysis of component and connector architecture models it is necessary to have a machine-

```

System simple-cs = {
  Component client = { port call-rpc; };
  Component server = { port rpc-request; };
  Connector rpc = {
    role client-side;
    role server-side;
  };
  Attachments = {
    client.call-rpc to rpc.client-side;
    server.rpc-request to rpc.server-side;
  }
}

```

Figure 2. Acme description for a simple client-server architecture.

```

System simple-cs = {
  ...
  Component server = {
    port rpc-request = {
      Property sync-requests : boolean
        = true;
    };
    Property max-transactions-per-sec : int = 5;
    Property max-clients-supported : int = 100;
  };
  Connector rpc = { ...
    Property protocol : string = "aix-rpc";
  }; ...
};

```

Figure 3. Properties in Acme. Analysis of architectural structure.

processable representation. In this paper we use the Acme ADL for this representation (Garlan et al. 2000).

Figure 2 shows an Acme specification of a simple client-server system consisting of a single client and a single server, interacting through a remote procedure-based connector. The system, named *simple-cs*, is declared in the first line of the specification. Following this are declarations of the two components, *client* and *server*, each with a single port (*call-rpc* and *rpc-request*, respectively). The connector, *rpc*, is declared to have two roles (*client-side* and *server-side*). Finally, the system is created by

attaching the appropriate ports to the respective roles of the connector.¹

The textual representation of a graphical picture does little more than provide an alternative depiction. But there are, nonetheless, opportunities for analysis even with such simple models. For example, after parsing, we might check the model to determine whether any connectors have unattached roles, whether every port of a component is attached to some connector. or whether the architectural substructure of a component provides interfaces to

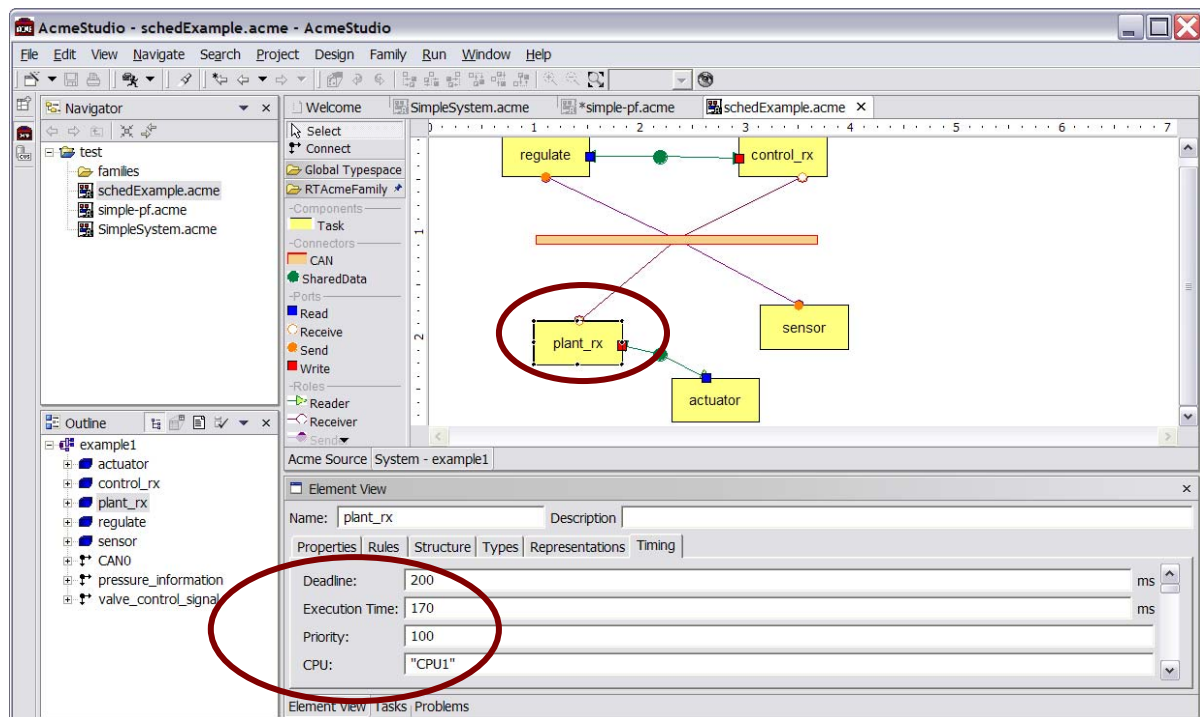


Figure 4. Specifying schedulability properties in AcmeStudio.

¹ Although we don't illustrate it in this simple example, at this structural level we could also provide representations of the

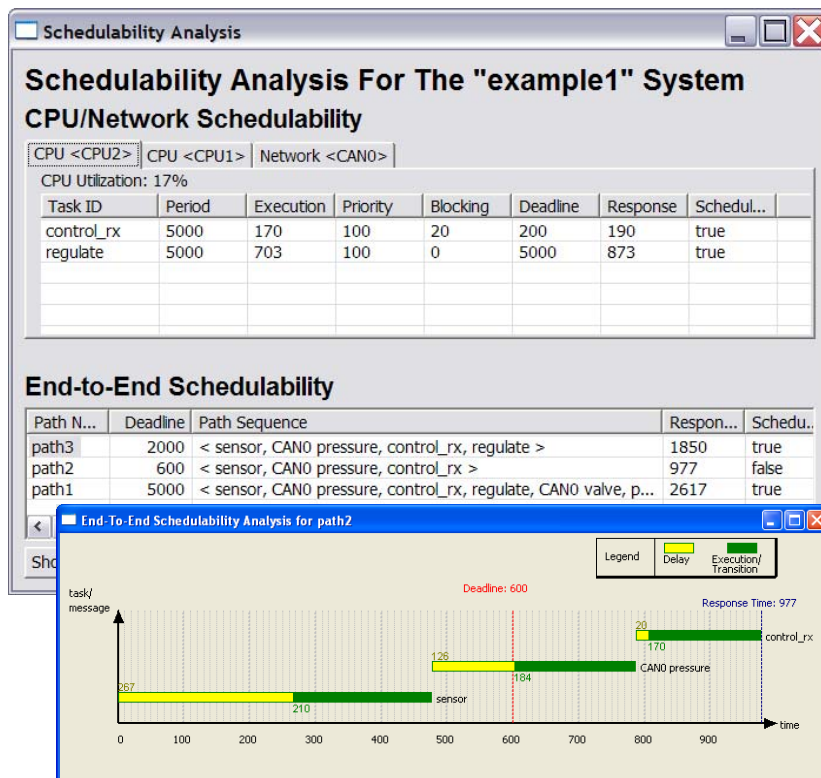


Figure 5. The results of the schedulability analysis.

support its own external interfaces. We can also check for naming conflicts (e.g., whether two ports of the same name on the same component).

3 Modelling architectural properties

While some analyses of pure structure are possible, to achieve significant analytic value from an architectural model we need to represent more of the semantics of the architecture. In Acme this is done by annotating the structure with **properties**.

3.1 Properties in Acme

Properties are simply typed name-value pairs that can be associated with any architectural element.² Types may be primitive (integer, boolean, etc.) or composite (sets, sequences and records).

Figure 3 illustrates the use of properties, elaborating Figure 2. This example illustrates properties associated with a port (indicating whether the client request is synchronous); a component (indicating the maximum number of transactions per second supported by the server), and a connector (indicating the name of the protocol that is expected to be used over it).

client and server, elaborating each component's architectural substructure. See Garlan et al. (2000) for details.

² We use the term architectural element to refer generally to components, connectors, ports, roles, representations, and systems.

3.2 Analysing architectural properties

The meaning of properties is not specified in Acme, which does not provide native support for their analysis. However, such properties can be used by external analysis tools to gain insight into the architecture by calculating global system properties from local properties of components and connectors. In many cases calculations can take advantage off-the-shelf theory and algorithms. Such analyses can be a powerful aid to architectural design, allowing architects to identify design errors early in the process, helping the architect document the expected run-time properties of architectural elements, and facilitating tool support for providing feedback and comparisons of analysis results.

We now illustrate these ideas with three examples: rate-monotonic analysis for automotive control

systems, queuing theory-based analysis for detecting server overloads, and Monte Carlo-style security simulation.

Example 1: Analysis of real-time schedulability,

Figure 4 depicts a simple automotive system represented in AcmeStudio (Schmerl and Garlan 2004), a framework for creating architecture design environments. AcmeStudio, written as a plug-in to the Eclipse framework, permits one to define domain-specific architectural styles³ and link in analysis tools that may be invoked by the user to analyse systems in those styles.

The architecture used in Figure 4 includes components that run as periodic tasks on a set of CPUs. Tasks can communicate directly with tasks on the same CPU, and with tasks on other CPUs using an automotive standard communication bus (here a CAN bus). An important question in the design of such systems is whether certain task scenarios (treated as paths through the architecture), can be scheduled on the available processors.

To evaluate this system-wide property, the style associates with each component a set of properties relevant to real-time schedulability. In the architectural style of this example these properties are modelled as its deadline, execution time, priority, and CPU. For example, in Figure 4 the selected component, plant-rx, has values of 200, 170, 100, and CPU1 as its respective property values.

³ We discuss architectural styles in detail in Section 5; for now, consider a style as providing element types specifying the properties that must be defined for instances of the elements.

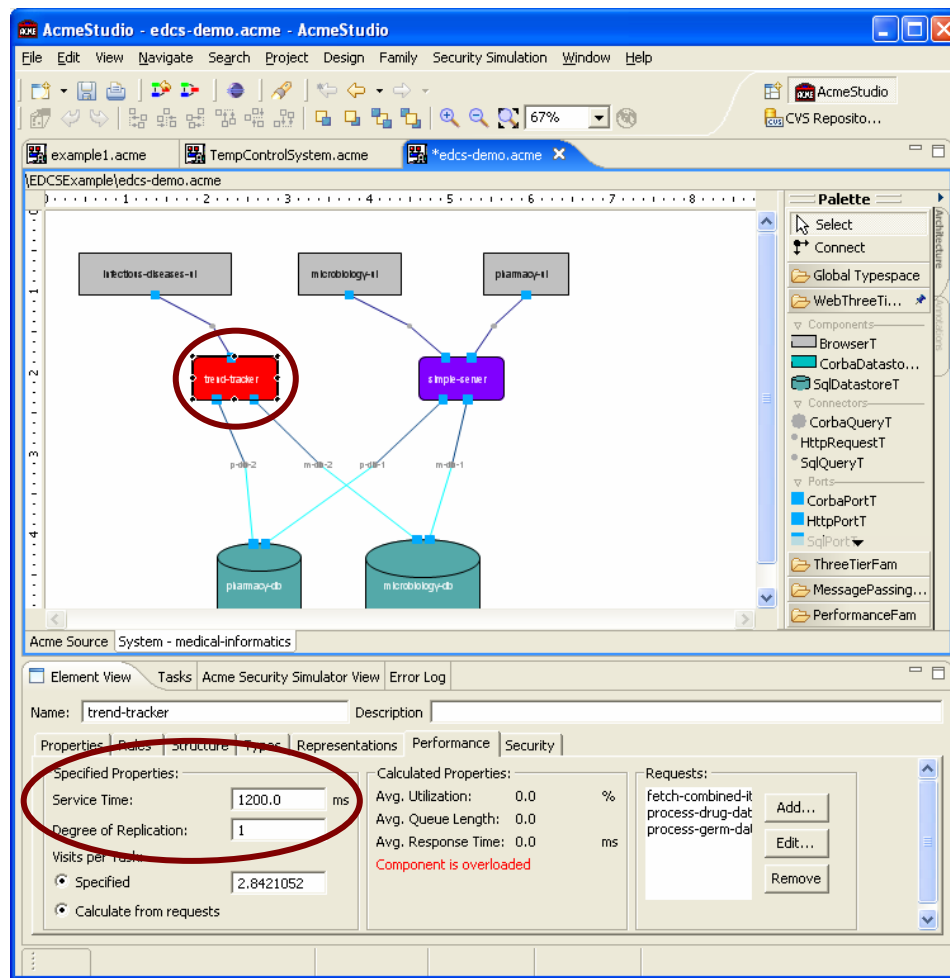


Figure 6. Performance Analysis in AcmeStudio.

When all components have been annotated with these properties (and the connectors with similar properties), we can invoke a tool to evaluate the CPU utilization, and the schedulability of specific pathways. In the figure three pathways are specified. The resulting analysis prints out the results of applying rate monotonic analysis (Sha and Goodenough 1991), indicating which paths are schedulable (Figure 5).

It is important to note that the actual analysis of schedulability is carried out using completely standard, off-the-shelf algorithms for rate-monotonic real-time analysis. Moreover, AcmeStudio makes it relatively easy to add such an analysis using a “plug-in” framework, which assists with creating specialized property editors (e.g., to specify pathways for evaluation), invoking analysis tools through menus, passing the relevant data to them for analysis, and displaying the results back in the graphical editing environment.

Example 2: Analysis of server-load.

Of course, not all systems in need of performance analysis are real-time systems. To illustrate how the same general ideas can be supported for different application domains, consider Figure 6. Here we have an example of a system defined as a tiered system in which clients queue requests for database service from a set of servers that

contain business logic to access a set of databases. The system model is shown in AcmeStudio.

To analyse performance of this system we take advantage of queuing theory to evaluate performance characteristics of such systems (Spitznagel and Garlan 1998, Di Marco and Inverardi 2004). To perform the analysis, we must first supply the values of a set of properties of the components and connectors, such as arrival rates (expressed as probability distributions), average service time for handing requests at a server, and degree of server replication. These properties are specified through an editing plug-in to AcmeStudio specific to performance analysis, as illustrated at the bottom of Figure 6.

Once these properties have been defined, as before we can pass the model to an analysis tool, which in this case calculates for each server a set of derived properties, including average server utilization, queue lengths, and response times using standard queuing-theoretic techniques. From these results the tool can further indicate whether any servers are overloaded. In Figure 6, the analysis has determined that the circled component in the diagram is overloaded, and has highlighted this fact by changing its colour to red.

Example 3: Analysis of security

It is also possible to analyse the security of a system through Monte Carlo-based architectural simulation, a

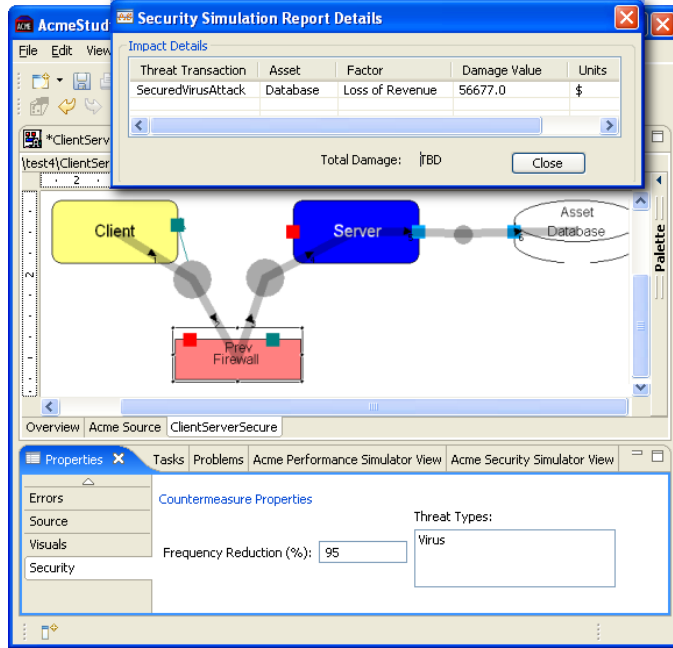


Figure 7. Security Simulation in AcmeStudio.

form of analysis that abstractly exercises an architecture using inputs and events drawn from probability distributions. The Security Simulator plug-in to AcmeStudio enables an architect to perform security simulations based on threat scenarios that are relevant to the system under design. The main concepts in the security analysis are threat types, assets, and countermeasures; the simulation is based on the approach outlined in Butler (2002).

Threat types specify the possible threats that can affect the system (e.g., a virus or denial of service attack). Because different systems may be subject to different types of threats, the architect must specify each of the threat types that may be posed to the system.

Assets are components that may be damaged by particular threats. Assets are assigned a monetary value, and the particular threat types that may affect the asset are specified. For example, a database component may not be susceptible to password sniffing attacks, but may be vulnerable to data corruption as the result of a virus.

Countermeasures are of three types: *Preventative* components affect the frequency at which threats occur; *Monitoring* components and *recovery* components reduce the effect of a threat. The architect specifies each of the countermeasure's target threat types, and the effectiveness or reduction that the countermeasure has on the target threat.

Once the relevant properties are specified, the architect must then define paths (consisting of components and connectors) through the architecture that particular threats may take. The threat type that affects that path and the frequency (as a stochastic function) of the threat type are specified. After the threat is specified, the assets associated with its path can be given outcome values. The outcome can be in terms of dollars, loss of life, loss of pro-

ductivity, etc. A weight is assigned to each outcome factor.

Threat scenarios are composed of one or more *transactions*. A scenario is used as basis for executing the simulation, and specifies the amount of time that will be used in performing the simulation. The simulation takes into account the threat entering the transaction path, the frequency of the threat type and the countermeasures in the path. Monte Carlo simulation is performed to determine the most probable damage value to each of the assets in the threat transaction. The value obtained is multiplied by the frequency of the threat transaction and the simulation time. This gives the total damage for the particular threat outcome factor. The end result of the simulation is a report that details the threat scenario, threat transaction, and total damage to the assets in the threat transaction path.

Consider the simple architecture illustrated in Figure 7, where we define the database as an asset (giving an asset value of \$100K), run a security simulation on a path originating at the client and going through the firewall and server to the database for a simulated virus attack. We define the scenario so that (1) the simulation time is two virtual months; and (2) a virus attack happens on average 5 times per day, with a maximum of 20 attacks per day. If the firewall is 95% effective against virus threats then running the scenario indicates that the damage is calculated as \$56,677. If we were to run the same simulation without the firewall, the simulation will indicate that the loss of revenue increases to \$1,112,409.

Such a simulation allows the architect to evaluate different scenarios, and to evaluate the effectiveness of different countermeasures against different attacks. Providing different sets of properties for an architectural model facilitates different analyses of that model. It is therefore possible to make trade-off based on different scenarios and quality attributes for the same architectural model, rather than have to use different environments and architectural models in potentially different architectural languages.

4 Modelling architectural behaviour

An important aspect of modelling software architectures is the specification of abstract behaviour. By knowing the behaviour of architectural elements we can significantly improve the clarity of architectural designs. We can also analyse these specifications, for example to spot protocol mismatches in which interactions between components can potentially lead to deadlock (Allen and Garlan 1994 and 1997, Allen, Garlan, and Ivers 1998).

To illustrate, consider the simple system consisting of a pipe that connects two filters, F1 and F2, illustrated in Figure 8. The intuition behind such a pipe-filter system is that components communicate through buffered streams,

writing through their output ports and reading through their input ports.

While the intuition may seem simple at first glance, understanding the real meaning of the figure (for example to implement F1 and F2) depends on detailed understanding of the interactions defined by the pipe. For example, from the figure alone it is impossible to answer the following questions:

- Which is the reading/writing end of the pipe?
- Is writing synchronous? That is, assuming F1 is the writer, does it block after writing?
- What if F2 tries to read and the pipe is empty? Does it block, or can it continue with other processing?
- Can F1 choose to stop writing?
- Can F2 choose to stop reading without consuming all of the data on the pipe?
- If F1 closes the pipe, can it start writing again at some future time?
- If F2 never reads, can F1 write indefinitely, or does F1 eventually block?

Note that there is no correct answer to these questions, since any set of answers could represent a possible pipe design. Indeed, in actual systems pipe implementations differ precisely along such dimensions of variability.

What is required is some way to specify the semantics of a pipe at the architectural level so that such questions can be answered easily. This would represent a marked improvement over existing practice in which decisions about such behaviour require one to examine the code of some implementation, existing examples of usage, or consult a human expert.

There are many possible ways in which one might represent architectural behaviour (Shaw and Garlan, 1995). Indeed, practically any behaviour specification will do, including process algebras, state machines, relational models, and timed automata. To illustrate the general principles, we use the Wright specification language, one of the first to use formal modelling to specify architectural behaviour (Allen 1997, Allen and Garlan 1994).

Wright uses a subset CSP (Hoare 1985), a well-known process algebra, which defines behaviour in terms of patterns of events. Some of the constructs are listed in Figure 9. These include *events* (representing architecturally-relevant actions), *processes* (representing patterns of events), *sequentiality* (representing the ability to follow one behaviour by another), *choice* (representing the ability to branch), and *parallel composition* (representing the ability to compose partial descriptions). These CSP-based specifications can be associated with various architectural structures, including ports and roles.

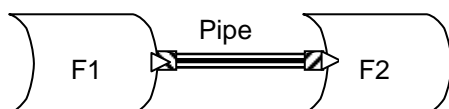


Figure 8. A simple pipe-filter system.

Events:	e, request, read?y, write!5
Processes:	P, Reader, Writer, Client, § (successful termination)
Sequence:	$e \rightarrow P, P ; Q$
Choice:	$P \sqcap Q, P \sqcup Q$
Composition:	$P \parallel Q$

Figure 9. Behavior specifications in Wright.

Figure 10 illustrates the basic ideas of behaviour description in Wright through a partial description of a pipe connector. Each role of the pipe (Reader and Writer) has an associated protocol defined in the subset of CSP summarized above. In addition, the connector has a “glue” specification (also a CSP process) that indicates how the roles interact through the connector itself.

Connector Pipe

Role Writer = (write!x → Writer) \sqcap (close → §)

Role Reader = Read \sqcap Exit

where Read = (read?x → Reader) \sqcap (eof → Exit)

Exit = close → §

Glue = Writer.write?x → Glue \sqcap
 Reader.read!y → Glue \sqcap
 Writer.close → ReadOnly \sqcap
 Reader.close → WriteOnly

where ...

Figure 10. Partial Wright specification of a Pipe connector.

Such specifications, although compact, provide direct answers to questions such as those posed above. For example, the specification in Figure 10 immediately tells us that a pipe writer can close at anytime, but cannot write again once it has close. A pipe reader can also close at any time, but if it chooses to read a value, it must be prepared to recognize an “end-of-file” (eof) marker and then immediately close.

Beyond clarification of design intent, specifications such as these permit a variety of analyses, including:

- Consistency of connectors: that the glue-mediated roles of a connector do not lead to a deadlocked state.
- Compatibility of component interface to connector interaction protocol: that a port satisfies the requirements of a connector role that it fills.
- Consistency of a component’s behaviour with respect to its interfaces: that a port’s specification represents a correct projection of a component’s internal behaviour at that interface point.

Many of these checks can be performed semi-automatically by model checkers. See Allen (1997) for details.

5 Modelling architectural styles

One notable feature of software architecture is the ability to reuse styles and patterns. For example, many systems

```

Family PipeFilterFam = {
  Component Type filterT = {
    Ports {In,Out} ;
    ... ;
  }
  Connector Type pipeT = {
    Role Reader = {Property datatype = ...} ;
    Role Writer = {Property datatype = ...} ;
    Invariant self.Reader.datatype ==
      self.Writer.datatype;
    ...
  }
}
System my-PF-System : PipeFilterFam = {
  Component F1: filterT = {...} ;
  Connector P: pipeT = {...} ;
  ...
}

```

Figure 11. Specification of a Pipe-Filter architectural style in Acme.

are described in terms like “client-server system”, “N-tiered system”, “pipe-filter system”, etc. Such terms refer to families of systems that share a common architectural design vocabulary (e.g., clients, servers, tiers, etc.) and a set of constraints on how that vocabulary can be used (e.g., that clients can’t talk directly to other clients, or that connections don’t cross more than one tier).

Important questions for architectural modelling and analysis are: How can we model an architectural style? How can we check that a given system is consistent with a given style? Can we combine several styles without leading to logical inconsistencies?

5.1 Architectural styles in Acme

We can specify styles by augmenting our architectural modelling notation with two things. First is the ability to define component, connector, and property types. These provide the basic vocabulary of design in that style. Second is the ability to define constraints on how instances of these types may be combined in a system description.⁴

For example, to define a pipe-filter style we would first need to define one or more filter component types and a pipe connector type. These would identify the kinds and number of ports on filters and roles on the connector. Additionally, we might define various property types, and indicate which properties are associated with which elements in the style. Next we would need to define constraints that might, for example, specify that there should be no dangling pipes or that a system should not have any cycles.

⁴ From a tooling perspective style definition may also entail specification of graphical conventions (shape, colour, layout) for the style, style-specific shortcuts for improving graphical editing (such as automatic creation of connectors based on naming conventions), and analysis tools to be included in an environment that uses the style.

Figure 11 illustrates the basic ideas with a partial definition of a pipe-filter style, or *family*, as it is termed in Acme. Here we have defined a *Filter* component type, and specified that it must have at least an *In* and an *Out* port. We have also defined a *Pipe* connector type, and specified that it must have a *Reader* and a *Writer* role, and that each role must specify the datatype that is transmitted through that role.

The connector also includes a constraint, in this case an invariant that says the type of data written to a pipe must match the data read from it. Such specifications are written in a first-order predicate language (similar to UML’s OCL), augmented with some functions that make it easier to refer to things like a component’s ports, or the roles attached to a port.

With the pipe-filter family in hand, we can now use it to define a specific system in that style. In Figure 11 we illustrate the description of a system, *my-PF-system*. Components and connectors may now be declared as instances of the types defined in the family.

5.2 Example: Mission Data Systems

To illustrate the concepts of modelling and analysing style-oriented architectural description in more depth, we now describe a larger example: NASA’s Mission Data System (MDS) (Rasmussen, 2001, Dvorak and Reinholtz 2004). MDS includes an experimental architectural style for defining space systems. It consists of a set of component types (e.g., sensors, actuators, state variables), and connector types (e.g., sensor query). It also defines a number of rules that define legal combinations of those types. Figure 12 graphically illustrates the style, which consists of 7 component types, 12 connector types.

Figure 13 shows a screenshot of a simple MDS system displayed in AcmeStudio. The system represents a temperature control system consisting of a temperature sensor (*TSEN*), a temperature estimator (*TEST*), a heating actuator (*SACT*), a temperature state variable (*CTSV*), a health state variable to indicate whether the sensor is behaving correctly (*SHSV*), a temperature controller (*TCON*) to issue commands to the actuator, and an executive that controls the value of the target temperature (*EXEC*). Appropriate connectors (of which there are 12 types) are used to define the interconnection topology.

The rules in MDS were initially defined in English and had to be hand translated into Acme constraints. A simple example of such a rule is

“For any given Sensor, the number of Measurement Notification ports must be equal to the number of Measurement Query ports (rule R5A).”

This rule, which is a small part of a larger rule (see below) indicates that for every query port that a sensor provides, it must also provide an announcement port (and vice versa).

This rule was translated into the following constraint, which is associated with the sensor component type:

```

numberOfPorts (self, MeasurementNotifReqPortT) ==
  numberOfPorts (self, MeasurementQueryProvPortT)

```

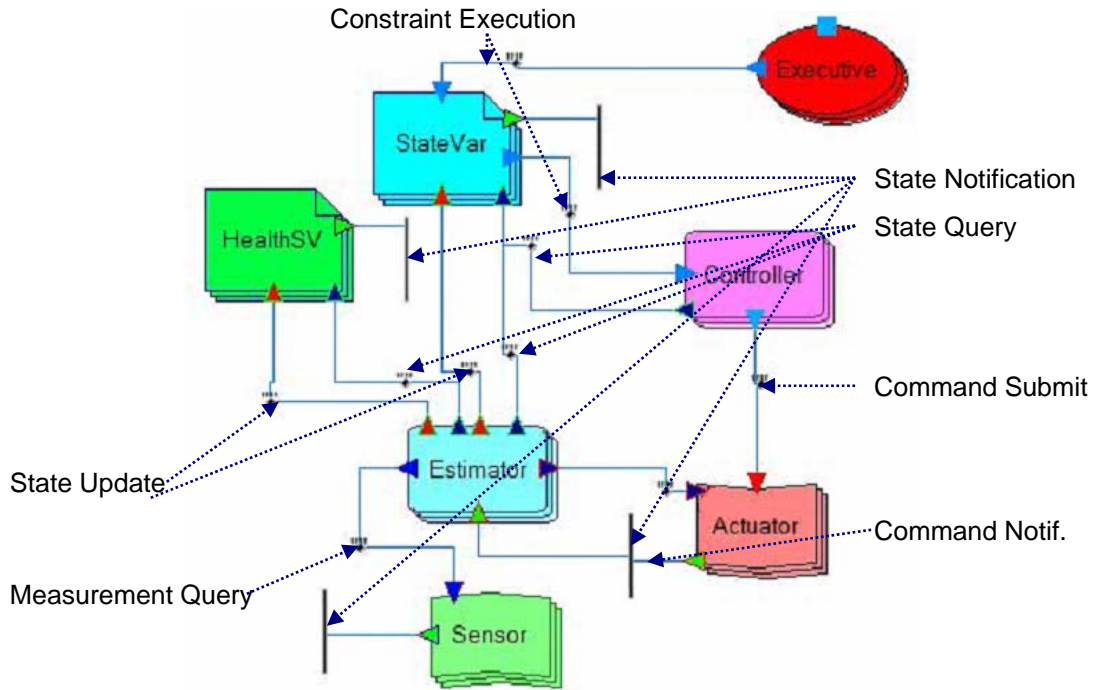



Figure 12. Definition of the MDS Architectural Style.

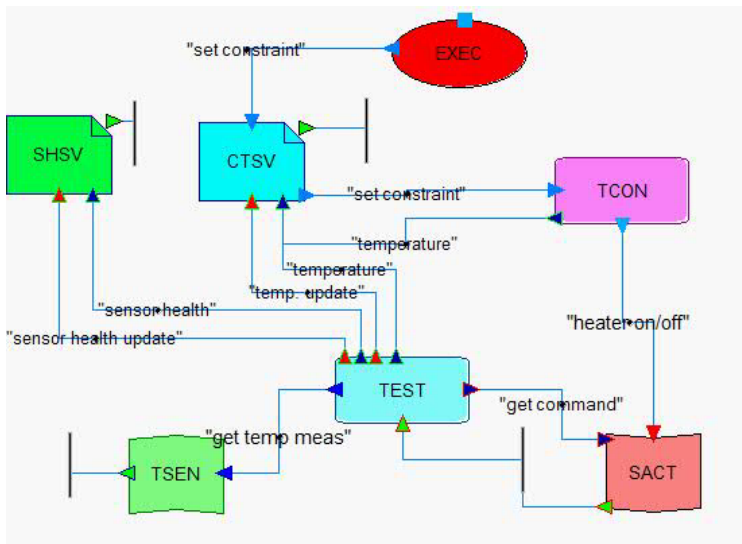


Figure 13. A simple control system in the MDS style.

Rules such as this one are continuously evaluated in AcmeStudio as the MDS architect creates an architectural description of an MDS System. If a rule is violated, the environment highlights the problem. Figure 14 illustrates how this appears to an architect, when the TSEN sensor component violates the property specified above.

Of course, checking rule satisfaction is relatively trivial for small systems and for such simple rules. Indeed, visual inspection could easily locate such rule violations. But in general MDS rules are much more complex, for example:

“Every estimator requires 0 or more Measurement Query ports. It can be 0 if estimator does not need/use measurements to make estimates, as in the case of estimation based solely on commands submit-

ted and/or other states. Every sensor provides one or more Measurement Query ports. It can be more than one if the sensor has separate sub-sensors and there is a desire to manage the measurement histories separately. For each sensor provided port there can be zero or more estimators connected to it. It can be zero if the measurement is simply raw data to be transported such as a science image. It can be more than one if the measurements are informative in the estimation of more than one state variable.”

This is one of 12 such rules. Moreover, MDS architectures typically have hundreds of components. Complete checking of rule satisfaction in those situations becomes a significant problem for which formal style specification provides an effective solution.

5.3 Other style-based analysis

In addition to checking whether a given system conforms to a given style, it is often useful to investigate properties of styles themselves. For example, it is possible to define a style in which constraints lead to inconsistencies. For such systems it is impossible to create *any* system instances. Moreover, we may want to investigate whether the constraints of a style imply properties not explicitly modelled. For example, local constraints on attachments can be used to imply global connectedness.

To evaluate such properties we can interpret an Acme style description as a specification of a class of models, and use a model generator to check for the existence of such models.

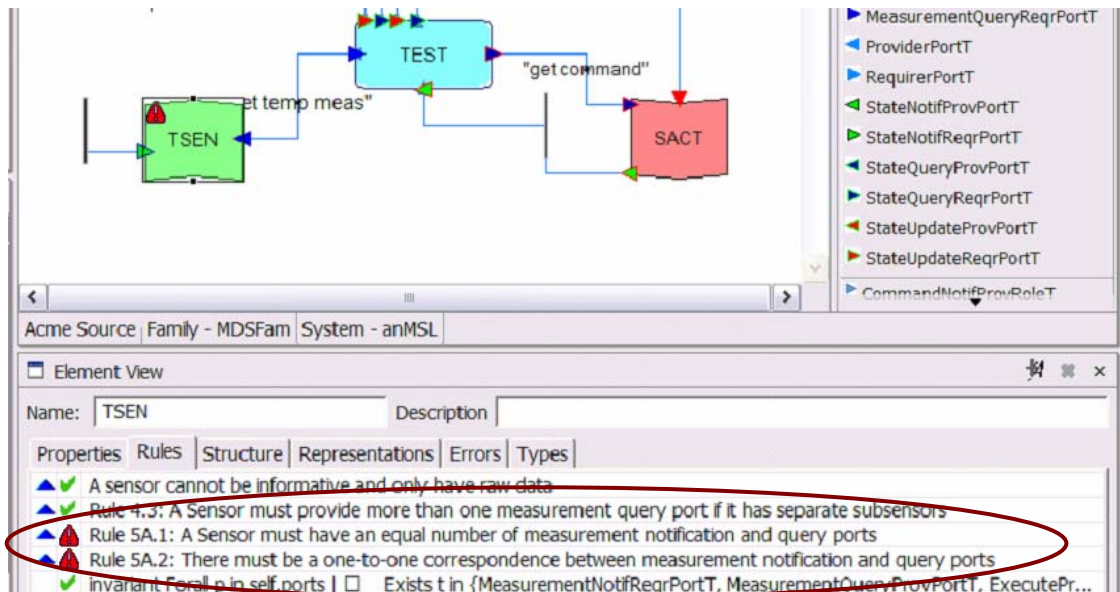


Figure 14. Displaying problems to the architect.

Specifically, we can translate a style into an Alloy model and use the Alloy Analyser (Jackson 2002) to investigate properties of the resulting specification. Details of this analysis are beyond the scope of this paper, but the interested reader is referred to Kim and Garlan (2006).

6 Mapping between architecture and implementation

One of the difficult problems for an architect is ensuring that the implemented system is consistent with the intended architecture. Formal modelling and analysis can also help solve this problem

The problem for architectures is similar to the problem for any model-based method of ensuring that an implementation meets its specification. In general, there are two basic solutions. First, one can attempt to ensure satisfaction *by construction*. This can be done through a process of formal *refinement* in which a concrete model is obtained by applying well-founded refinement rules to a more-abstract model (or specification). Sometimes this process can be completely automated, in which case it is often termed *generation*. The second technique is to demonstrate that a lower-level model is consistent with a

higher-level model by *comparison*. This is often done by providing a *mapping relation* between the two models.

Both techniques can be used for software architectural models.

6.1 Refinement and generation

Although using refinement in the most general case of software architecture is as difficult as any other form of model-based refinement, in many cases the problem is greatly simplified by exploiting architectural styles. That is to say, by limiting the problem to a specific class of systems and a specific class of implementations, it is often possible to build automated assistance for mapping architectures to implementations. The assistance can be in the form of automated transformations, or in the extreme case, code generation of all or part of the target system.

We now illustrate this concept with two examples:

Example 1: Model generation of automotive control systems

Some automotive companies have in place a component-based approach to control systems. Starting with an ab-

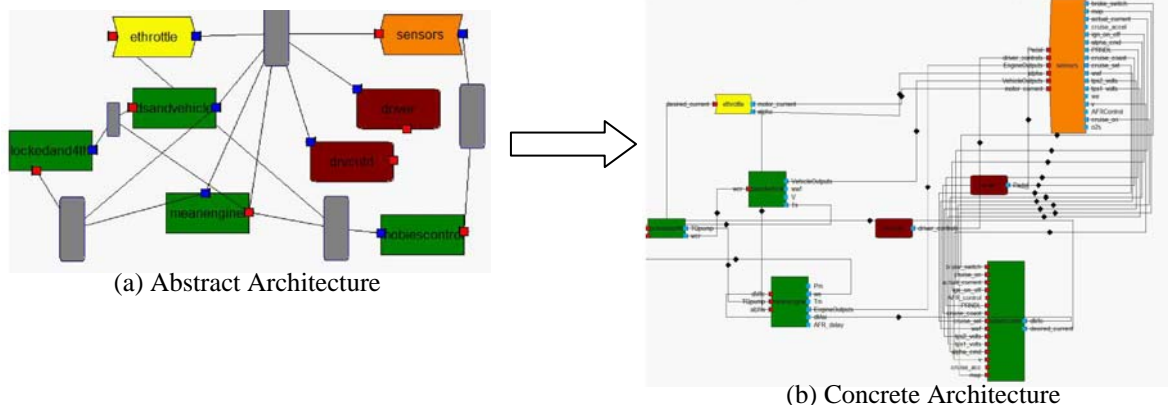


Figure 15. Mapping abstract automotive architecture to concrete automotive architecture.

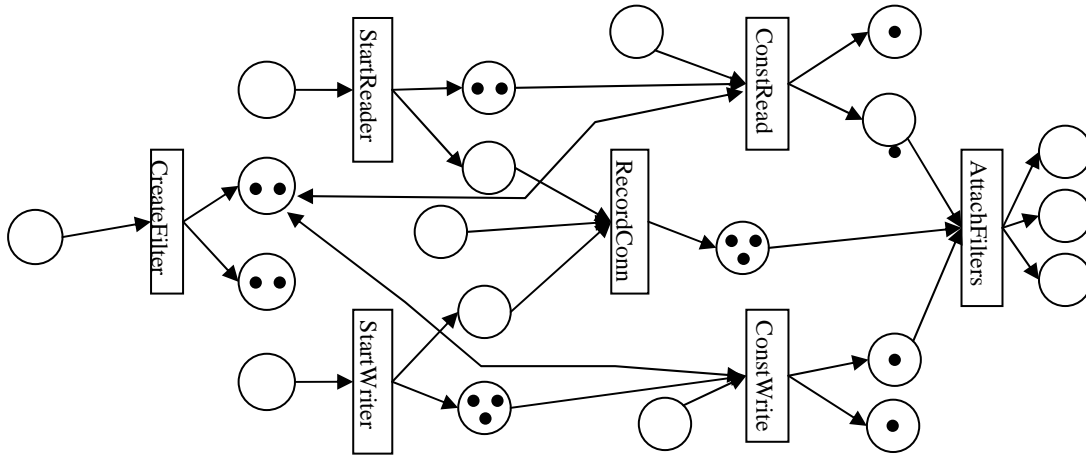


Figure 16. A DiscoTest Coloured Petri Net for Discovering Pipe-Filter Systems.

stract architectural description, pre-specified components drawn from libraries are substituted to produce a full system definition. In many cases the concrete components have formal models suitable for simulation, and in some cases code generation.

In Steppe et al. (2004) we describe a two-tiered approach that uses Acme architectural models of the architecture of an automotive system in two levels. At the higher (abstract) level, an architecture is described in terms of generic abstract components and simple virtual connectors (Figure 15a). In the lower (concrete) level model, concrete components are chosen from a repository of automotive components and substituted for the abstract ones, and detailed connections are made between them (Figure 15b). This concrete composition can then be sent to formal simulation tools for analysis.

While refinement of generic architectures to concrete architectures using component selection is a major step forward, one of the stumbling blocks is that refinement is done manually. In particular, the hooking up of concrete components, which may have dozens of ports is typically a time consuming process. Moreover, there are often dependencies between different components, so that choices of one component may affect others. Making sure that integrity rules of component composition are respected is a difficult, and again time-consuming, task.

However, it turns out that in many cases there are straightforward rules that can be applied to do most of the interconnecting. Indeed, in the case of automotive control systems when certain naming conventions are followed, almost all of the interconnecting can be done automatically. Further integrity rules can be specified as constraints in the style (as illustrated earlier). Indeed, the concrete version of the automotive software in Figure 15b was in fact generated directly using a plug-in to a version of AcmeStudio that had been specialized to model architectures in the two (abstract and concrete) styles.

Example 2: Code generation for MDS space flight systems

With certain modifications to the nature of the connectors in the MDS style we were able to provide a prototype

code generator for MDS systems (Garlan et al. 2005). A key feature of that generator is the ability to target the resulting implementation to different platforms. For example, one platform might be the space environment, which requires power- and space-efficient code, while another platform might be the NASA testing environment in which resources are plentiful and there is a premium on support for debugging and monitoring.

The ability to generate retargetable implementations relies on the following:

1. There is a substantial body of reusable infrastructure code that supports inter-component communication, concurrency, and shared data.
2. It is possible to create a library of component implementations whose processing is not dependent on the implementation of the communication infrastructure. This code treats most components as input-output transformers, where the mechanisms for transporting code between components is irrelevant to the algorithms they implement.
3. There are a small set of attributes that determine the characteristics of the target platform. These attributes include the threading model, the amount and nature of debugging code, the target implementation language, and the task scheduler implementation.

Automatic generation of implementations in this domain allows engineers to work at a relatively high level of abstraction, in which the architectural principles of MDS are a primary focus at all times. The generator guarantees that the resulting implementation is consistent with the architectural model, and moreover does so in a way that is appropriate for the targeted run-time platform on which the system will be executed.

6.2 Direct comparison

The second technique for ensuring compatibility between architecture and implementation is to find a way to compare the two. Since an implementation necessarily has considerably more detail than the architecture, the chief problem to solve is to abstract away the details of the implementation that are irrelevant to the architecture.

Two approaches are typically used. One is to perform *static analysis* on the code to infer high-level structure. The other is to use *dynamic analysis* on the running system to capture actual run-time behaviour and relate it to architectural models. Static analysis is particularly effective for recovering (or inferring) module-oriented structures, since, in general, determining dynamic behaviour of a system (e.g., creating new components or connections) is undecidable. Dynamic analysis is particularly effective for inferring run-time structures, such as C&C views. For that reason we focus on dynamic analysis.

The basic model for dynamic analysis is a process involving a series of steps. First a system is monitored to extract low-level behaviour, such as object and thread creation, method invocation, and variable assignment. Next, low-level, implementation-oriented events are processed to produce high-level, architecturally-relevant events. An architectural model is dynamically constructed by applying the abstract architectural events to an evolving model. Finally, the as-observed architectural model is compared to the as-designed architectural model (or style) to detect inconsistencies.

The main challenge in this process is the abstraction from low-level events to architectural events. This is difficult to do because it may be necessary to observe many low-level events before it is clear what architectural events have occurred. Moreover, these implementation events may be highly interleaved. For example, creating a pipe might involve creating both ends of the pipe and then joining them together. In this process it is possible that many writing ends of a set of pipes are created before any reading end is created.

To account for this complexity we need to define a formal mapping engine. In our own work we have developed the DiscoTect system to do this (Yan et al. 2004, Schmerl et al. 2006). At its core, DiscoTect represents a mapping engine that uses a formal mapping language to describe the relationship between patterns of low-level and high-level events. The output of a mapping description is a coloured Petri net (Jensen, 1994). After some filtering, low-level events enter the net as input tokens. Successive events may cause those tokens to move through the net, eventually emerging as output tokens representing architectural events.

Figure 16 shows the net that creates pipe-filter architectures from Java implementations that use Java pipe libraries, and represent filters as classes that adopt certain naming conventions. The tokens in the figure represent the current state of architectural reconstruction. Specifically, two filters have been constructed, one with a write port and one with a read port, and the pipe connection between them is about to be formed.

7 Related work

As noted in the Introduction, over the past two decades there has been considerable research devoted to modelling and analysis of software architectures (Shaw and Garlan, 1995). This work falls into several categories.

7.1 Architecture description languages

A large number of ADLs and associated toolsets have been proposed by researchers (e.g., Balasubramaniam et al. 2004, Dashofy et al. 2002, Moriconi and Riemschneider 1997, Terry et al. 1995). Like the architectural modelling based on Acme described in this paper, most of these ADLs focus on component and connector structures and their properties. Several of them are specialized to specific architectural styles such as hierarchical publish-subscribe (Taylor et al. 1996), real-time control (Vestal 1996 and SAE International, 2004), or dataflow (Gorlick and Razouk 1991). Collectively they represent an impressive body of evidence about the utility of architectural modelling and analysis.

UML 2.0 by the Object Management Group (2005) provides an architectural modelling language for components and connectors that adopts many of the principles of Acme. However, these extensions are relatively new, and few tools have been developed to exploit them fully. Moreover, as a general-purpose modelling language UML is ill-suited to the problem of supporting domain-specific models that can take advantage of specialized analyses (Garlan, Kompanek, and Cheng, 2002). However, several domain-specific profiles of UML have been proposed or are in the process of being ratified by the Object Management Group. Many of these have the benefits and power of the modelling approaches sketched in this paper.

7.2 Specification and analysis of architectural behaviour

Wright, summarized in this paper, was one of the first modelling notations that attempted to provide behavioural modelling and analysis for software architecture (Garlan, Allen, and Ockerbloom 1994). Since then numerous behavioural formalisms have been used to provide complementary capabilities, including Chemical Abstract Machine (Inverardi and Wolf 1995), PO-Sets (Luckham 1996), Category Theory (Wermelinger 1998), Pi Calculus (Magee et al. 1995), Statecharts (Vieira, Dias, and Richardson, 2001) and many others.

Most of these approaches share the goal of detecting mismatches in component compositions. The primary differences are the kinds of behaviour that can be modelled, and hence the kinds of mismatches that can be detected.

7.3 Refinement and generation

Moriconi and colleagues were among the first to recognize the importance of formal mappings between architectures and implementations (Moriconi et al. 1995). Their approach uses structural transformation patterns to constructively create implementation-oriented models from architectural models.

UniCon, developed by Shaw et al. (1995) supports code generation from architectural models. Their approach creates a set of specialized compilation techniques for the various kinds of connectors that may go into an architecture. The goal is to provide a set of tools where any

change to the implementation of a system must take place through the architecture. Other ADLs also have a certain amount of code generation capability (Luckham 1996, Taylor et al. 1997).

A number of projects have looked at reconstruction of architectures using static analysis. For example, Dali uses a variety of analysis techniques to create a high-level view of a system's implementation structures (Kazman and Carriere 1999). Since they focus on module-oriented views, they are complementary to the C&C-oriented approaches described in this paper.

ArchJava (Aldrich, Chambers, and Notkin 2002) augments Java with constructs for components and connectors, and uses typechecking to guarantee certain kinds of conformance between the component and connector levels of the system description and the lower-level implementation structures (classes, methods, etc.). In particular, the tools can guarantee that if two components are not connected at the architectural level, they cannot directly interact at the code level (e.g., through shared global variables).

A large number of people have become interested in "Model-driven Architecture", an approach that advocates a staged and automated approach to refinement of architectural designs to implementations. This is a natural complement to "Architecture-driven Models" – the theme of this paper. Much of the current work in MDA has focused on a staging in which platform dependencies are abstracted away in the high-level model, and bound during refinement. This is a special case of the approaches to refinement and generation outlined in this paper.

8 Discussion and conclusions

In this paper we have illustrated a number of ways in which formal architectural modelling and analysis can address important issues in software architecture, including clarifying design intent, supporting rich forms of analysis to enable detection of design flaws and make principled tradeoffs between quality of service goals, and allowing tools to help guarantee that implementations are consistent with the intent of their architectures. While the specific techniques described here draw heavily on research carried out by the Able Group at CMU over the past 15 years, many other research efforts have produced similar results.

There are several broad lessons that can be learned from this body of research.

1. **A little formality goes a long way.** The formalisms outlined in this paper are relatively simple. Simple structures with types, properties, relations, and behavioural descriptions can go a long way toward providing more improved capabilities for architectural design. Moreover, formal specification can be incremental: not all aspects of interest need be formalized or analysed.
2. **Reuse of existing methods.** The formal modelling and analysis techniques described in this paper rely on a large body of existing formal methods and tools, including model checkers, simulators, constraint

checkers, and model generators. This is good news for software architects since it means that existing theory and tools can be applied with only minor modifications to the enterprise of software architecture design.

3. **One size does not fit all.** Architecture reveals a classic tradeoff between power and generality: the more general-purpose a model, the fewer opportunities for deep analysis. In our work we rely heavily on architectural style, and our ability to easily create style-specific tools, to exploit specific forms of analysis.

Although our ability to gain insight in software architectures through modelling and analysis has improved tremendously over the past decade, there remain a number of areas for which our techniques need to be improved. These include

- **Dynamic Architectures:** How can we reason about architectures whose structure changes dynamically? How can we determine when architecture changes can be performed safely on a system without restarting it? When can architectural changes be executed in parallel?
- **Software Architectures for Emerging Systems:** As technology advances so does our need to create systems that can take advantage of it. Today, for example, we are on the verge of ubiquitous computing systems that must work in the presence of hundreds of cooperating computational units, from cell phones, to sensors, to traditional computing platforms. What architectures are needed to handle such systems? Similarly, we are starting to see components whose behaviour is determined by machine learning. How can we specify what these components do and ensure that they are compatible with other components?
- **Managing Multiple Views:** So far, much tool support for architectural modelling focuses on a particular view, such as C&C views. How do we manage the relationships among multiple views of an architecture? To what extent can we ensure consistency between these views? How can we separate a particular architectural view into multiple views highlighting different concerns, to manage scalability?

Acknowledgements

The research described in this paper reflects work over the past 15 years funded by a variety of governmental agencies and corporations, including DARPA, NSF, ONR, ARO, Siemens, IBM, HP, and Microsoft. We gratefully acknowledge their support, and note that the opinions, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these funding agencies or corporations.

Numerous students and staff have contributed to the body of work summarized here, including Robert Allen, Elizabeth Bigelow, Shang-Wen Cheng, James Ivers, Jung Soo Kim, Andrew Kompanek, Charles Krueger, Ralph Melton, Bob Monroe, John Ockerbloom, Nicholas Sherman, and Bridget Spitznagel. Their hard work, insight, and

inventiveness are largely responsible for the advances that we have made, and we thank them for their varied but crucial support.

References

- Aldrich, J., Chambers, C., and Notkin, D. (2002): ArchJava: Connecting Software Architecture to Implementation. *Proc. ICSE 24*, Orlando, Florida.
- Allen, R. (1997): A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University School of Computer Science Technical Report CMU-CS-97-144.
- Allen, R., Garlan, D. (1994): Formalizing Architectural Connection. *Proc. the 1994 International Conference on Software Engineering*.
- Allen, R. and Garlan, D. (1997): A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3).
- Allen, R., Garlan, D., and Ivers, J. (1998): Formal modeling and analysis of the HLA Component Integration Standard. *Proc. the 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Beuna Vista, Florida.
- Balasubramaniam, D., Morrison, R., Kirby, G.N.C, Mickan, K., and Norcross, S. (2004): ArchWare ADL Release 1 User Reference Manual. ArchWare Project IST-2001-32360 Report D4.3.
- Bass, L., Clements, P., and Kazman, R. (2003): *Software Architecture in Practice*, 2nd Edition, Addison-Wesley.
- Boehm, B., and Turner, R. (2003): *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison Wesley Professional.
- Bosch, J. (2000): *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*. Addison Wesley.
- Brookes, F. (1975): *The Mythical Man Month: Essays on Software Engineering*. Addison Wesley Professional.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996): *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley.
- Butler, S. (2002): Security Attribute Evaluation Method: A Cost-Benefit Approach. *Proc. 24th International Conference on Software Engineering (ICSE2002)*. Orlando, Florida, pp. 232-240.
- Clements, P., Kazman, R., and Klein, M. (2001): *Evaluating Software Architectures*. Addison Wesley Professional: The SEI Series in Software Engineering.
- Clements, P. and Northrop, L. (2001): *Software Product Lines: Practices and Patterns*. Addison Wesley SEI Series in Software Engineering,
- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002): *Documenting Software Architectures: Views and Beyond*, Addison Wesley.
- Dashofy, E., van der Hoek, A., and Taylor, R.N. (2002) An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *Proc. 24th International Conference on Software Engineering (ICSE2002)*. Orlando, Florida.
- Di Marco, A. and Inverardi, P. (2004): Compositional Generation of Software Architecture Performance QN Models. *Proc. the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*. Oslo, Norway.
- Dvorak, D., and Reinholtz, W.K. (2004): Separating Essential from Incidentals, An Execution Architecture for Real-Time Control Systems. *Proc. 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Austria.
- Garlan, D., Monroe, R., and Wile, D. (2000) "Acme: Architectural Description of Component-Based Systems." In *Foundations of Component-Based Systems*, Cambridge University Press.
- Garlan, D., Allen, R.J., and Ockerbloom, J. (1994): Exploiting Style in Architectural Design, *Proc. of ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*.
- Garlan, D.; Kompanek, A. J.; & Cheng, S.-W. (2002): Reconciling the Needs of Architectural Description with Object Modeling Notations. *Science of Computer Programming* 44, 1, pp. 23-49.
- Garlan, D., Reinholtz, W.K., Schmerl, B., Sherman, N., and Tseng, T. (2005): Bridging the Gap between Systems Design and Space Systems Software. *Proc. 29th Annual IEEE/NASA Software Engineering Workshop (SEW-29)*, Greenbelt, MD.
- Gorlick, M.M. and Razouk, R.R. (1991): Using Weaves for Software Construction and Analysis. *Proc. 13th International Conference on Software Engineering (ICSE13)*. IEEE Computer Society Press.
- Hoare, C.A.R. (1995): *Communicating Sequential Processes*. Prentice Hall.
- IEEE. (2000): *IEEE Recommended Practice for Architectural Description of Software Intensive Systems (IEEE Std 1471-2000)*.
- Inverardi, P. and Wolf, A. (1995): Formal Specification and Analysis of Software Architecture Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering* 21(4).
- Jackson, D. (2002): Alloy: A Lightweight Object Modeling Notation. *IEEE Transactions on Software Engineering and Methodology* 11(2).
- Jensen, K. (1994): An Introduction to the Theoretical Aspects of Coloured Petri Nets. In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): *A Decade of Concurrency*, Lecture Notes in Computer Science vol. 803, Springer-Verlag, pp. 230-272.
- Kazman, R. and Carriere, S.J. (1999): Playing Detective: Reconstructing Software Architecture from Available Evidence, *Journal of Automated Software Engineering* 6(2), 1999.

- Kim, J.S. and Garlan, D. (2006): Analyzing Architectural Styles with Alloy. *Proc. Workshop on the Role of Software Architecture for Testing and Analysis 2006 (ROSATEA 2006)*, Portland, ME.
- Luckham, D.C. (1996): Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events, *Proc. of DIMACS Partial Order Methods Workshop*.
- Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995): Specifying Distributed Software Architectures. *Proc. 5th European Software Engineering Conference (ESEC 95)*.
- Moriconi, M., Quian, X., and Riemenschneider, R. (1995): Correct Architecture Refinement. *IEEE Trans. Soft. Eng.* **21**(4).
- Moriconi, M. and Reimenschnider, R. (1997): Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. *Technical Report SRI-CSL-97-01*, SRI International.
- Object Management Group. MDA (2003): The Architecture of Choice for a Changing World. <http://www.omg.org/mda>. Accessed November 29, 2006.
- Object Management Group (2005). *Unified Modeling Language (UML), Version 2.0*. <http://www.omg.org/technology/documents/formal/uml.htm>. Accessed November 29, 2006.
- Object Management Group (2006). *OMG SysML Specification*. <http://www.sysml.org/docs/specs/OMGSysML-FAS-06-05-04.pdf>. Accessed November 29, 2006.
- Perry, D. and Wolf, A. (1992): Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, **17**(4):40-52.
- Rasmussen, R. (2001): Goal-Based Fault Tolerance for Space Systems using the Mission Data Systems. *Proc. 2001 IEEE Aerospace Conference*, Big Sky, MT.
- Rosanski, N. and Woods, E. (2005): *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison Wesley.
- SAE International. (2004): *Architecture Analysis and Design Language (AADL)*. Document Number AS5506.
- Schmerl, B. and Garlan, D. (2004): Supporting Style-Centered Architecture Development. *ICSE 26*, Edinburgh, Scotland.
- Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., and Yan, H. (2006): Discovering Architectures from Running Systems. *IEEE Transactions on Software Engineering* **32**(7).
- Sha, K. and Goodenough, J. (1991): Rate Monotonic Analysis for Real-Time Systems. *Foundations of Real-Time Computing: Scheduling and Resource Management*, pp. 129-155. Kluwer Academic Publishers.
- Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., Zelesnik, G. (1995): Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, **21**(4):314-335.
- Shaw, M. and Garlan, D. (1995): Formulations and Formalisms in Software Architecture. In Jan Van Leeuwen (Editor), *Computer Science Today: Recent Trends and Developments*. LNCS 1000:307-323, Springer-Verlag.
- Shaw, M. and Garlan, D. (1996): *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.
- Spitznagel, B. and Garlan, D. (1998): Architecture-Based Performance Analysis. *Proc. 1998 Conference on Software Engineering and Knowledge Engineering*, San Francisco.
- Steppe, K., Bylenok, G., Garlan, D., Schmerl, B., Abirov, K., and Shevchenko, N. (2004): Two-tiered Architectural Design for Automotive Control Systems: An Experience Report. *Proc. Automotive Software Workshop on Future Generation Software Architecture in the Automotive Domain*, San Diego, CA.
- Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, E.J., Nies, K.A., Oriezy, P., and Dubrow, D. (1996): A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering* **22**(6).
- Terry, A., London, R., Papanogopoulos, G., Devito, M. (1995): The ARDEC/Tecknowledge Architecture Description Language (ArTek), Version 4. Technical Report, Tecknowledge Federal Systems, and U.S. Army Armament Research, Development, and Eng. Center.
- Vestel, S. (1996): "MetaH Programmer's Manual, Version 1.09." Technical Report, Honeywell Technology Center.
- Vieira, M., Dias, M., Richardson, D.J. (2001): Software Architecture based on Statechart Semantics, *Proc. of the 10th International Workshop on Component Based Software Engineering*.
- Wermelinger, M. and Fiadeiro, J.L. (1998): Towards and algebra of architectural connectors: A case study on synchronization for mobility. *Proc. 9th Workshop on Software Specification and Design*.
- Yan, H., Garlan, D., Schmerl, B., Aldrich, J., and Kazman, R. (2004): DiscoTect: A System for Discovering Architectures from Running Systems, *Proc. of 26th International Conference on Software Engineering*, Edinburgh, Scotland.

Certified Software Factory: Open Software Toolsuites, Safe Methodologies and System Architectures

J. U. Gärtner

Esterel Technologies GmbH

Otto-Hahn-Str. 13b

85521 Ottobrunn-Riemerling

Germany

Abstract

This paper discusses model-based design in the context of the **Safety Critical Application Development Environment (SCADE)**, developed by Esterel Technologies.¹

1. Introduction

The last few decades have seen the concept of model-based design develop to the point where it is now the state-of-the art for most embedded applications. A large number of parallel approaches exist here. Those tools have evolved from pure specification and documentation tools to tool suites allowing design of executable specifications that, in some cases, allow the automatic generation of application code.

These tools can be grouped into several classes, including

- UML-based tools
- Simulation-centric proprietary tools
- Formal tools and methods
- Domain-specific software tools

Two contradictory trends can be observed. Some tool providers follow the path to open standards (such as UML2) or open interfaces and formats (such as Eclipse and XML) and thus enable the user to build his own environment tailored to his needs. Other tool providers hope to be heavy-weighted enough to build their own community based on a proprietary format (for example Simulink, StateMate).

In safety-related systems design, the usage of software design tools is highly recommended. However, the industry trend to automatic code generation is facing some difficulties in this domain, because of the following:

- process integration;
- safety requirements: code generation only pays off if the code generator is trusted by certification bodies; and

- domain-specific solutions lack openness and momentum because they are only deployed in niche areas

There is an obvious need for a solution that combines certified automatic code generation with truly open tool architecture and interface.

We will discuss these topics in the context of SCADE, the **Safety Critical Application Development Environment**, developed by Esterel Technologies.

SCADE provides a modelling environment from which code can automatically be generated, while its open and documented interfaces provide full and seamless integration capabilities into existing development flows and processes.

2. Layered Architecture

The prerequisite for seamless integration in existing or new software design processes is an open, scaleable tool architecture.

When discussing the interface architecture of a core tool, which is intended to be able to provide a hub-like functionality in the flows inside a tool workbench, some requirements soon become obvious:

- Abstraction: the system needs to be layered in a way that on each level provides abstract and encapsulated information;
- Openness: all relevant information must be readily accessible; and
- Standardization: the interfaces must be based on commonly accepted industry standards

This is achieved by implementing a layered tool architecture. An open architecture outside layer provides abstract access to all the information, which is contained in the core.

The core and interface layer together provide the basis for the Certified Software Factory.

Copyright © 2006, Australian Computer Society, Inc. This paper appeared at the *11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, Melbourne. Conferences in Research and Practice in Information Technology, Vol. 69. Tony Cant, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

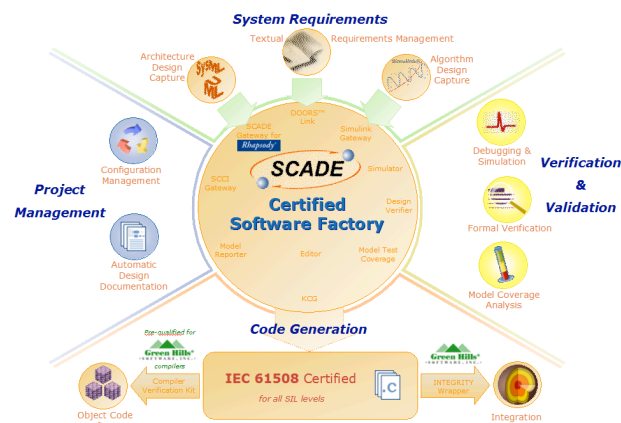


IMAGE 1. Layered architecture of the certified software factory

2.1. Interfaces based on open software architecture concept

Open software architecture interfaces rely on several concepts:

- Standard, openly documented file formats, equally readable by humans and machines
 - SCADE relies on standards such as XMI2, XML and ASAM-MCD2
- Standard, openly documented APIs (application programming interfaces)
 - SCADE provides TCL and C-based interfaces as well as an Eclipse-Plug-in.
- Models are stored as Meta-models so that they can be transformed to and from any other model format
 - SCADE stores the information in an UML-Metamodel

2.1.1. Example: SysML interface

When transforming models, one must take care to do meaningful translations conforming to the semantic properties of the underlying modelling languages.

Building such an interface requires analysis of the data formats as well as the semantics.

SysML	SCADE
<ul style="list-style-type: none"> • <u>Overview</u> <ul style="list-style-type: none"> • Semi-formal • Asynchronous • Object-oriented • Good for structural description • <u>Main construct</u> <ul style="list-style-type: none"> • Classifier & Behaviour • <u>Dynamic</u> (Instance/link creation) • <u>Explicit & implicit flows</u> (connectors or object references) 	<ul style="list-style-type: none"> • <u>Overview</u> <ul style="list-style-type: none"> • Formal • Synchronous • Functional with state • Good for behavioural description • <u>Main construct</u> <ul style="list-style-type: none"> • Node (close to UML behaviour) • <u>Static</u> (everything pre-instantiated) • <u>Explicit flows only</u>

IMAGE 2. SysML and SCADE semantic comparison

Deeper analysis shows that the SCADE and SysML notations are very complementary. The ideal pivot point for model transformations is the class/ node interface. When concentrating on this construct, the user gains a hybrid view on the overall model: a dynamic, object-oriented view of the model architecture linked with a static, instantiated and synchronous view on behaviour.

If such a model translator is additionally based on OSA (open software architecture) concepts and commonly accepted standards such as XMI2 and a meta-model approach, it can easily be built in a very generic way, allowing adaptations for all kinds of UML2/SysML dialects and specific profiles.

The SCADE Gateway to Rhapsody® is an instance of such an implementation.

2.1.2. Example Requirements management interface

Requirements are usually formulated in textual form and stored either in a database or in text processing tools.

Requirements may be further refined and result in software or system design, CAD drawings or other format.

An open development platform must therefore provide a means to link requirements specifications (in whatever format) with designs and models, test cases or source code (in whatever format).

The SCADE requirements management gateway enables the user to link all his tools and data together and have instant and global understanding of the interdependencies and relationships.

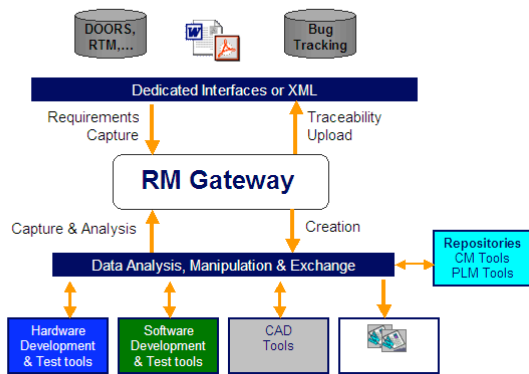


IMAGE 3. Requirements management gateway providing traceability throughout the entire life-cycle

2.2. The core of the certified software factory

At the centre of the software factory, there is a repository containing the information that describes the behaviour of the software.

On the one hand, this model complies with the notion of an UML-Metamodel, meaning that the contained information can readily be accessed through a standardized interface (script language or Eclipse).

On the other hand, the model must obey very strict requirements in order to comply with the requirements imposed by the standards that drive safety-related systems development: DO-178B, IEC61508-1 and -3, EN50128.

High integrity levels imply formal models and unambiguous semantics that allow representing the typical features of embedded software systems: reactive systems with data flow, discrete states and concurrency, coupled with hard real-time constraints.

The SCADE modelling language has evolved from LUSTRE, a formal, synchronous model description language.

The user interface provides the developer with a very intuitive view, based on block diagrams and state charts, tightly integrated. Powerful constructs for vectorization of flows and operators tackle even the most complex problems.

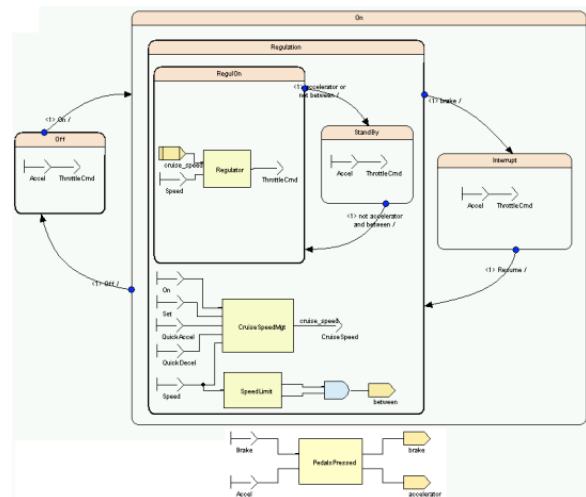


IMAGE 4. SCADE model representing a fully functional automotive cruise control application

This model is immediately executable for verification and validation purposes.

The development environment includes a powerful software- in- the- loop simulator with model- level debugging features.

Thanks to the formal nature of the model, it can also be examined by formal/mathematical analysis and proof engines, such as the integral SAT-solver *Design Verifier*, which provides a formal proof of functional safety properties.

The open software architecture makes this model fully accessible through the customer's specific tool suite and provides transformation engines to and from this environment.

It is also the basis for automatic generation of SDD documents (software design descriptions) and, more importantly, serves as direct input for certified code generation.

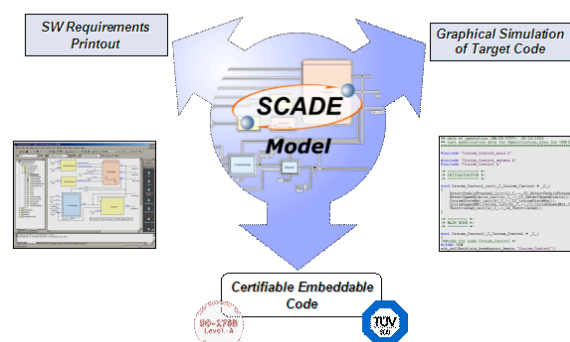


IMAGE 5. Formal model as the hub of the SW design process

Certified code generation ensures, that

- The code complies 100% with the model in the sense that the code fully and deterministically represents the behaviour described in the model

- The generated code complies with each and every objective and requirement imposed by the standards to which it has been qualified (DO-178B up to Level A) and certified (IEC61508, up to SIL 4)

For example, the generated C code does not contain operations on pointers, no global variables, no indefinite loops, no dynamic memory allocation etc.

At the same time, it fulfils very stringent requirements related to memory usage and execution time.

It is absolutely comparable with highly optimized hand-written code.

Moreover, it is totally target-agnostic and therefore easily to be integrated on all platforms, from bare machine to complex distributed systems.

Certified code generation is a key to a completely defined process that covers all steps from requirements capture down to integration on target.

High quality of generated code and restriction to a very small subset of C allow also to verify correct compilation through compilation and automated verification of a representative model containing the complete generable subset of C in all its possible combinations and nested operators, resulting in a combined testing process which ensures and guarantees that each requirement is correctly designed, modelled, coded, and the integrated on the target hardware.

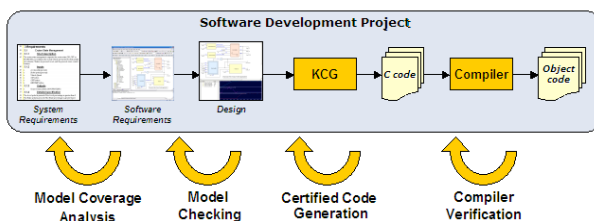


IMAGE 6. The combined testing process

3. Safe Systems Architectures

The tool suite and process outlined above ensure that no systematic errors can be introduced into the software design when transforming the system requirements relevant for software into an application.

Safe system architecture needs to also ensure that hazards or spontaneous, non-systematic errors on the hardware, sensors or from the environment will not affect safe operation of the system.

Various approaches exist to this problem, some of which include redundancy, dissimilarity and built-in tests.

All of them go beyond the scope of this paper, but share the same principle: A layered systems design that

clearly separates the application from the hardware and usually incorporates a safe and certified operating system.

An open software development environment must directly support automatic integration of the generated code onto such safe HW/SW platforms.

SCADE provides such interface to several certified/qualifiable operating systems such as GHS Integrity, Sysgo PikeOS or MicroC.

On proof-test intervals for safety functions implemented in software

Alena Griffiths

System Safety & Quality Engineering Pty Ltd
11 Doris Street, Hill End. Qld. 4101

alenag@uqconnect.net

Abstract

Given a target probability of functional failure on demand for a system, a corresponding dangerous failure rate for the system can be derived, provided that a proof-test interval for the function is known. IEC 61508, and related standards, requires that this calculation be performed, for certain kinds of systems that are required to provide safety functionality on demand. This paper explains why it is necessary to consider what constitutes a proof-test interval for a function, and then considers what this means for software. We show that there are several problems with the proof-test concept, as applied to software, and describe the problems this presents to practitioners wanting to derive safety integrity levels for system safety functions.

Keywords: SIL derivation, IEC 61508, high and low demand mode functions, proof tests.

1 Introduction

In many industries – rail, chemical process control, oil and gas, motor vehicle, nuclear, and to a lesser extent, defence – the approach to safety engineering is strongly influenced by the risk-based approach outlined in IEC61508 (1998–2000), and other standards based upon it.

These standards describe an approach to deriving safety targets in which the safety target assigned to a function corresponds to the risk reduction required to be achieved by it. Depending on whether the system can initiate an accident sequence, or is required to provide protection against other hazards that could occur, safety targets will be expressed as a dangerous failure rate, or as a probability of failure on demand, respectively. Depending on the technology used to implement the function, these targets may then be used to infer safety integrity levels (SILs) for the development of the function.

This paper describes the SIL derivation process with reference to an accident sequence model. We argue that the issue of whether a function's safety target should be expressed as a failure rate or as a probability of dangerous failure on demand, should not depend on its classification as a high or low demand mode function, but rather on its position in the accident sequence. We also show that, if it is desired to transform safety targets to uniform

dimensions and thus to compare them (i.e. to express all safety targets in terms of dangerous failure frequencies, for example), the issue of what constitutes a valid proof-test interval for the function *must* be considered. To the extent that a function is implemented in software (or software-like technologies), this means that the concept of proof-test intervals for software must also be considered. It also means that safety target derivation, if it goes so far as to assign SILs, is not a process that can be independent of the technology used to implement the safety functions. This is disappointing, as intuitively, we would like safety requirements derivation, and safety requirements implementation, to be separable concepts.

The concept of a proof-test interval, as it applies in the field of reliability engineering, is then reviewed. We consider what this means for functions implemented in software. We survey various arguments that might be offered to justify the selection of different proof-test intervals – noting their strengths and limitations – and conclude that there is no consensus about what is a valid proof-test interval for software.

Although this paper uses an accident sequence model to explain the relevance of proof-test intervals to the SIL derivation process, in fact the proof-test issue arises in any framework in which quantitative safety targets need to be achieved with technology whose integrity is difficult to quantify. This suggests that the SIL concept either needs to be expanded to better correspond to the two types of statistic used to express safety targets, or that assignment of SILs should be deferred to later in the development life-cycle. The pros and cons associated with both approaches are briefly surveyed.

2 Accident Sequence Models

The process of determining safety requirements for a system commences with a hazard identification and analysis activity. There are many techniques for performing such an activity, and it is beyond the scope of this paper to discuss these. However, at the conclusion of the hazard identification and analysis phase, one should have identified all reasonably foreseeable ways that a state or event of the system can cause or contribute to an accident.

One way to document one's understanding of how the system can lead to accidents is to prepare a set of accident sequences.

A typical accident sequence is illustrated in Figure 1.

“Copyright © 2006, Australian Computer Society, Inc. This paper appeared at the *11th Australian Conference on Safety Related Programmable Systems (SCS '06)*, Melbourne. Conferences in Research and Practice in Information Technology, Vol. 69. Tony Cant, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.”

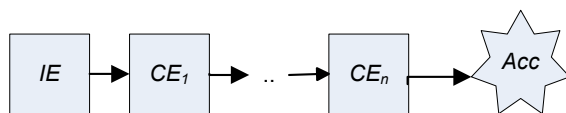


Figure 1 Accident Sequence

As shown in Figure 1, a typical accident sequence consists of:

1. An initiating event (*IE*), which is the event that initiates the sequence of events that can lead to an accident.
2. Zero or more contributing events ($CE_1 \dots CE_n$). A contributing event is an event that must occur, in order for the accident to result.
3. An accident (*Acc*), which is an event involving harm.

Depending on the way a safety program has been scoped, the concept of harm may be limited to harm to humans, but it could also be expanded to incorporate damage to property or the environment, service outages, or mission failures.

Accident sequences may be documented pictorially – as in Figure 1 – however it is usually more efficient to document them in a tabular fashion. Tables 1 and 2 illustrate how this can be done, and also describe the accident sequences for two working examples we will use throughout this paper.

The first example is taken from a risk analysis for a traditional railway signalling system. The signalling system receives (non-vital) requests to set routes. On receipt of a request, it checks the safety of the request against (vital) inputs from track circuits, and against any relevant interlocking history. If it is safe to do so, it sets the route, which may involve issuing vital outputs to points and signals, and storing a record of this authority (i.e. updating the interlocking history). The accident sequence in Table 1 below examines the consequences of unsafe failure of the interlocking system, leading to a system allowing green signals to be displayed over a common section of track.

Event ID	Event Description
IE (*)	Interlocking shows proceed aspects on conflicting routes, A and B.
CE ₁	Driver on route A, on the strength of the proceed aspect, moves into a section X.
CE ₂	Driver on route B, on the strength of the proceed aspect, moves into section X.
CE ₃	Driver on route B fails to notice presence of A, in sufficient time to slow train to avert a collision.
Acc	Train collision/derailment

Table 1 Signalling Example

In the example above, the event labelled *IE* is annotated with a (*), to indicate that it is an event of the system under

analysis. This is a convention we adopt throughout the paper.

The second example is taken from a risk analysis for a tunnel ventilation control system (TVCS) for an underground railway. The TVCS is responsible for, among other things, activating powerful exhaust fans to clear air in tunnels, in the event that it becomes contaminated by smoke or gas. Activation of such fans is usually triggered automatically, in response to stimuli received from a train supervision system, and a fire alarm system. However, in the author's experience, it is usually the case that in the event of failure of the automatic activation system, the fans can be manually activated, either indirectly, via a computer-based display on the TVCS, or directly, via a hard-wired panel in a plant room. The latter control point usually bypasses the control system completely. In the following example, we assume the existence of a hard-wired manual control location.

Event ID	Event Description
IE	Train stops in tunnel
CE ₁	Fire in tunnel/station
CE ₂	Air supply in tunnel contaminated as a result of smoke or noxious gas
CE ₃ (*)	Control system fails to automatically activate exhaust fans, in response to train stoppage/fire alarms
CE ₄	Delayed manual fan activation leads to period of diminished air quality for persons in train
Acc	Persons in train suffer poisoning/asphyxiation

Table 2 Tunnel Ventilation Example

Note that in both examples given above, the accident sequences are extreme simplifications of the analyses that would occur for real systems, and are provided for illustrative purposes only. Also, the accident sequences focus on events of interest to the system under analysis. For example, in the tunnel ventilation example, failure of the exhaust fans is an alternative cause of the final accident, and potentially has far more serious consequences, since it would lead to total ventilation failure, not just delays associated with failure to activate the ventilation automatically. Nevertheless, fan failure would not generally be considered in much detail in a study aimed exclusively at deriving safety targets for the TVCS, because the fans do not mitigate failures of the TVCS¹.

In both examples presented above, exactly one of the events *IE*, $CE_1 \dots CE_n$ in the accident sequence corresponds to a hazard of the system under analysis. Sometimes,

¹ Note that the failure rate of the fans does, however, constitute a practical upper bound on the reliability of the function as a whole, and so might constitute an upper limit on the integrity of the TVCS that should be targeted.

however, more than one event of the accident sequence will correspond to the system under analysis. For example, where a system has diagnostic and back-up facilities, then the initiating event might correspond to a critical failure of the system, and subsequent contributing events might be failure of the system to detect the critical failure and take protective action.

The accident sequence can then be used to derive safety requirements for the system under consideration, in that any event corresponding to an action or inaction of the system gives rise to a corresponding safety requirements on the system – i.e. that the event ought not to occur.

So, returning to the signalling example, a safety requirement corresponding to event *IE* might be:

Requirement 1: *The interlocking system shall not command a proceed aspect unless it is safe to do so (having regard to the current state of vital inputs and the history of the interlocking).*

Similarly, for the tunnel ventilation example, the safety requirement associated with *CE₃* might be:

Requirement 2: *The TVCS shall, in response to signals indicating that a train is stopped in a tunnel, and that a fire alarm is active, automatically initiate emergency ventilation mode.*

Note that, in addition to providing information about safety requirements for the system under analysis, accident sequences also contain other information about the safe deployment of that system. For example, *CE₄* in the tunnel ventilation example documents an important assumption that underpins the analysis – namely that there is an alternative control location, and that staff are trained to operate fans from that location. In general, accident sequences should be comprehensively mined for other, safety-related information, which should be encoded as third-party requirements, assumptions, application-specific safety conditions, etc.

Note that accident sequences are not the only way to document how system hazards can contribute to accidents. Event tree analysis, fault tree analysis and cause-consequence analysis are other techniques that achieve a similar goal. These methods, however, are ways of more concisely presenting groups of related accident sequences, rather than being entirely different techniques.

The accident sequence model has been criticised in work by Leveson (2002), on the grounds that it focuses on the (most) proximate causes to the accident, often at the expense of underlying systems-theoretic causes (for example – lack of a management commitment to safety in the organisation operating the safety-critical plant). Also, it may lead one to a mistaken belief that events are independent, because they are associated with different systems, when in fact there may be a common underlying cause (e.g. if both systems are maintained by the one organisation, which has applied aggressive cost-cutting measures to its maintenance operations). Further, the desire to quantify the models, an issue we explore in the next section, may lead one to unconsciously discount events from an accident sequence, simply because quantifying the likelihood of occurrence is hard.

These criticisms are accepted, and, if constructing an accident model for an operating railway as a whole, we might well prefer to adopt the systems-theoretic approach recommended by Leveson (2002). However, in the author's practical experience, safety requirements derivation is typically performed in a much more specific context – i.e. it usually falls to the supplier of the TVCS to make the case for derivation of appropriate safety requirements and targets. Persons working at that specific context typically do not have access to detailed information about say, the railway as a whole, and must instead make informed judgements about what to expect of the surrounding environment, which form caveats on the safe use of the system they supply. It then falls to parties at the railway level to ensure that the assumptions made by the TVCS suppliers are appropriate in the context of their specific railway. Also, the criticisms advanced by Leveson (2002) are not faults exhibited by all sets of accident sequences. They are just common traps into which developers of accident sequences might fall. An awareness of these issues, and subsequent review of the accident sequences to search for these weaknesses, can help to avoid them.

3 Safety Target Derivation

In addition to identifying the system safety requirements, in a functional sense, accident sequences can also be quantified and as such used to derive quantitative safety targets.

To see how this can be done, it is first necessary to introduce the mathematics behind accident sequences.

If $F(X)$ denotes the frequency of event X , and $P(Y)$ denotes the conditional probability that Y occurs, given that preceding events in the accident sequence have already occurred, then the frequency of accident occurrence can be computed using the following formula:

$$F(Acc) = F(IE).P(CE_1). \dots .P(CE_n) \quad \text{Eqn 1}$$

To derive safety targets one then applies the following process.

First, the severity of the accident must be assessed, and a target rate of occurrence for the accident must be derived. How to do this is beyond the scope of the paper, however we note that:

1. Usually, one has regard to a risk matrix that indicates risk tolerability as a function of accident severity and frequency of occurrence – see for example the matrix in Clause 4.6.3.4 of EN 50126.
2. Account must be taken of the aggregate risk posed by the system as a whole, not simply the risk posed this particular accident sequence.
3. Depending on the industry in which one is working, the final arbiter of what constitutes acceptable risk will vary, as can the risk standards that will be imposed.
4. If it is necessary to demonstrate that risk has been reduced as low as reasonably practicable

(ALARP), it will be necessary to show that the costs associated with achieving a lower target accident frequency are disproportionate to the risk benefit that would be achieved. This may require the safety target derivation process to be iterated a number of times, with indicative costings prepared for different derived targets.

Let us suppose that, for the signalling example, the tolerable frequency of a rail collision arising from this specific accident sequence (having regard to the nature of the rail service, the volume of rail traffic, and the number and type of other accident sequences that can give rise to a similar result) is $1\text{E-}05/\text{year}$ or $1.14\text{E-}09/\text{hr}$.

Further let us suppose that for the tunnel ventilation example, a similar consideration yields a target of $3.42\text{E-}09/\text{hr}$.

Next, one assigns values to the parameters of Eqn 1, for events that are outside the system of interest. This will require data collection and the application of engineering judgement, and often, will require some assumptions to be made.

Thirdly, for events that are hazards of the system under analysis, one assigns conservative values, for the purposes of initial assessment *only*, in the following way.

If the event is a contributing event, then a probability of 1 is assigned, indicating that initially, we pessimistically assume that the contributing event will always occur when it is dangerous for that to happen.

If the event is an initiating event, then it is usual practice to assign a failure frequency, *for the purposes of initial assessment only*, that is outside the claim limit for SIL 1.

Per IEC 61508, the failure rate range for SIL 1, high-demand mode systems, is less than $1\text{E-}05$ failures/hour, but greater than or equal to $1\text{E-}06$ failures/hour. A value of $3.2\text{E-}05$ failures/hour is therefore one order of magnitude more frequent than the geometric mean of the SIL 1 range. It is suggested to be an appropriate value to use, for the purposes of initial risk assessment only, however it is important to note that this does give rise to a later validation obligation to show that such reliance is reasonable. In particular, we note Clause 7.5.2.4 of IEC 61508-1 which allows dangerous failure rate claims to be associated with systems that are not designated as safety-related systems, provided that the rate claimed is higher than the SIL 1 failure rate boundary, and that the claim can be justified.

The process described above is applied to the signalling example and the tunnel ventilation example, as shown in Table 3 and Table 4 below. The resultant frequency assessments are also illustrated there.

It is clear that for both examples, the initial accident frequency calculation results in a frequency much higher than the maximum allowable value, and that risk reduction is therefore needed.

In the context of the accident sequence model, risk reduction can occur in a number of ways, including:

1. Redesign so that the accident sequence is no longer possible.
2. Introduce measures to reduce the severity of the resulting accident.

Event ID	Event Description	Initial Estimate	Rationale
IE (*)	Interlocking shows proceed aspects on conflicting routes, A and B.	$3.2\text{E-}05/\text{hr}$	The initial estimate is purely a placeholder value -- for a system of lower than SIL 1 integrity.
CE ₁	Driver on route A, on the strength of the proceed aspect, moves into a section X.	1	Drivers proceed on the strength of the authorities communicated via signals. This event is a natural consequence of the initiating event.
CE ₂	Driver on route B, on the strength of the proceed aspect, moves into section X.	1	Drivers proceed on the strength of the authorities communicated via signals. This event is a natural consequence of the initiating event.
CE ₃	Driver on route B fails to notice presence of A, in sufficient time to slow train to avert a collision.	1/2	Drivers are expected to remain alert and to look for hazards on the track. Depending on the conditions at the time, the speed of the traffic, the topology of the track, some credit may be taken for the fact that the driver on route B may notice the presence of A and slow his train to avoid collision (or to lessen the impact).
Acc	Train collision/derailment		A train collision or derailment is always assumed to have catastrophic effects.
Accident frequency ==>		1/7 years	

Table 3 Signalling Example – Initial Quantitative Risk Calculation

3. Introduce additional mitigations, that would prevent accident occurrence – this corresponds to adding additional events to the accident sequence.
4. Reduce the frequency of the initiating event, or the conditional probability of one of the contributing events, to reduce the accident frequency – this usually means assigning integrity requirements to one or more of the systems involved, although a similar result can be achieved by limiting hazard exposure periods.

As noted above, this paper is written from the perspective of a supplier seeking to derive appropriate safety integrity requirements for a system that he has been tasked to build. In this context, the options available to the supplier are usually limited to the last two. In relation to Option 3, a supplier can introduce redundancy into the design of the

system he has been tasked to build (with consequent increases in cost and complexity, and note that redundancy offers no protection against common-cause failure modes, particularly those arising from design error). In relation to Option 4, the supplier can typically only alter the safety integrity requirements of the system he is supplying.

The remainder of this paper focuses on achieving risk reduction through the assignment of safety integrity requirements to the system under consideration, i.e. Option 4 above.

For the signalling example, as the initial accident frequency is once in seven years, or $1.6\text{E-}05/\text{hr}$, a failure frequency of $2.28\text{E-}09/\text{hr}$ must be assigned to the initiating event, in order to reduce the accident frequency to the tolerable value.

Event ID	Event Description	Initial Estimate	Rationale
IE	Train stops in tunnel, and cannot be readily restarted	1/2 years	While short term delays are common, and any of these might cause a train to be stopped in a tunnel, lengthier delays requiring passenger evacuation are far more infrequent.
CE ₁	Fire in tunnel/station	1/100	While very minor fires in station waste bins and the like are common, fires large enough to give off significant amounts of smoke are extremely uncommon in stations, and the system is designed so these should not occur in tunnels.
CE ₂	Air supply in tunnel contaminated as a result of smoke or noxious gas	1	The estimate of conditional probability, in the previous event, takes "credit" for the unlikelihood of fires large enough to emit tangible quantities of smoke. It is therefore reasonable to assume that if such a fire did occur, the air supply in the tunnel would be contaminated.
CE ₃ (*)	Control system fails to automatically activate exhaust fans, in response to train stoppage/fire alarms	1	Initially, no credit is taken for control system integrity, as this is the system under consideration.
CE ₄	Delayed manual fan activation (5 minutes) leads to period of diminished air quality for persons in train	1	If the events preceding this one have all occurred, then staff would have no option but to attempt to action emergency mode from the plant room. Staff are trained for this eventuality and the target for manual activation is less than 5 minutes. A period of diminished air quality would be expected.
Acc	Persons in train suffer poisoning/asphyxiation		It is expected that a period of diminished air quality could result in as many as five deaths, for passengers with pre-existing conditions, and cause other major/minor injuries, with significant distress to all passengers involved. This is therefore a catastrophic event.
Accident frequency ==>		1/200 years	

Table 4 TVCS Example – Initial Quantitative Risk Calculation

For the tunnel ventilation example, as the initial accident frequency is once in 200 years, or $5.71\text{E-}07/\text{hr}$, a conditional probability of failure on demand $6\text{E-}03$ must be assigned to contributing event CE_3 (i.e. to the probability that the control system will fail to automatically activate the fans when required).

As such, the safety requirements formulated earlier can be associated with explicit safety targets, as follows:

Requirement 1 Safety Target: *2.28E-09 dangerous failures per hour*

Requirement 2 Safety Target: *6E-03 probability of a dangerous failure on demand*

4 Deriving SILs from Safety Targets

Having determined the relevant safety requirements, and associated quantitative safety targets, the next job is to develop a strategy for achieving these targets.

To the extent that the system is comprised of simple hardware, the field of reliability engineering provides us with techniques to achieve, and to prove achievement of this goal. However, to the extent that the system is comprised of software, it is difficult to say what the target means, and for software of high integrity, beyond the bounds of current technology to demonstrate that it has been met.

In lieu of absolute demonstration of achievement of the quantitative target, one must, to make a case that a system is acceptably safe, fall back on an argument that one has exercised due diligence, applying good practice appropriate to the nature and integrity requirement of the system under consideration.

To provide guidance here, IEC 61508 provides the safety integrity level concept, which associates suites of development and assurance techniques with different quantitative target ranges. The SIL concept is well explained in Redmill (2000).

The SIL concept, as described in IEC 61508 and related standards, has been criticised by Lindsay & McDermid (2000) on the grounds that the link between suites of development and assurance practices, and target failure rate ranges, is not supported by empirical studies. Other criticisms relate to the fact that the overall process is open-loop; that is, one determines the required tolerable hazard rate, infers the corresponding SIL, and then applies the development practices appropriate to that SIL. However, the loop is often not closed, in the sense that no argument is made that taken as a whole, the evidence generated in the course of development is sufficient to establish that the required safety has been achieved.

While these criticisms are noted, and accepted, the fact remains that the SIL concept is prevalent in all industries affected by IEC 61508 or related standards, and must therefore be applied. Also, we note that many of the criticisms of the SIL concept are not so much of the concept itself, as of the way it is misused. Being alert to these pitfalls is a step along the journey to avoiding them.

IEC 61508 recognises that safety-related systems can operate in two modes, and that the unit associated with the safety target will vary because of this. It recognises low-demand operation, where the frequency of demands for operation are no greater than once per year, and no greater than twice the proof-test frequency. For low demand mode operation, the associated target failure measures are expressed as probability of failure to perform the function when demanded. It also recognises high demand or continuous mode, where the frequency of demands for the function is greater than once per year or greater than twice the proof-test frequency. For high demand functions, the associated target failure measures are expressed as dangerous failure rates.

With reference back to our accident sequence model, and to Eqn 1, we can see that this might reflect a possible assumption within IEC 61508 that if systems are operating in high-demand or continuous mode of operation, their failures will constitute initiating events in an accident sequence. Alternatively, if systems are operating in low-demand mode, their failures will constitute contributing events in an accident sequence. Unfortunately, this is not always true.

Referring back to our signalling example, the derivation of a safety integrity level from the quantitative safety target is in this case apparently straightforward. Most would argue that the system, performing interlocking functionality continuously, is operating in high demand mode. As such, it is straightforward to take the quantitative safety target of $2.28\text{E-}09$ failures/hr, note that it lies within the range for SIL 4 functions, and hence infer that the interlocking function must be developed using SIL 4 development and assurance practices.

For the tunnel ventilation example, however, where the target is expressed as a conditional probability, how do we perform the SIL translation?

The function of activating emergency control mode, as required for this particular accident sequence, will not be demanded more frequently than once a year. However, in some tunnel ventilation control systems, the modes used for responding to fire situations are also used to respond to some classes of standard congestion situation. Standard congestion situations happen frequently, as a result of traffic congestion or other more routine disturbances (short term glitches in traction power supply, hazards on the track that need to be cleared, etc.). As such, the actual function may be exercised frequently, although the safety demands related to this accident sequence may be less frequent. Furthermore, the software on the actual TVCS is no doubt performing many functions, some of which would be operating continuously.

The standard also requires that, in order for a function to qualify as a low demand function, it must be shown that the demand is less frequent than twice the proof-test frequency. As such, to show that a function is a low demand mode function, it is not enough to just consider the demand frequency, one must also consider the proof-test frequency.

While the standard is unclear here, in the author's experience most people would argue that the function of

activating emergency mode is nevertheless a high demand function (in this hypothetical example), if the mode is activated more frequently than once per year, albeit for other reasons than indicated in this accident sequence.

This presents a problem, since our safety target is expressed as a probability of failure on demand, and the appropriate target failure measures for high-demand mode functions are expressed as dangerous failure rates.

We can use the formula for the reliability of a system at time t – expressed as $R(t)$ – to relate a dangerous failure rate f to the probability of a dangerous failure, during a demand period t . The relevant formulae are:

$$P(\text{Dangerous failure during period } t) = 1 - R(t)$$

And,

$$1 - R(t) = 1 - e^{-(f \cdot t)},$$

And,

$$1 - e^{-(f \cdot t)} \approx f \cdot t, \text{ for small } f \cdot t.$$

Giving,

$$P(\text{Dangerous failure during period } t) \approx f \cdot t, \text{ for small } f \cdot t.$$

In order to apply this scheme, however, it is necessary to determine an appropriate period t .

For this accident sequence, suppose the function was demanded at time t_i , then the relevant period to use is the time during which a failure of the TVCS would be dangerous. Clearly, this period of time must start before t_i , since if the TVCS happened to be in a failed state when the function was demanded, then the function would be unavailable. As such, the period of time must extend backwards to the last time the TVCS was tested and proved to be working, relative to this functional demand. That is – to the last time a proof test for this function was conducted. Note that the period must also extend forward until such time as we can reasonably expect the situation to have been resolved, since a failure of the TVCS following a successful activation of the mode might, depending on the design of the system, cause the mode to stop with consequent harm to passengers on the train. In summary, the relevant period t is the length of the interval beginning from when the function was last proof tested, through to t_i , and extending up until the time after t_i at which the situation can expect to have been resolved.

It would seem then, that to infer from the target probability of failure on demand, a corresponding dangerous failure rate, and hence to infer a SIL for the TVCS, we need to consider what constitutes an appropriate proof test for this function.

5 Proof-test Intervals and Software

A proof test is a test that proves that some function of a component is working. If the test fails, the component is repaired or replaced. For components subject to random failures, with a failure rate that is constant over time, frequent proof testing can be used to effectively reduce the probability that a component will fail during a defined interval.

The ideal proof test is non-invasive, and does not diminish the reliability of the component. For example, consider a room that is continuously illuminated by a single light bulb. The simple act of opening a door to see if the light is still on is a non-invasive proof test for the light bulb. In most cases however, a proof test involves actually demanding the function, and observing the result to be sure that the component is able to provide the function on demand. Examples are turning on a light to check that it is still working, line integrity tests for telecommunications equipment, physical point-to-point tests for control equipment, etc. For components that are normally in an active state, polling the device and confirming that a response is received is also a kind of proof test.

Generally, the more complex the device, the more difficult it is to perform a perfect proof test. Consider a programmable logic controller, wired to physical input and output points. A test that the controller is powered up and responsive when polled could be considered to constitute a proof test of sorts. Such a test, however, probably would not reveal latent circuit failures on the output cards, and hence would not be considered a sufficient proof test for the function of controlling a particular output point.

Returning to the example of the TVCS, let us suppose the architecture for the system comprises:

1. A LAN connection between the TVCS server, and other systems. This connection delivers the demand from the emergency ventilation mode.
2. A TVCS server, which is commercial grade computing hardware, running software operating on a UNIX platform. The server detects the demand and forwards it to the PLC.
3. An optical fibre connection to one or more PLCs, which perform the low-level plant control.
4. One or more PLCs that take care of local plant interlocks (e.g. dampers should be opened before exhaust fans are activated, etc.), and which are physically wired to the plant.

Note that in the architecture just described there is software in the TVCS server and also in the PLC(s). The question is: *what constitutes an adequate proof test for the function of actioning an emergency mode?* From a hardware perspective, the demand to action the mode is received over a network connection, from another system, so a test that the connection to the other device is working is sufficient. The connection between the server and the PLC would be similarly proved. For the server on which the processing is performed, a test that the server is powered up and “healthy” would be sufficient. For the PLC itself, as just discussed, a valid proof test would need to exercise the circuits involved in the activation of emergency mode.

This is, however, a software-based system, which begs the question of what constitutes a proof test for the software? To answer this, we need to consider the nature of the software, and the way that it fails. Software behaves systematically, in the sense that when a particular logical path is exercised, with a certain set of values assigned to the variables, the result that is produced will either

conform to specification, or not, and hence the software will be considered to have succeeded, or failed. Every time the same path through the software is exercised, with the same value to variable assignment, the result will be identical. Provided the underlying platform is sound, there is no possibility that the software will “stop working”, at some point.

This means that the concept of a proof test, as used in IEC 61508, is essentially meaningless for software. The complexity of even simple safety-related software systems usually renders exhaustive testing infeasible (Butler & Finelli, 1993, and Littlewood, 2000), meaning that for a particular function, it will not be possible to exercise all possible paths, with all possible variable-value assignments. That is, for most practical purposes, it will be impossible to construct a perfect proof test. Returning to our example, a functional demand on the software to action emergency mode, performed as part of a proof test during non-traffic hours on the railway, may exercise the software differently to a demand made “in anger”, when there are high levels of network traffic, other software activity, and when the value of other variables accessed by the program may have changed.

On the other hand, if the software that implements the function was sufficiently simple to allow a “perfect” proof test to be constructed, then it would be enough to do it once – subsequent tests would not add value, from the perspective of software reliability.

In both cases above – where it is not possible, and where it is possible, to construct a perfect proof test, the idea of manipulating proof-test intervals is meaningless for software, since it cannot be argued to affect the probability of failure on demand of the software.

Nevertheless, from a practical point of view, if one is to perform SIL derivation within the IEC 61508 framework, some sort of proof-test interval must be postulated, in order to perform the conversion between the failure on demand target and a dangerous failure rate.

In the author’s experience, the following options have been used:

1. A test that the software is operational, i.e. that it is not currently in a failed state. Within the high integrity systems community, there is some support for the notion that a simple test of this nature is sufficient to constitute a proof test (refer discussion in the High Integrity Systems Engineering (HISE) mailing list, in reference list). Since polling can be performed very frequently, however, this has the consequence of reducing the proof-test interval to milliseconds, which in practical terms means that even an extremely high failure rate can yield a low probability of failure on demand. That is, using this as a proof-test interval can artificially deflate the required safety integrity level. In the case of the TVCS, this would lead to counter-intuitive results, in that it would yield a low failure rate requirement (to the point of requiring no integrity level to be assigned at all), for a function which is clearly relied on to reduce risk.
2. A demand of the relevant function. While simply demanding the function does not constitute a perfect proof test, for the function at large, it is a test of the function with certain input values, and certain environmental variables (i.e. certain state). By performing such tests regularly, over an extended period, and taking care to perform the tests at different times of the day and in different operating modes, it is reasonable to expect that over time a certain level of coverage will be achieved, with respect to the actual operational profile of the function. This strategy is consistent with the point of view that although software fails systematically, for sufficiently complex systems, the demands placed on those systems by the environment in which they are embedded are sufficiently random to be able to be viewed stochastically (Musa, 1998, and Parnas et al., 1990)
3. The time since the software was last reset. There is a compelling argument that for many systems, the value of the program’s variables (i.e. its state) evolves over the time that the software is operational. Accordingly, as the mission time of the software increases, so does the likelihood that the environment will make a demand on the software that has not previously been made. By frequently resetting the software therefore, one increases the likelihood that the demands that are made are in parts of the operational profile that has previously been explored (and therefore is known to work) (Parnas et al. 1990.).
4. Ignore the issue of software proof-test interval, and use an interval that is appropriate for the hardware used to implement the system. This approach appears to ignore the issue altogether, but is actually sympathetic to the general philosophy of IEC 61508, which employs traditional (hardware) reliability engineering methods to derive quantitative targets, and then switches to new ideas (the SIL concept), to achieve them.

6 Discussion

This paper has raised a number of issues that warrant further discussion.

6.1 Primary and Non-primary Functions

The examples in this paper have focused on the use of software to achieve what might be termed the “primary functions” of a system. By primary functions, we mean the functions which are representative of the purpose for which the system was created. Examples include the control of vital outputs by an interlocking system. However it is important to note that software is also used to perform hardware integrity checks and other kinds of diagnostic functions. For example, an interlocking system typically also includes software that continuously checks the electrical integrity of the input and output cards. These functions are usually ancillary to the main purpose of the

system, but are relied on to achieve overall safety. Consideration of the role such functions play is in the author's experience typically delayed to the more detailed system hazard analysis phase, and is not performed quantitatively. That is, one performs quantitative analysis on the primary functions of the system, of the kind described in this paper, to infer that control of vital outputs needs to be assured to (say) SIL 4. Then, during detailed systems analysis, since failure to detect a critical hardware error could contribute to non-achievement of the function, one would also infer that diagnostic software to detect such errors must also be developed to SIL 4. The existence of such diagnostic functions – which are actually proof tests of the underlying hardware – is usually relied on in a detailed reliability analysis for the system hardware.

6.2 Initiating and Contributory Events

The examples in this paper illustrate safety functions whose failures have been modelled as initiating and contributory events. It is interesting to consider whether such events can be classified, and if so, what characteristics ought to be used for the classification scheme.

In the signalling system example, the relevant event is an error of commission (i.e. the system actively does something dangerous), and the derived safety requirement is phrased negatively (i.e. the system *shall not* ...). In contrast, in the TVCS example, the relevant event is an error of omission (i.e. the system fails to do something when required), and the derived safety requirement is phrased positively (i.e. the system *shall* ...). This suggests a distinction based on the nature of the failure mode that characterises the event.

The distinction, however, does not appear to apply universally. Consider the case of a railway traction power SCADA system, where the accident sequence of interest involves inadvertent energisation of high-voltage equipment while work on the line is in progress, and other manual safe-guards have failed. In this case, most would characterise the initiating event on the basis of time, and say that the sequence starts with the (normal) occurrence of maintenance, and contributing events include failure to apply manual safe-guards, and then an act of commission by the SCADA system, during the maintenance period, to energise equipment spuriously.

Also, judgement must be applied in determining the initiating event for any accident sequence. Indeed, a major theme of the criticisms by Leveson (2002) is that accident sequences do not trace far enough backward in the search for causes, and rarely consider organisational causes (e.g. an accident sequence may include operator error as the initiating event, but most analysts would not list the management decisions that led to operators being rostered on for excessively long shifts as an initiating event). This dilemma is also evident in the signalling example in this paper. Another analyst might, for example, list the inappropriate route request submitted by a train control system to be the initiating event, and characterise failure of the interlocking system to block the request as a contributing event (interestingly, it is an error of omission).

This is an area for further research, and would be a necessary pre-cursor to any quantification scheme based on accident sequence position.

6.3 Accident Sequence Inversion

Contrary to the idea explored in the previous subsection, that hazardous events can be absolutely characterised, is the observation that it is usually possible to invert accident sequences.

As shown in the paper, problems of quantification usually arise with contributing events, where it is necessary to convert failure on demand targets to failure rate targets. If the accident sequence can be inverted so that the contributing event is instead re-cast as an initiating event, an alternative view of the system can be obtained. For example, returning to the tunnel ventilation example, imagine instead an accident sequence where the initiating event was failure of the TVCS in a mode that made subsequent activation of emergency mode impossible, the contributing events are then train stoppage and fire, during the period in which the TVCS failure remains undetected.

This accident sequence inversion technique can provide a useful way of “sanity checking” results that depend on a failure on demand to failure rate conversion. It is interesting to note, however, that in the specific example used in this paper, the problem does not go away. To quantify the likelihood of train stoppage and tunnel fire, one must know the maximum likely period for the TVCS failure to remain undetected, which is just the proof-test interval anyway.

7 Related Work

So far as the author is aware, the issue of what constitutes a valid proof test for software does not appear to have been addressed in the literature. Within the safety-critical systems community, there has been some discussion on the High Integrity Systems Engineering mailing list (see reference list), about proof testing in general. However the thrust of that discussion seemed to be about determining what constituted a valid proof test, for the purposes of satisfying both limbs of the definition for “low-demand function” (i.e. the requirement that the demand be less often than once a year, and no more frequent than twice the proof-test interval). The discussion did not seem to confront the relevance of proof tests to the task of transforming probability of failure on demand statistics to dangerous failure rate statistics. Nevertheless, the discussion in that thread is anecdotal confirmation that there is a broad range of opinion on the topic, and little consensus!

The accident sequence model and its use in the derivation of safety integrity targets, and SILs, is similar to the method described in a paper by Lindsay, McDermid and Tombs (2000).

8 Conclusions

This paper has considered what constitutes a valid proof-test interval for software. The following conclusions emerge:

1. Currently, the IEC 61508 framework requires a scheme for converting between targets expressed as probability of failure on demand, and those expressed as a dangerous failure rate, in order to derive SILs for high-demand mode systems that provide response-type functionality. Also, EN50129 has completely eliminated the concept of a low-demand function, forcing one to apply such a scheme to any system providing response-type functionality.
2. Such a scheme demands that the concept of a proof test be addressed.
3. Even if the system is operating in low-demand mode, one must still consider the proof-test frequency, and hence what constitutes a proof test, in order to satisfy the second limb of the definition of “low-demand mode”.
4. For components with random failure modes, proof testing provides an opportunity to discover latent failures and remove them, before a safety-related demand is made. In quantitative terms, this reduces the probability that the component will fail when a safety-related demand is made.
5. However, for components in which systematic failure modes dominate, such as software, it is usually infeasible to design a perfect proof test. Paradoxically, if such a test can be designed, then it is enough to run it once! Reducing the proof-test interval will not affect software reliability.
6. Despite the above difficulties with the concept, IEC 61508 forces practitioners to confront the issue, if quantitative safety targets are to be derived. The paper reviewed a number of approaches to arriving at a value to use for such a calculation. We do not recommend any specific approach; we simply note the pros and cons associated with each of them. It is suggested that the decision about which approach to use will need to have regard to the particular application at hand.
7. This is an area where more guidance, from the standards bodies, is badly needed!

Ancillary to the main theme of the paper, the following observations are made:

1. In practical terms, the specification of the requirements for a system is usually informed by ideas about how those requirements will be implemented. However, the systems engineering approach suggests that these issues should, so far as possible, be kept separate. That is, one should determine what a system must do, without being concerned with how it shall achieve that. This leaves maximum freedom to designers. This paper has shown that this will not be possible, however, if it is desired to associate with system requirements a safety target that is expressed as a safety integrity level. This is because some

knowledge of system implementation is necessary to determine what constitutes a valid proof test, and knowing what constitutes a valid proof test is essential to proposing a reasonable proof-test frequency.

2. There is a move to eliminate the low-demand/high-demand distinction, and force all safety targets to be translated to dangerous failure rates. This will mean the issues considered in this paper will come to light more frequently. An alternative approach that could be considered by standards authors is to continue the current scheme of having two tables, according to the units in which the safety target is expressed. However, the current basis for distinction (i.e. high vs. low demand) could be replaced with a classification scheme based on where the system appears in the accident sequence (i.e. whether its failures are initiating events or contributing events).

9 Acknowledgements

The ideas in this paper have evolved based on my experiences and observations over the last five years. I am particularly indebted to Neil Robinson for recent stimulating discussions about the issues, and to the anonymous reviewers for many helpful comments. Errors are of course my own.

10 References

- IEC 61508, Functional Safety of electrical/electronic/programmable electronic safety-related systems, Parts 1—7, 1998—2000, International Electro-technical Commission.
- IEC 61511, Function Safety – Safety instrumented systems for the process industry sector, Parts 1—3, 2004, International Electro-technical Commission.
- EN50126, Railway applications – The specification & demonstration of reliability, availability, maintainability and safety (RAMS), 1999, CENELEC.
- EN50128, Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems, 2001, CENELEC.
- EN50129, Railway applications – Communications, signalling and processing systems – Safety related electronic systems for signalling, 2003, CENELEC.
- Leveson, N.G., *Safeware: System Safety and Computers*, Addison Wesley, 1995.
- Leveson, N.G., *System Safety Engineering: Back to the Future*, 2002. <http://sunnyday.mit.edu/book2.pdf>
- Redmill, F. Safety Integrity Levels – Theory and Problems, in *Lessons in System Safety, Proceedings of the Eighth Safety-Critical Systems Symposium*, Springer Verlag, 2000.

- Redmill, F. *Understanding the Use, Misuse and Abuse of SILs*,
http://www.csr.ncl.ac.uk/FELIX_Web/3A.SILs.pdf
- Lindsay, P.A., and McDermid J.A., A systematic approach to software safety integrity levels, Software Verification Research Centre Technical Report 00-17, May 2000.
<http://svrc.it.uq.edu.au>.
- Lindsay, P.A., McDermid, J.A., and Tombs, D. A systematic approach to software safety integrity levels, In *Proceedings of the 19th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2000)*, Springer Verlag, 2000.
- Safety Critical Mailing List Archive 2005:
<http://www.cs.york.ac.uk/hise/safety-critical-archive/2005/0122.html>
- D. L. Parnas, J. v. Schouwen, and S. P. Kwan, Evaluation of safety critical software, *Communications of the ACM*, 33(6):636-48, 1990.
- Musa, J. *Software Reliability Engineering*, McGraw-Hill, 1998.
- Butler, R.W., and Finelli, G.B., The infeasibility of quantifying the reliability of life-critical real-time software, *IEEE Transactions on Software Engineering*, Vol 19, No. 1, Jan. 1993, pp3—12.
- Littlewood, B. The problems of assessing software reliability ... when you really need to rely on it, Centre for Software Reliability, City University, London, UK. 2000.
http://www.csr.city.ac.uk/people/bev.littlewood/bl_public_papers/SCSS_2000/SCSS_2000.pdf

Dynamic Design and Evaluation of Software Architecture in Critical Systems Development

Klaus Marius Hansen

Lisa Wells

Computer Science Department

University of Aarhus

DK-8200 Aarhus N

Email: {klaus.m.hansen,wells}@daimi.au.dk

Abstract

The software architecture of a computing system is an abstracted structure of the system in terms of elements and relationships. Such structures may be viewed from a number of viewpoints including static/module, dynamic/execution, and deployment viewpoints. Software architecture fundamentally influences systems from all of these viewpoints and designing and implementing proper software architectures is thus critical in many problem domain areas, including the ones that pertain to safety-critical systems.

With respect to safety-critical systems, a particular problem with focusing on software architecture is that there may be a large abstraction gap between an architectural description and an executing system or a formal model thereof thus potentially leading to inconsistencies between models and implementation. Addressing this problem, this paper presents tools and techniques for specifying executable software architectures and for validating these with formal models such as statecharts and Petri nets.

1 Introduction

Safety-critical systems are systems that can cause undesired loss or damage to life, property, or the environment, and safety-critical software is any software that can contribute to such loss or damage [20]. Since safety-critical systems have the potential to cause extensive damage, there are many standards and guidelines describing processes, techniques, and methods for developing such systems. For example, the IEC 61508 [14] is a standard for achieving functional safety of programmable electronic safety-related systems, and the Australian Defence standard 5679 [9] is concerned with the procurement of computer-based safety critical systems. Such standards contain recommendations regarding which techniques and measures should be used when developing software.

One of the techniques that these and other standards recommend or even require is the use of semi-formal or formal methods through various development phases for improving the quality of the safety-

critical software. The use of formal methods and supporting tools "provide increased repeatability of analysis, increased soundness and extra assurance" [9]. The IEC 61508 standard recommends that (semi-)formal methods should be used at various development states, including software safety requirement specification, software architecture design, detailed software design and development, and software safety validation. The recommended methods include (semi-) formal models for representing both static and dynamic characteristics of the software. Here we are only interested in models for representing dynamic behaviour of systems. Such models can be used for either specifying desired behaviour of software and/or for validating and verifying that modelled software behaves as desired.

While the standards advocate the use of (semi-) formal models, they do not necessarily make any recommendations about how to ensure consistency between models of software behaviour and the corresponding executable software. It is clearly a good idea to model software behaviour, however, the usefulness of such models will be compromised if it is not possible to ensure some consistency between the model of the behaviour, and the behaviour of the executable software. This paper presents tools and techniques for validating the behaviour of executable software against models of the behaviour of the software, and thereby for reducing the gap between the software and the model.

1.1 Modelling Software Behaviour

Models of software behaviour can be used for many different purposes, such as for specifying software requirements, for designing software, and for analysing the behaviour of software. Since the majority of accidents in which software was involved can be traced to requirements flaws [20], it is of particular importance to develop complete and unambiguous requirement specifications for safety-critical software. Several standards recommend that requirements be specified as (semi-)formal models, and there is even rigorous language and tool support for checking completeness and consistency of software specifications [11]. The behaviour of software can be modelled both by static models, such as decision tables and Unified Modeling Language (UML) sequence diagrams and by dynamic models with executable behaviour, such as finite and timed automata, statecharts, and Petri nets. One of the advantages of using dynamic models is that it is possible to investigate and, in some cases, even verify the behaviour of the model in an appropriate tool.

Dynamic models that represent states of a system and transitions from one state to another can represent either discrete or continuous changes between states. When modelling the behaviour of (safety-

The research presented in this paper has been supported by ISIS Katrinebjerg Competency Centre, <http://www.isis.alexandra.dk>.

critical) software, it is rarely interesting to have an accurate model of continuous state changes, and in most cases it is sufficient to consider a set of discrete state changes. For example, when modelling software that controls the speed of a conveyor belt, it would not be necessary to model all possible speeds of the conveyor belt, but it would be sufficient to consider a number of different discrete classes of speeds, such as stopped, within range, and above acceptable range.

In this paper we consider only *discrete-state models* which are state-based models with discrete transitions between states. Transitions between states will also be called events. A more formal definition of the kind of models that we are interested in will be provided in Sect. 3.2. As always, when using models it is important to find an appropriate level of abstraction for the models. If the models are too detailed, then it may be too time-consuming to develop them, and it may be difficult, if not impossible, to do reasonable analysis of the behaviour of the model. Discrete-state models are well-suited for specifying fairly high-level requirements, and for analysing the behaviour of relatively small systems.

A variety of tools provide support for creating and analysing different kinds of discrete-state models of software behaviour. For example, SPIN [13] and UPPAAL [19] support model checking of finite and timed automata respectively, visualSTATE [25] and STATEMATE [10] support analysis of statecharts, and CPN Tools [7] supports analysis of a kind of high-level Petri nets which will be introduced in Sect. 4. With some of these tools, it is possible to generate executable code from the models, in which case, it is possible to ensure that there is consistency between the model and the code (assuming that the code is regenerated or updated if the model is modified). However, if code is not or cannot be generated from models, then there is likely to be a large gap between the models of software behaviour and the executable code that is modelled. And in particular, even though code may be generated, it is not certain that it corresponds to a required or desired software architecture. This lets us to consider the concept of *software architecture*.

1.2 Software Architecture

Software architecture is concerned with abstracted structures of software systems. A generally accepted definition of the term ‘software architecture’ is

Definition 1 (Software Architecture) *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [4]*

The definition implies a number of characteristics of software architecture. First, a system has many structures/views of interest (e.g., module structure, dynamic structure at runtime in terms of processes and communication, and deployment structure in terms of processors and components deployed) [18]. Secondly, software architecture is abstract in the sense that it is only concerned with externally visible properties of elements and relations and thus not concerned with the inner structure of components. Thirdly, all systems have a software architecture whether intended or not.

All of these characteristics are relevant in relation to software safety. Software architecture highly influences various system quality attributes such as performance, modifiability, and testability because these are influenced by structures in various views [4]. A

consequence of the second characteristic is that software architecture descriptions may be more manageable than the actual system (or a less abstract description thereof) making the descriptions amenable to, e.g., analyses and communication. And a consequence of the third characteristics (in combination with the above) is that software architecture is well worth to be concerned with in safety-critical system development.

A large number of techniques for software architecture requirements analysis such as Quality Attribute Workshops [1] and Global Analysis [12]; techniques for software architecture design such as Attribute-Driven Design [5] and architecture pattern-base design [6]; and techniques for software architecture evaluation such as the Architecture Tradeoff Analysis Method and Architecture Level Prediction of Software Maintenance [8] have been developed and tested. One characteristic of these are that they are almost all specification-based in that they use and produce *descriptions* of software architectures rather than software architectures of actual systems. Some problematic consequences of basing software architecture work solely on such descriptions can be that the architecture-as-built differs from the architecture-as-designed, that quality attributes are not properly addressed, or that software architects tend to design conservatively even if the conservative choice may not be appropriate.

As a way to mitigate some of these problems, and as a supplement to existing well-documented techniques related to software architecture, we have previously introduced the concept of *architectural prototyping* [2, 3]:

Definition 2 (Architectural Prototype) *An architectural prototype consists of a set of executables created to investigate architectural qualities related to concerns raised by stakeholders of a system under development. Architectural prototyping is the process of designing, building, and evaluating architectural prototypes [2]*

Architectural prototypes are characterized by having no functionality per se and thus often being cheap to implement. Often architectural prototypes experiment with and evaluate infrastructure and middleware, e.g., to decide whether a push or a pull message passing architecture is most suitable for an embedded control system [2]. Section 2.4 presents an architectural prototype constructed in a safety-critical system development context.

In this paper we claim that architectural prototypes are useful in safety-critical software development in that the technique promises a cost-effective way to implement various architectural alternatives. Further, we provide a way of validating such executable software architectures. In doing this, we are in line with the views of [21]: what matters more than how or by which principles it was developed is that the designed software architecture is safe.

1.3 Software Architectures and Discrete-State Models

Given the above discussion, there are a number of issues in combining the use of software architecture and discrete-state model in the development of (critical) software systems.

Most fundamental is that software architecture is concerned with structures (of systems) whereas discrete-state models are concerned with behaviour. Further, discrete-state models typically provide one, behavioural view of a system whereas software architecture provides several as discussed in Section 1.2.

An example of why this may be an issue is that a deployment decision (such as about the type of network used in a concrete distributed system) may impact behavioural characteristics such as performance.

This also means that discrete-state models are mostly concerned with runtime system quality attributes (e.g., logical correctness, reliability, performance, or scalability) whereas software architecture is also concerned with development time system qualities (e.g., modifiability, testability, or interoperability).

Finally, discrete-state models and software architectures may also often be orthogonal abstractions of a system. In our case study, presented in Section 2, discrete-state models were used to model requirements of the system where a software architecture is used to represent the system per se.

These problems make, e.g., traceability between software architectures and discrete-state models and reasoning about whether software architectures fulfill requirements modeled by discrete-state models hard. Section 3 introduces our approach to handling parts of these problems.

1.4 Contributions

The main contribution of this paper is the introduction of the Heimdall¹ tool. The tool enables the validation of sequences of program execution events against a discrete-state model. We present a real-life case study in which Heimdall is applied by using aspect-oriented instrumentation to an architectural prototype of a frequency converter for safety-critical applications for which program execution events are then mapped to a formal model of requirements described by a Coloured Petri Net [15].

The rest of the paper is structured as follows. Section 2 describes the case study which emphasised the need for tools like the Heimdall tool. Section 3 describes the architecture and functionality of the Heimdall tool, and it also illustrates the current implementation of the tool. Section 5 discusses ideas for future work and concludes the paper.

2 Frequency Converter Case

Several of the problems and issues that were discussed above were encountered in a collaborative research project between Danfoss Drives², Systematic Software Engineering³, and the Computer Science Department, University of Aarhus⁴. Danfoss Drives produces frequency converters which are used to control the speed of motors, e.g. for elevators, cranes, and conveyor belts. A new generation of frequency converters is being developed in accordance with IEC 61508. One part of the project investigated different (semi-)formal methods for specifying software safety requirements. Another part of the project focused on the design of the software architecture for the frequency converter. In this project we experienced the problem of a large gap between the models specifying the software safety requirements and the executable prototype of the software architecture. This section will briefly present the case study which is described in more detail in [26].

¹Heimdall is the watchman of the Gods in Norse mythology. Using his excellent hearing and vision he watches the rainbow, Bifrost, that leads to Asgard, the home of the gods, sounding his alarming horn when danger approaches

²<http://drives.danfoss.com>

³<http://www.systematic.dk>

⁴<http://www.daimi.au.dk>

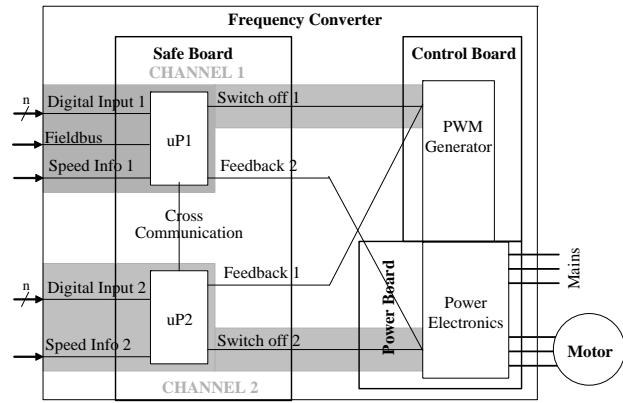


Figure 1: Hardware structure of a frequency converter with safety functions.

2.1 Hardware and Software

In the new generation of frequency converters, safety-critical software runs on two microprocessors. The hardware structure of the frequency converter is shown in Figure 1. The two blocks *PWM Generator* and *Power Electronics* control the speed of the attached motor, and they make up the normal, “non safety-related” part of a frequency converter.

The safety functionality is achieved by an additional subsystem on the *Safe Board* composed of *Channels 1* and *2*, each containing a microprocessor (*uP*), a *Switch-off* path, and three *Digital Inputs*. The two microprocessors can, independently from each other, activate its own switch-off path to stop the motor. The two *Channels* cross-monitor each other through *Feedbacks 1* and *2* and through the *Cross Communication* connection.

A number of so-called *designated safety functions* (DSF, or safety function) are implemented in software that runs on the two microprocessors on the *Safe Board*. The simplest safety function is a so-called ‘uncontrolled stop’ which immediately stops power supply to the motor. Another safety function is a ‘controlled stop’ or ‘safe delay’, where the stopping of the power supply to the motor is delayed, allowing the non-safety-related part of the frequency converter to ramp the motor down in a controlled way. A more complex example is the ‘safe speed’ where an uncontrolled stop is made if the motor speed exceeds a set limit. A frequency converter is configurable, and users can determine which safety function is associated with each of the $n=3$ digital inputs. A specific safety function is activated upon reception of signals at the appropriate digital input at each of the *Channels*.

All diagnostic functionality with respect to cross monitoring and self monitoring of the *Channels* is implemented in software. On detection of a dangerous failure, an appropriate fault reaction is initiated, and the motor is stopped.

2.2 Specifying Safety Requirements

The software that runs on the two microprocessors on the *Safe Board* is safety-critical since it can contribute to loss or damage to the environment of the frequency converter through its effect on the speed and control of the attached motor. System-level safety requirements were already defined at the outset of the project. These requirements addressed issues such as, when output to the motor should be enabled, what should happen when an error occurs (either in hardware or software), how requests for safety functions

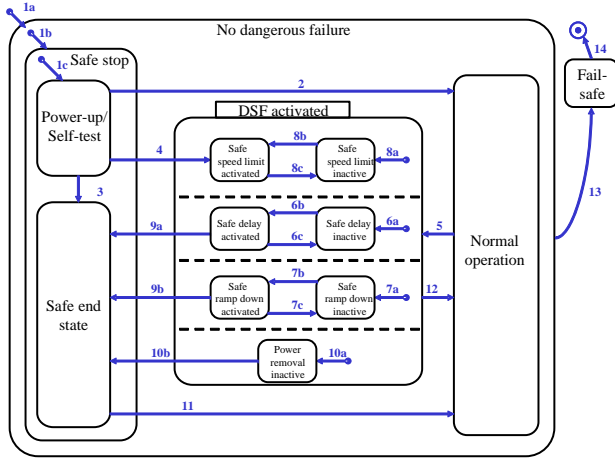


Figure 2: Informal statechart specification of system-level safety requirements.

should be made and handled, and what should happen after a safety function completes.

As mentioned previously, one of the recommendations of standard IEC 61508 is that semi-formal methods should be used to specify safety requirements. In order to comply with this recommendation, Danfoss developed an informal statechart model (shown in Figure 2) that was included in the initial product proposal that was approved by the certification authorities. The model is informal in that it was drawn in a generic drawing tool, and the states, transitions, and event triggers are described separately in simple, natural-language texts. It is not important to understand the details of the behaviour specified by the statechart, but it will be briefly explained.

The statechart specifies that the frequency converter must always be in one of three top-level states, namely *No dangerous failure*, *Fail-safe* or the *Final state* (denoted by a dot in a circle in the upper right-hand corner of the figure). If any kind of error is detected, then the frequency converter must enter *Fail-safe* state, and the power supply to the motor must be stopped. The only way to leave *Fail-safe* state is to turn the frequency converter off (Transition 14), and thereby enter *Final state*. If no errors are detected, then the frequency converter must be in *No dangerous failure* state, and more specifically, in one of its three composed states: *Normal operation*, *DSF activated* or *Safe stop*. In *Safe stop* state, output to the motor is always disabled.

One of the goals of the project was to specify software safety requirements based on the informal statechart of the system safety requirements. The software safety requirements were a refinement of the system safety requirements. Again, the IEC 61508 standard highly recommended that semi-formal methods should be used to define software safety requirements.

2.3 CPN Model of Requirements

A very detailed model of software safety requirements was developed in the formal modelling language Coloured Petri Nets (CPN or CP-nets) [15, 17]. This section will provide a brief overview of the CPN model of the frequency converter, and the formal definition of CPN will be introduced in Sect. 4. All of the requirements that were specified in the statechart model from Figure 2 are included in the CPN model. Those requirements have been specified more formally, and the specification is much more detailed. In addition, the CPN model specifies requirements that are not addressed in the statechart model, such

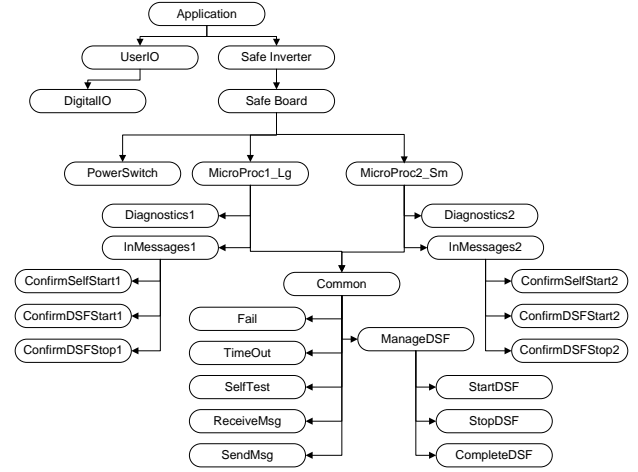


Figure 3: Module hierarchy of the CPN model.

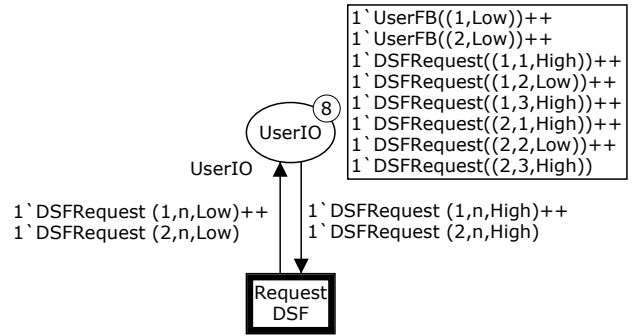


Figure 4: DigitalIO module from the CPN model.

as diagnostics and synchronisation of the state of the software on the two microprocessors.

Figure 3 provides an overview of the CPN model which was created in CPN Tools. Each node in Figure 3 represents a module in the model, and an arc from one node to another indicates that the source node contains an abstract representation of some behaviour that is specified in more detail in the module of the destination node.

The *Application* module (at the top of Figure 3) is the most abstract representation of the frequency converter and its environment. This module has two submodule, namely *UserIO* and *Safe Inverter*, modelling the means for user input/output, i.e. the digital inputs (module *DigitalIO*) shown in Figure 1, and the frequency converter itself, respectively. The software for the two microprocessors is modelled by the modules *MicroProc1.Lg* and *MicroProc2.Sm*. Both of these modules share some common functionality as specified by the module *Common* and its submodules. The two microprocessors send different kinds of messages and have different diagnostic algorithms, which is why there are separate modules for modelling these characteristics.

Figure 4 shows a simplified version of the *DigitalIO* module of the model. Requests for activating safety functions are modelled in this module. The behaviour of the module will be discussed in detail in Sect. 4.1.

Simulations of the model were run for three main purposes: for debugging the model, for analysing the behaviour of the model, and for discussing the software requirement specification with the project team. Even though an exhaustive investigation of the behaviour of the model was not performed, a number of important problems were identified through the

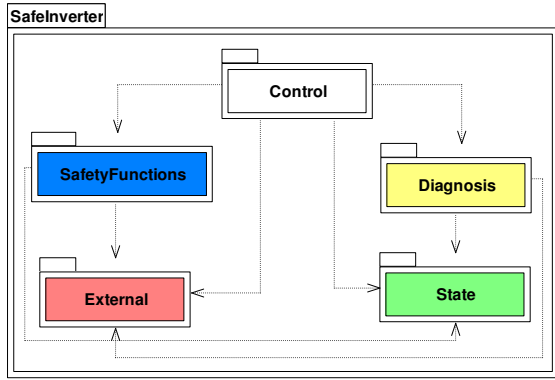


Figure 5: Package diagram for the software architecture.

construction and simulation of the model. Examples of these problems were: a simple diagnostic algorithm could lead to deadlock, and outdated messages in message queues could lead to hazards, such as enabling power supply to the motor after an error had been detected in one of the microprocessors.

2.4 Executable Architecture Prototype

Another goal of this project was to investigate and develop techniques for ensuring that safety-critical software fulfills the corresponding software safety requirements. In other words, we were interested in closing the gap between a semi-formal requirement specification and a software implementation. We focused on techniques for specifying and validating a software architecture (rather than the final software) for the frequency converter. A software architecture was developed and documented using a technique similar to Kruchten's 4+1 technique [18] in which an architecture is described in different views.

The architecture was defined largely by UML diagrams, including class, package, deployment, and sequence diagrams. Figure 5 shows the package diagram for the software architecture. The **Control** package contains classes for ensuring strict scheduling requirements for the frequency converter, including regular checks for requests on digital inputs, diagnostics, and checking microprocessor state consistency. The **Safety Functions** package contains classes for the safety functions. The classes in the **Diagnosis** package initiate, coordinate, and perform diagnostics. The **State** package is used by software on the two microprocessors to regularly communicate and compare their internal states. The **External** package contains classes for reading and setting digital input/output values.

An executable architecture prototype was implemented as skeleton classes in Java. Figure 6 shows an abstract class from the **Safety Functions** package for the architecture prototype. Classes for each of the different safety functions are defined as specialisations of this abstract class. A number of important use scenarios were described as sequence diagrams, such as initialisation during power-up and requesting safety functions. These use scenarios were implemented as simple Java programs that exercised the architecture prototype by emulating external events of the frequency converter, e.g. pressing the power button or requesting a safety function by activating a digital input, by calling appropriate methods in the executable architecture prototype. Given this architecture prototype, Danfoss was interested in developing techniques for ensuring that the architecture fulfilled the software safety requirements, including those specified by the CPN model. An early prototype of the Heimdall tool

```
public abstract class SafetyFunction {
    State state;
    boolean isrequested = false;

    public SafetyFunction (State state) {
        this.state = state;
        selfCheck();
    }
    public abstract void activate();
    public abstract void selfCheck();
    public abstract boolean isRequested();
}
```

Figure 6: Java skeleton class from executable software architecture.

was developed during this project. We introduce the Heimdall tool next.

3 The Heimdall Tool

Informally, the intended function of the Heimdall tool is to map a sequence of well-defined program execution events to a sequence of well-defined model events of a discrete-state model (Figure 7).

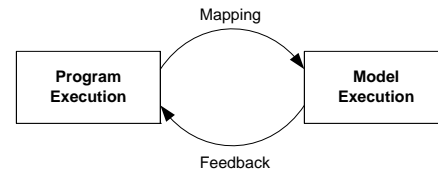


Figure 7: Conceptual overview of the Heimdall tool

The mapping is introduced more formally in Section 3.2 and concrete examples of the specification of mappings is given in Section 4.3. The mapping should be done in such a way that for an implementation that violates the model, the execution should at some point lead to a corresponding sequence of model events that are illegal with respect to requirements and feedback should be given. Conversely, the execution of a correct implementation should not lead to violations in the corresponding sequence of model events.

In the following sections we first present an overview of the architecture of Heimdall followed by a more precise introduction of the mapping of execution events to model events. Next, we present our concrete instantiation of the architecture to be used with Coloured Petri Nets and show how the Heimdall tool has been applied to architectural prototypes in the frequency converter case.

3.1 Heimdall Software Architecture

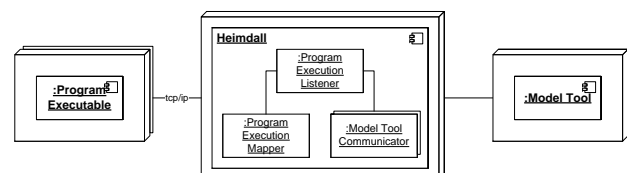


Figure 8: UML deployment overview of software architecture of the Heimdall tool

An overview of the architecture of the Heimdall tool is shown in Figure 8. A *Model Tool* is a tool which can execute and analyse a discrete-state model of the behaviour of an executable program. A set of

Program Executables are instrumented to send execution events to the *Program Execution Listener* of the Heimdall tool. Instrumentation may be done, e.g., by using a debugger, by instrumenting source code with tracing functions, or by using an aspect-oriented approach (such as AspectC++ [24] for C/C++, AspectJ [16] for Java, or (eventually) AspectAda [23] for Ada95). The instrumentation sends information about relevant program execution events to the Heimdall tool instance using a TCP/IP-based protocol.

The Heimdall tool instantiates a *Program Execution Mapper* based on a description of the mapping of execution events to model events. This mapping is described in an XML format of which Figure 10 gives an example. Whenever the Program Execution Listener receives an event from a program execution, it consults the Program Execution Mapper. The Program Execution Mapper maintains traces of program execution events and returns corresponding model events as appropriate (cf. Definition 8 in Section 3.2). Given a match, the *Model Tool Communicator* is used to communicate with a Model Tool in order to examine whether the mapped model events are legal in the model that the program execution is validated against.

The Model Tool Communicators are tool-specific. The requirements on Model Tools that are to be used with Heimdall is provisions for tool integration, e.g., through plug-in capabilities, trace replay, or using a tool-specific protocol. Our current status is that CPN Tools can interact with the Heimdall tool (see Section 4). Also traces of execution events need not be replayed immediately, but may be saved and (re)executed later, meaning that different mapping could be tested against the same program execution trace. Correspondingly, mapped model elements could also be saved for later transfer.

3.2 Mapping from Execution Events to Model Events

The intent in Heimdall is to validate that a sequence of execution events corresponds to a valid sequence of model events. This is achieved, in part, by mapping sequences of *join points* [16] in a *program execution* to a sequence of valid events in a discrete-state model. A join point is a well-defined point in the execution of a program. We are primarily interested in join points corresponding to method/procedure calls, setting field/data values, and getting field/data values.

In Sect. 1.1 we informally described discrete-state models, now we will provide a formal definition of the models in which we are interested. A discrete-state model is a model that is equivalent to a *labelled transition system*:

Definition 3 (Labelled Transition System) A labelled transition system is a tuple $LTS=(S,i,\Lambda,T)$ where S is a set of states, $i \in S$ is the initial state, Λ is a set of labels, and $T \subseteq (S \times \Lambda \times S)$ is the set of labelled transitions.

Note that both the set of states and the set of labels may be uncountable. An LTS is said to be finite if its sets of states and labels are finite. For a labelled transition system with states s_1, s_2 , and label l where $(s_1, l, s_2) \in T$, we will write $s_1 \xrightarrow{l} s_2$ and further, for a set of labels, Λ , Λ^* denotes the set of all sequences of labels from Λ . An element of Λ^* is legal or valid if it is a *trace*:

Definition 4 (Trace) Given a labelled transition system $LTS=(S,i,\Lambda,T)$, a sequence of labels $l_1 l_2 \dots l_n \in \Lambda^*$ is a trace of LTS if $\exists s_1, s_2, \dots, s_n \in S$ so that $i \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \dots \xrightarrow{l_n} s_n$.

We also consider the set of possible *program executions* of a program as a labelled transition system where the states are program states of interest (which may be discerned by heap contents, stack contents etc.) during execution and where the labels are join point executions and related state, i.e., events of interest in the program execution:

Definition 5 (Program Execution System) A program execution system is a labelled transition system $P = (S_P, i_P, \Lambda_P, T_P)$ which is a representation of all of the possible executions of a program in which execution states are abstracted into S_P and where call, set, and get join point executions are abstracted into Λ_P .

Note that the definition of a program execution system is somewhat imprecise in that the definition of the set of states and set of labels is left to the discretion of those who are interested in validating an executable program against a discrete-state model of the behaviour of the software. Thus a reasonable set of “interesting” states and “interesting” join point labels that will be used during the validation process will have to be defined. Section 4 will discuss the states and join point labels that were used when validating the executable architecture prototype of the frequency converter against the CPN model of the software safety requirements.

Recall that the purpose of the Heimdall tool is to provide support for validating an executable program against a model of the behaviour of the program. We have just defined discrete-state models and program execution systems in terms of labelled transition systems. LTSs are quite general in that they allow for non-deterministic behaviour and infinitely many states and labels. All that we need now is a way to show that two labelled transition systems are (more or less) equivalent. A large body of research is concerned with this issue, and bisimulation and weak bisimulation can be used to show equivalences between two LTS. However, the problem with these techniques is that they are difficult, if not impossible, to use for large LTSs with (infinitely) many states and labels.

Many systems, and in particular safety-critical systems, can be represented by finite labelled transition systems, which are somewhat more practical to deal with. Furthermore, the behaviour of safety-critical systems is generally deterministic, which means that the set of transitions for the LTS for the system would be deterministic, in other words, if $s_1 \xrightarrow{l} s_2$ and $s_1 \xrightarrow{l} s_3$ then $s_2 = s_3$. Finite, deterministic labelled transition systems are somewhat easier to deal with, however it is rarely possible to construct and analyse an LTS for complicated, industrial-sized systems. So it is still necessary to develop techniques that can be used to check and validate the behaviour of software for non-trivial systems.

Given the definitions above, our primary interest is now to define mappings between *program executions* and *model executions* that will allow us to validate the behaviour of an executing program against the behaviour of a discrete-state model. Program and model executions are defined as traces of an LTS:

Definition 6 (Program and Model Executions) Given a program execution system, $P = (S_P, i_P, \Lambda_P, T_P)$, a program execution for this system is a trace $p \in \Lambda_P^*$.

Given a discrete-state model, $M = (S_M, i_M, \Lambda_M, T_M)$, a model execution is a trace $m \in \Lambda_M^*$.

In this context, a program execution is considered to be a sequence of execution join points that form a

trace. Similarly, a model execution is a sequence of legal model events. For such program executions, we are interested in mappings of these to corresponding events in the discrete-state model that is an abstract representation of the behaviour of the program execution system of the program execution. More precisely, we define an *execution mapping* as:

Definition 7 (Execution Mapping) *Given a program execution system $P = (S_P, i_P, \Lambda_P, T_P)$ and a discrete-state model expressed as an LTS $M = (S_M, i_M, \Lambda_M, T_M)$, an execution mapping for P and M is a set $E \subseteq (\Lambda_P^* \times \Lambda_M^*)$.*

In other words, an element e in an execution mapping specifies how sequences of program join points map to sequences of model events. An example of a mapping element would be $(l_{p_1}l_{p_2}l_{p_3}, l_{m_1}l_{m_2})$ meaning that $l_{p_1}l_{p_2}l_{p_3} \in \Lambda_P^*$ maps to $l_{m_1}l_{m_2} \in \Lambda_M^*$. The goals of the validation process will help to determine how detailed the execution mapping should be.

Based on program execution systems and mapping definitions we are now able to define when a program execution may be considered correct:

Definition 8 (Correctness) *A program execution $p = l_{p_1} \dots l_{p_k}$ of a program execution system $P = (S_P, i_P, \Lambda_P, T_P)$ is correct with respect to a discrete-state model $M = (S_M, i_M, \Lambda_M, T_M)$ and an execution mapping E if*

1. $\exists e = (l_{p_1} \dots l_{p_u}, l_{m_1} \dots l_{m_v}) \in E, s_{p_1}, \dots, s_{p_u} \in S_P, s_{m_1}, \dots, s_{m_v} \in S_M : i_P \xrightarrow{l_{p_1}} s_{p_1} \xrightarrow{l_{p_2}} \dots \xrightarrow{l_{p_u}} s_{p_u} \wedge i_M \xrightarrow{l_{m_1}} s_{m_1} \xrightarrow{l_{m_2}} \dots \xrightarrow{l_{m_v}} s_{m_v}$ where $l_{p_1} \dots l_{p_u}$ is a prefix of p and
2. $p' = l_{p_{u+1}} \dots l_{p_k}$ of $P' = (S_P, s_{p_u}, \Lambda_P, T_P)$ is correct with respect to $M' = (S_M, s_{m_v}, \Lambda_M, T_M)$ and E where p' is the remainder of p after the prefix $l_{p_1} \dots l_{p_u}$ has been removed.

Note that P' and M' are essentially the same as P and M — the only difference is the initial states.

In some cases a discrete-state model may contain events that do not correspond to any “interesting” execution events. For example, the model may specify behaviour that is more detailed than what is currently implemented in the software, or there may be model events that are used to initialise parts of the model at the beginning of an execution. Since the definition of an execution mapping allows an empty sequence of join points to be mapped to a non-empty sequence of model events, it is still possible for unmapped model events to occur when checking correctness of a program execution.

Even though a set of program executions are correct with respect to a mapping, they are not necessarily “good” in the sense that they cover all states of the discrete-state model. Ideally, we also want *completeness* for this set of program executions:

Definition 9 (Completeness) *A set of correct program executions are complete with respect to an execution mapping and a discrete-state model if the set of all states of the model execution mapped to is the complete set of states of the discrete-state model.*

Ideally we would like to do an exhaustive verification of program executions against discrete-system models, but this is rarely possible in practice which is why there is a need for tools like Heimdall. In a safety-critical system setting, we may aim for establishing that program executions should be correct *and* complete with respect to a set of critical states/states of interest in the labelled transition system.

The next section will give an example of how this is realised in practice with the specific program executions being executions of Java architectural prototypes and where the concrete discrete-state model is a Coloured Petri Net.

4 The Heimdall Tool for Coloured Petri Nets

This section discusses the current implementation of the Heimdall tool that has been used to validate the executable architecture prototype for the frequency converter against the CPN model of the software safety requirements.

4.1 CPN and CPN Tools

Coloured Petri Nets is a formal, graphical modelling language with well-defined syntax and semantics. We will provide a very brief and somewhat informal introduction to CP-nets which is taken from [15]. An example following the formal definition will be used to illustrate several concepts from the definition. The structure of a non-hierarchical CP-net is formally defined as a tuple:

Definition 10 (Coloured Petri Net) *A non-hierarchical CP-net is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$, where Σ is a finite set of non-empty types called colour sets; P, T , and A are non-empty finite, disjoint sets of places, transitions, and arcs, respectively; N is a node function defined from A into $(P \times T) \cup (T \times P)$; C is a colour function defined from P into Σ ; G is a guard function defined from T into boolean expressions; E is an arc expression function defined from A into expressions such that the arc expression for an arc evaluates to a multi-set of values from $C(p)$ where p is the place that the arc is connected to; and I is an initialization function defined from P into expressions that do not contain variables such that the initialization expression for place p evaluates to a multi-set of values from $C(p)$.*

Note that arc and guard expressions may contain variables. A similar definition exists for hierarchical CP-nets, in which modules are connected via well-defined interfaces.

Recall that Figure 4 shows a simplified version of the DigitalIO module for the CPN model described in Section 2.3. The ellipse `UserIO` is a place representing digital inputs and outputs for the two microprocessors. The `UserIO` place acts an interface for this particular module. The colour set for the place is determined by the inscription `UserIO` to the lower left of the place. The states of a CP-net are represented by a number of *tokens* distributed on the places in the model. A token on a place carries a data value, and the type of the data value must correspond to the colour set of the place. Figure 4 shows a state in which there are eight tokens on place `UserIO`, as indicated by the small circle with the number next to the place, and the box next to the small circle shows the values of the eight tokens. Two tokens indicate that the voltage for the digital outputs (which are not shown in Figure 1) for the user feedback (`UserFB`) at microprocessors 1 and 2 are both `Low`. The other six tokens represent the three digital inputs that are used to request safety functions for the two microprocessors. The format for such a data value is `DSFRequest((x,y,voltage))` where x indicates the number of the microprocessor (1 or 2), y indicates the number of the digital input (1, 2, or 3), and `voltage` indicates the voltage of the digital input where there are three possible values (`High`, `Low`, and `Error`).

The formal semantics of CP-nets determine which events can occur in a given state, and how the state will change when a particular event occurs. Events in a system are modelled by transitions. The rectangle Request DSF is a transition that represents the request for the activation of a safety function. The arc expressions on the arcs between UserIO and Request DSF determine how the state of the model will change when the Request DSF transition occurs. The arc expression on the arc from UserIO to Request DSF contains only one variable which is *n*, and it determines that two tokens will be removed from the place when the transition occurs. A transition together with a binding of all of its variables is known as a *binding element*. This transition can only occur if the voltage of digital input *n* at microprocessors 1 and 2 is High. When the transition occurs, two tokens will be added to the place, representing the fact that the voltage of the digital inputs is changed to Low which indicates that a request is being made for the safety function that corresponds to input *n*. In Figure 4 the safety function corresponding to digital inputs 2 has been requested (and possibly activated, but this cannot be seen in this module), but it is currently possible to request the safety functions that correspond to inputs 1 and 3.

Coloured Petri Nets have been used to specify software safety requirements, but we have said that the models that are used with Heimdall must be discrete-state models that can be expressed as an LTS. This is not a problem, because it is possible to define a labelled transition system that is equivalent to a CP-net. Given a CP-net CPN , let $LTS_C = (S_C, i_C, \Lambda_C, T_C)$ where S_C is the set of states of CPN that are reachable by sequences of transition occurrences from the initial state of CPN , i_C is the initial state of CPN , Λ_C is the set of binding elements of CPN , and T_C is the set $\{(s, be, s')\}$ where s is a reachable state of CPN , be is a binding element that is enabled in s , and s' is the state that is reached when be occurs in s .

CPN Tools is a tool supporting the construction and analysis of CP-nets. There is support for running two kinds of simulations: interactive and automatic. In interactive simulations, it is possible for the user to select which transitions should occur. The choice of how transition variables should be bound can either be left to the simulator or the user can manually pick among the legal bindings in a given state. In automatic simulations the simulator randomly picks among the events that are enabled in a given state. In either case, the simulator will update the state of the model after each event occurs.

CPN Tools can execute and analyse models that are equivalent to labelled transition systems, and it therefore fulfills some of the requirements that must be met by the modelling tools that should interact with Heimdall. In order for the Heimdall tool to work with CPN Tools, it must be possible to run and control simulations without (or with very minimal) manual interaction between a user and the GUI of CPN Tools. The simulator for CPN Tools is implemented in Standard ML [22] which means that arbitrary SML functions can be written to control simulations via the predefined primitives in the simulator. The simulator has primitives for running automatic simulations and for selecting which transitions should occur in a simulation, but it lacks a primitive for selecting a transition together with particular bindings of some or all of the variables of the transition. The existing primitives for selecting a particular binding required manual interaction with the GUI by a user. The simulator has been modified, and a new primitive makes it possible to specify that a transition with a particular binding of some of its variables should occur

```
public aspect SafeInverterTracer extends HeimdallTracer {
    pointcut calls() :
        call(* safeinverter.*(..)) &&
        !call(* safeinverter.Factory.*(..)) &&
        !call(* safeinverter..main(..));
    pointcut initializers() :
        initialization(safeinverter.*.new(..));
}
```

Figure 9: Aspect for extracting join points from executable software architecture.

(assuming that the corresponding event can occur in the current state of the model). Support for communicating with the Heimdall tool and for running simulations based on the information received from Heimdall has been implemented in SML, and it will be discussed in Section 4.3.

4.2 Aspects for the Architecture Prototype

As mentioned in Section 2.4 the executable architecture prototype for the frequency converter was implemented as skeleton classes in Java. The classes reflect the design of the software architecture, and they are very simple. Each class contains a number of important methods and, in some cases, some important state variables. The methods are also very simple — they take few, if any, arguments, and the only actions that they perform is that they may update local state variables or call other methods in the architecture prototype.

In order to validate the architecture prototype of the frequency converter, information about join points must be extracted from the prototype during execution, as described in Section 3.1. AspectJ is used for this purpose. We provide an abstract aspect, `HeimdallTracer`, with functionality for communicating with the Program Execution Listener in the Heimdall tool. The aspect allows for tracing of method calls and object constructors. Object constructors are traced in order to provide a object id to correlated with method calls which is necessary in order to distinguish between instances of classes. The default object id is simply derived from the sequence in which objects of interest are constructed, a default approach that may be reasonable in cases where object creation order is deterministic.

The aspect named `SafeInverterTracer` (the new frequency converters are also known as safe inverters), shown in Figure 9, determines which method call join points will be sent to the Heimdall tool. Further, it defines which objects should have their ids tracked. In this case the join points that are to be validated are virtually all method calls in the architecture prototype which is defined in the `safeinverter` package. However, join points for calls to methods in the `Factory` class and calls to main methods will not be sent to the Heimdall tool. The abstract class `SafetyFunction` from Figure 6 has three method call join points that may be validated, namely when the `selfCheck` is performed during initialization of the frequency converter, whenever a call is made to `activate` the safety function, and whenever a check is made to see if a safety function `isRequested`.

We will now turn our attention to the execution mapping for the architecture prototype and the CPN model.

4.3 Mapping Execution Events to Model Events

In the architecture prototype for the frequency converter, the only join points of interest are method call join points. These join points (and join points for the


```

<element>
  <joinpointevents>
    <callevnt>
      <id>4</id>
      <call>safeinverter.external.DigitalIO.requestDSF</call>
    </callevnt>
  </joinpointevents>
  <modelevents>
    <modelevent><id>DigitalIO'Request_DSF(n,3)</id></modelevent>
  </modelevents>
</element>

```

Figure 10: An excerpt from the execution mapping.

```

public class DigitalIO extends IO {
  private SafetyFunction safetyFunction;

  public DigitalIO (State state,
                   SafetyFunction safetyFunction) {
    super(state);
    this.safetyFunction = safetyFunction;
  }
  public void selfCheck() {}
  public void requestDSF() {
    safetyFunction.activate();
  }
}

```

Figure 11: The DigitalIO class from the executable software architecture.

construction of objects of interest, cf. Section 4.2) are specified in the aspect in Figure 9. The architecture prototype contains very few join point for getting or setting fields, and none of these join points need to be validated. Currently, the XML file specifying the mapping must be created manually. The mapping was created after a careful and systematic examination of the architecture prototype and the CPN model.

In the execution mapping for the architecture prototype, each method call join point is mapped to one or more events in the CPN model. Many join points are mapped to just a single transition, while one of the join points is mapped to ten model events. Figure 10 shows an excerpt from the execution mapping. This example shows the XML format of the mapping of a single method call join point to a single model event. In this case a call to the `requestDSF` method to the object with id 4 from the class `safeinverter.external.DigitalIO` is mapped to the model event which is the transition `Request_DSF` (shown in Figure 4) in the module `DigitalIO`, where the variable `n` of the transition is bound to 3. In the CPN model, digital input number 3 is associated with the 'controlled stop' or 'safe delay' safety function. In the architectural prototype, the object with id 4 is an instance of `DigitalIO` that is associated with the safe delay safety function. The `DigitalIO` class is shown in Figure 11.

Let us consider what steps are taken when validating the architecture prototype and the `requestDSF` method is called in a `DigitalIO` object. When the method is called, the `SafeInverterTracer` aspect will cause the signature for the method call as well as the id of the target object to be sent to the Program Execution Listener in the Heimdall tool. The Program Execution Listener will then use the Program Execution Mapper to locate the model events (if any) that the program execution event is mapped to. Given the information in Figure 10, we know that this join point is mapped to a model event corresponding to the transition `Request_DSF` with the variable `n` bound to 3. The textual representation of this model event shown near the bottom of Figure 10 is sent from the Model Tool Communicator to CPN Tools.

A library that allows CPN Tools to interact with

Heimdall has been implemented. This library contains functions for sending and receiving data via a TCP connection with a Model Tool Communicator in Heimdall. Additional functions are used to run simulations based on the commands that are received from the Model Tool Communicator. When a specification of a model event is received from the Model Tool Communicator, there are three possible outcomes. If the event specification corresponds to an event in the model and the corresponding event can occur, then the event will occur in the simulator, and an appropriate response will be returned to the Model Tool Communicator. If the event specification corresponds to an event but the event cannot occur in the current state of the model, then the state of the model remains unchanged, and the response to the Model Tool Communicator indicates that the event cannot occur. The fact that a particular event cannot occur may indicate that the behaviour of the executable code is not consistent with the behaviour specified by the model. Finally, the event specification may not correspond to any known events in the model, and this indicates that there is an inconsistency somewhere, i.e. either in the model, in the executable code, or in the mapping from the code to the model. If the Model Tool Communicator requests that the `Request_DSF` transition should occur, then this is a known event in the model, and a response will be sent back to the Model Tool Communicator indicating either that the event has occurred, thus validating the most recent sequence of join points, or that the event cannot occur in the current state of the model. If the event does not occur in the model, then the program execution has performed a sequence of execution events that cannot be validated, and an error has been found.

5 Discussion and Conclusion

This paper has introduced the Heimdall tool and its associated approach to mapping program execution events to events in a discrete-state model. The tool has been integrated with CPN Tools and has been used to validate architectural prototypes. Even though the evaluations have been made in the context of a real safety-critical system development project, the Heimdall tool can still be considered experimental in nature.

First of all, a full validation during a development project is needed. This will stress the usability of the actual mapping mechanism. The current mapping mechanism is essentially simple since one of our goals have been to support experimentation with mapping from architectural prototype executions and other types of program executions. One area in which the mapping mechanism could be improved is in considering transitions that do not correspond to method calls. It should be possible for them to occur if they are enabled: e.g., if a particular transition that is mapped from a method invocation is not enabled, then it might become enabled if one or more of the unmapped transitions/events occur.

Also a more thorough evaluation could potentially illustrate to which extent architectural prototyping is actually useful and beneficial in safety-critical system development.

Secondly, there is a definite lack of proper tool support for constructing Heimdall mappings. One step in this direction would be to be able to generate a set of possible program execution events/model events to base the mapping construction on. In particular if iterations on the software architecture and models are considered, better tool support is of importance.

Even though the Heimdall tool can in no way prove that a specific architecture will lead to safe software,

it may help in doing so by allowing architects to experiment with and partly validate their architectural designs thus potentially leading to better and safer software architectures.

References

- [1] M. R. Barbacci, R. Ellison, A. J. Lattanze, J. A. Stafford, C. B. Weinstock, and W. G. Wood. Quality Attribute Workshops (QAWs), Third Edition. Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, 2003.
- [2] J. Bardram, H. B. Christensen, and K. M. Hansen. Architectural Prototyping: An Approach for Grounding Architectural Design and Learning. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 15–24, Oslo, Norway, 2004.
- [3] J. Bardram, H. B. Christensen, and K. M. Hansen. Exploring quality attributes using architectural prototyping. In *Proceedings of the First International Conference on the Quality of Software Architectures, QoSA 2005*, volume 3712 of *LNCS*, pages 155–170, Erfurt, Germany, 2005.
- [4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.
- [5] L. Bass, M. Klein, and F. Bachmann. Quality attribute design primitives and the attribute driven design method. In *Proceedings of the 4th International Workshop on Product Family Engineering*, 2001.
- [6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- [7] CPN Tools. Online: <http://www.daimi.au.dk/CPNTools/>.
- [8] L. Dobrica and E. Niemela. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.
- [9] DEF (AUST) 5679: *The Procurement of Computer-based Safety Critical Systems*, 1998. Australian Defence Standard.
- [10] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
- [11] M. Heimdahl and N. Leveson. Completeness and consistency checking of software requirements. *IEEE Transactions on Software Engineering*, 22(6), 1996.
- [12] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1999.
- [13] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
- [14] International Electrotechnical Commission. *Functional Safety of Electrical/ Electronic/ Programmable Electronic Safety-Related Systems*, 1st edition, 1998–2000. International Standard IEC 61508, Parts 1–7.
- [15] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.
- [16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353, 2001.
- [17] L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner’s guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.
- [18] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.
- [19] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.
- [20] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [21] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–256, 1986.
- [22] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.
- [23] K. H. Pedersen and C. Constantinides. Aspectada: Aspect oriented programming for Ada95. In *SigAda ’05: Proceedings of the 2005 annual ACM SIGAda international conference on Ada*, pages 79–92, New York, NY, USA, 2005. ACM Press.
- [24] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: an AOP extension for C++. *Software Developer’s Journal*, (5):68–76, 2005.
- [25] visualSTATE. Online: <http://www.iar.com/vs>.
- [26] L. Wells and T. Maier. Specifying and analyzing software safety requirements of a frequency converter using coloured Petri nets. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *LNCS*, pages 403–422. Springer, 2005.

Assuring Separation of Safety and Non-safety Related Systems

Bruce Hunter

Thales Training & Simulation
Thales Services Division, Building 314, Garden Island, Sydney
Locked Bag 2700, Potts Point, NSW 2011, Australia

bruce.hunter@thalesgroup.com

Abstract

Safety standards call for the separation of safety and non-safety related systems. Although good guidance is provided in these standards on how to achieve the required hazard analysis, safety integrity assignment and validation to prove a safe system, there is little available on establishing safety boundaries around the critical components and the proof of isolation from non-safety functions. Delineation between safety and non-safety systems is particularly important where it is impractical to substantiate a Safety Integrity Level of the overall system due to the complexity of some components. In this case it is better to assume high failure probability of the non-safety system and prove isolation from the safety-related system.

This paper explores a conceptual methodology (including the use of Fault Tree Analysis and Common Cause Failure Analysis) for establishing and assuring separation of systems and some examples from training simulators that are an example of this situation drawn from real-life.

Keywords: Functional Safety Separation, Functional Safety Boundaries, Simulator Functional Safety.

1 Introduction

Separating safety-critical and safety-related systems from systems where safety integrity is unable to be established or maintained is an important aspect of system safety design. When implementing a system safety program it is important to *suspect all* components as being unsafe unless assured otherwise and then *target the few* areas where safety requirements are allocated. Coupling between components of complex systems can be subtle and interaction with non-safety related systems have led to harmful outcomes in safety related systems.

An example of this coupling occurred on 19 February 1996, when a Boeing 747-433 Combi aircraft operating as Air Canada flight 899, was on a scheduled passenger/freight flight from Toronto/Lester B. Pearson International Airport, Ontario, to Vancouver International Airport, British Columbia. As the aircraft was taking off, the underside of the tail struck the runway, and, during the climb-out, considerable nose-down stabilizer trim was

required to trim the aircraft for flight. The Canadian Transport Safety Board (TSB) report (A9600030) findings included:

- “A recently modified computer application, ALPAC, used by load agents to plan loads and compute aircraft weight and balance, incorrectly computed the aircraft take-off C of G.
- The ALPAC-computed aircraft take-off C of G was near the centre of the aircraft flight envelope, while the actual C of G was beyond the aft limit.
- The ALPAC application produced a large error in the aircraft C of G calculation; however, there was no defence in place to detect such a critical error in the application itself, at the aircraft loading stage, or in the flight crew confirmations of load and trim setting.
- The modified computer application was not adequately tested before it was released for operational use.
- The modified computer application was not monitored effectively for accuracy after it was placed in operational use.”

In this case the software that led to the incident was not even aboard the aircraft and was operated by a different party. Interaction across what appears to be valid safety boundaries can sometimes be nebulous. While this failure may be considered as incomplete hazard analysis of the changes to the ALPAC system and the impact on the performance of the loaded aircraft, it also can provide a good example of where coupling between systems may be overlooked.

2 Setting Functional Safety Boundaries

2.1 The need for having boundaries

Taking the extreme position, very few systems are fully independent in their operation and to be completely assured of the absence of interaction or common-cause failure between the safety-related and other systems would take an inordinate amount of time and effort. This could cause the opposite effect to delay introducing the safety benefits of the deployment of a safety-related system. At some point a determination must be made that all possible influences are controlled or risks sufficiently known so the safety analysis can be bounded.

Taking the above tail-strike incident as an example of an indirect influence on system safety, the safety analysis boundary could well be established around the flight

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at the *11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, Melbourne. Conferences in Research and Practice in Information Technology, Vol. 69. Tony Cant, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

control systems. Further investigation of the use of the off-board planning system would have identified its criticality to the Centre of Gravity of the loaded aircraft and extended the functional safety boundary to include this and any changes made to it.

2.2 Objectives of Functional Safety Boundaries

This paper introduces a concept of Functional Safety Boundaries, which can be used to contain areas where specific safety integrity measures are to be employed. Objectives of these boundaries are to:

- Minimise the interfaces across the safety boundary to direct focus on the safety separation implemented in these;
- Minimise likelihood of common-cause failures across the boundary;
- Exclude non-safety related functions where these are volatile or subject to undefined or non-safety related controls; and
- Allow a Safety Integrity Level (SIL) to be achieved within the boundary.

2.3 Identifying safety functions within a boundary

A useful method to establish the functional safety boundary between systems or subsystems is to undertake a Fault Tree Analysis (FTA) of the contributing factors to failure of the system, which may lead to hazardous events identified in the preliminary hazard analysis. The first attempt at a boundary would be around the systems that are implicated in the FTA. This FTA needs to be extensive and complete from all initiating situations to the system failure that is a causal factor for the hazardous event. Then flowing down the tree, mark off those functions that are related to systems that should be excluded due to:

- The possibility of common-cause failure;
- High levels of complexity and non-deterministic failure rate; or
- Components that may not always be present or enabled.

Failure probabilities are then assigned to the contributing and basic events. Figure 1 shows a very simplified fault tree for a safety-related system and its isolation from non-safety-related and non-deterministic functions (SIL0).

In Figure 2, the boundary is set around the failure associated with the SIL0 system (A) which then requires the failure probability of the associated protective isolation mechanism (B) to be made no less than the failure probability (C) of the SIL rated system within the boundary to achieve an end failure probability commensurate with the SIL rated system (E).

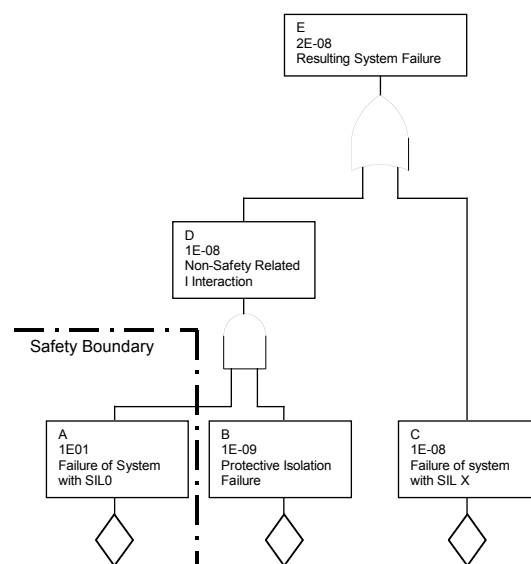


Figure 1 Simple FTA of Safety System Coupling

In a similar manner, the boundary must be extended to include common-cause failures that effectively defeat the independence across the boundary as shown in Figure 2.

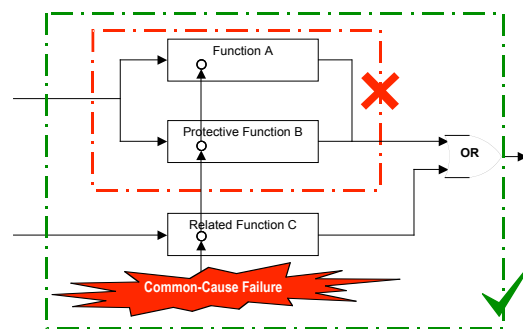


Figure 2 Setting boundaries outside possible Common-Cause Failures

2.4 The problem with software

At a system level, this process looks reasonably straightforward but the problem comes with setting boundaries with distributed software architectures. In this situation it is very difficult to identify boundaries that don't involve the possibility of common-cause failures. Some useful work on partitioning in this context has been done by Conmy, Nicholson, Purwantoro and McDermid (2002), *Identifying Safety Dependencies in Modular Computer Systems*.

If the layering approach from this work is extended to a generic model, common cause failures can be seen to involve lower layers of the architecture (hardware failures, resource sharing failures, communication failures, memory leakage failures etc). For this reason it is essential that any functional safety boundary must include all the layers that support that function, as shown in Figure 3.

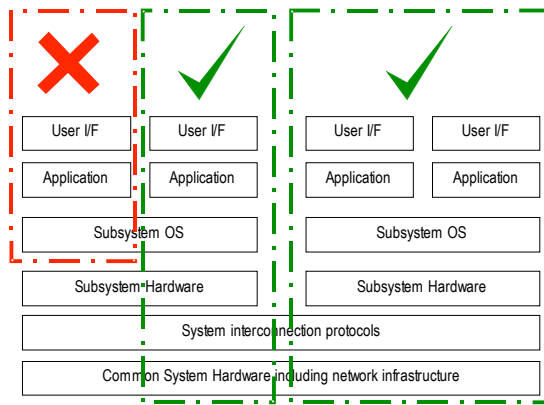


Figure 3 Distributed system acceptable boundaries

Common-cause failures and dependencies extending over the distributed communication networks must also be considered and the functional safety boundary set accordingly. These may include:

- Global variables accessed by network
- Security attack and security blocking issues
- Affects of network lock-up on functional safety

The separation requirements over the functional safety boundary must take these failures into account.

2.5 Setting boundaries in the safety lifecycle

As part of the safety lifecycle, identification of Functional Safety Boundaries and Functional Safety Separation should be included in setting overall safety requirements and the allocation of these to systems and their components. The following table identifies the lifecycle phases from IEC 61508, *Functional safety of electrical/electronic/ programmable electronic safety-related systems*, where segmentation and separation of safety should be undertaken.

IEC 61508 Safety Lifecycle	Functional Safety Separation Activities
Phase 4. Overall Safety Requirements	Determine safety boundaries
Phase 5. Overall Safety Requirement Allocation	Determine separation requirements
Phase 9. System Safety Requirements Specification	Specify trans-boundary information allowed and prohibited
Phase 10. Safety-related Systems Realisation	Establishment of separation measures
Phase 13. Overall Safety Validation	Proof of separation of non-safety systems or influences
Phase 14. Overall Operation, Maintenance and Repair	Monitoring for compromised separation
Phase 15. Overall Modification and Retrofit	Re-evaluating safety boundaries and separation

Table 1: Lifecycle Consideration of Safety Boundaries and Separation

3 Assuring Functional Safety Separation

Safety standards do call for independence of safety-related functions but aren't very specific about what is acceptable or how to dependably achieve this. Although it is a difficult area to quantify for completeness and repeatability, I believe it is important that standards don't avoid addressing this issue and should specify a generic methodology for assuring independence or separation.

3.1 What the standards say

3.1.1 Key IEC 61508 extracts

IEC 61508 identifies qualitative requirements for independence of safety-related functions.

- IEC 61508-2 clause 7.4.2.3 “Where an E/E/PE safety-related system is to implement both safety and non-safety functions, then all the hardware and software shall be treated as safety-related unless it can be shown that the implementation of the safety and non-safety functions is sufficiently independent (i.e. that the failure of any non-safety-related functions does not cause a dangerous failure of the safety-related functions). Wherever practicable, the safety-related functions should be separated from the non-safety-related functions.”

NOTE 1 Sufficient independence of implementation is established by showing that the probability of a dependent failure between the non-safety and safety-related parts is sufficiently low in comparison with the highest safety integrity level associated with the safety functions involved.”

- IEC 61508-2 clause 7.4.2.5 “When independence between safety functions is required (see 7.4.2.3 and 7.4.2.4) then the following shall be documented during the design:

- a) the method of achieving independence;
- b) the justification of the method.”

Although not quantified, this does support the use of safety boundary setting (in Section 2) and for identifying the level of separation (Section 3.2). However this does allow varying levels of rigour in establishing the required independence.

IEC 61508-3 (Software Requirements) clause 7.4.2.7 has requirements requiring: “Where the software is to implement both safety and non-safety functions, then all of the software shall be treated as safety-related, unless adequate independence between the functions can be demonstrated in the design.”

Clause 7.4.2.8 also requires “Where the software is to implement safety functions of different safety integrity levels, then all of the software shall be treated as belonging to the highest safety integrity level, unless adequate independence between the safety functions of the different safety integrity levels can be shown in the design. The justification for independence shall be documented.”

The concept of Safety Separation Levels could be the basis for demonstration of this “independence” for these clauses.

Notes to clause 7.4.2.8, in the new committee draft, expand the requirements that allow independence to be shown on a single computer system by means of spatial and temporal techniques. In my view some of these may further erode the rigour required by the standard due to the lack of formality in establishing this independence unless substantiated by some form of quantification of the independence level required.

3.1.2 DEF (AUST) 5679A extracts

DEF (AUST) 5679A, *The procurement of Computer-Based Safety-Critical Systems*, still has a qualitative approach but is more specific about the requirements of independence and its dimensions to be considered.

- Section 15.5 Component Independence - “.... One Component is independent of another if its operation cannot be changed, misdirected, delayed or inhibited by the other Component.
- Section 15.5.2 “The notion of Component independence has several dimensions. These include:
 - a) physical isolation (for example with software components this means that each Component runs on a separate processor);
 - b) diversity of implementation, for example, one Component may be implemented in software, another implemented by hardware or operator procedure;
 - c) data independence (for example, the input data for the Components is not to be generated by the same mechanism); and
 - d) control independence, meaning that one Component cannot affect the control flow of another Component...”
- Section 15.5.4 “If a SIL assignment depends on the independence of components, evidence of the independence shall be documented. The documented evidence shall state how independence is achieved and how independence is used as a protective measure.”

DEF (AUST) 5679 is quite helpful in identifying some key concepts of independence along with required practices and evidence that components can be considered as independent. I believe that the techniques of functional safety boundaries and Safety Separation Level would satisfy the evidence required.

3.2 Concept of Safety Separation Level (SSL)

A means of quantifying and comparing independence could be achieved with the use of a Safety Separation Level achieved by the assignment of failure of separation probability equivalent to the SIL target failures of IEC 61508-1 7.6.2.9 as shown in Table 2

SSL	Probability of propagating dangerous failure for low demand mode (<1 per year)	Probability of propagating dangerous failure for continuous/high-demand mode
4	$\Rightarrow 10^{-5}$ to $<10^{-4}$	$\Rightarrow 10^{-9}$ to $<10^{-8}$
3	$\Rightarrow 10^{-4}$ to $<10^{-3}$	$\Rightarrow 10^{-8}$ to $<10^{-7}$
2	$\Rightarrow 10^{-3}$ to $<10^{-2}$	$\Rightarrow 10^{-7}$ to $<10^{-6}$
1	$\Rightarrow 10^{-2}$ to $<10^{-1}$	$\Rightarrow 10^{-6}$ to $<10^{-5}$

Table 2 Proposed allocation of dangerous interaction probability to SSL

Taking this concept further, Table 3 shows a proposed method of assigning a Safety Separation Level (SSL) to differences in SIL across safety boundary. This attempt to quantify independence between safety-related systems meets the intent of IEC 61508 parts 2 and 3 and DEF(AUST) 5679A section 15.5.

		System 1				
		Unclaimed (SIL0)	SIL1	SIL2	SIL3	SIL4
System 2	Unclaimed (SIL0)	N/A	SSL1	SSL2	SSL3	SSL4
	SIL1	SSL1	N/A	SSL1	SSL2	SSL3
	SIL2	SSL2	SSL1	N/A	SSL1	SSL2
	SIL3	SSL3	SSL2	SSL1	N/A	SSL1
	SIL4	SSL4	SSL3	SSL2	SSL1	N/A

Table 3: Allocation of SSL to differences in systems SIL ratings

Relating this back to the FTA model of separation in Figure 1, independence between SIL0 and SIL4 systems would require an SSL of 4, equivalent to the reliability of a SIL 4 system. Achievement of these separation levels could use similar compliance routes identified in IEC 61508-2 section 7.4. Establishing Safety Integrity Levels in a homogeneous system without external interfaces is adequately although sometime controversially dealt with in existing standards. The relationship between SIL differences and proposed minimum requirements for SSL would need further work to justify more than the extremes. Simply, where the SIL requirement is the same, this is effectively an extension of the safety system therefore no SSL is required. Where there is an interface to SIL0 system this requires the same rigour as the higher integrity system.

3.3 Setting separation requirements

Establishing the level of independence could use the effective definitions in DEF (AUST) 5679A section 15.5.2 where each of the dimensional attributes would be assessed against separation characteristics commensurate with the SSL required from Table 3.

These independence dimensions (physical, data independence and control independence, and diversity of implementation) should be assessed for ability to change, misdirect, delay or inhibit safety functions of the safety related system across the functional safety boundary.

Independence Dimension	SSL0	SSL1	SSL2	SSL3	SSL4
Diversity	Common development and design implementation.	TBD	TBD	Separate subsystem development approaches and system implementation. Strong prevention of CCF across FSB	Independent development and design technologies. Thorough prevention of CCF across FSB
Physical	Fully integrated (e.g. single MCU)	TBD	TBD	In separate enclosures with special or physical protection.	Fully separated (e.g. housing, environment, power, access)
Data	Dependent on data across system (e.g. global variables)	TBD	TBD	Strong checking on out of range data and protection against flooding of information	No data dependencies across FSB except contained within approved controls or read-only access.
Control	Many system-wide controls without limitation of their impact	TBD	TBD	Few controls and all verified and authenticated for dangerous impact.	Few controls and all verified and authenticated for dangerous impact.

Table 4 Possible SSL independence attributes

Setting a common process for this assessment would require considerable development and agreement before inclusion in a standard could be contemplated but Table 4 proposes a possible framework (albeit incomplete in this paper) where assignment to SSL objectives may be possible. Further work and substantiation would be required on the assignment of separation levels, but in my view this would be beneficial to accommodate the complexity of emerging systems.

3.4 Separation in the Maintenance Lifecycle

One of the strengths of IEC 61508 is the full life-cycle approach that it takes in respect of establishing and maintaining functional safety. This is particularly important with maintaining independence across functional safety boundaries, as changes to maintenance, repair and updates could defeat the isolation measures taken.

Due to the subtlety and far reaching impact of some safety separation issues (see Introduction), continuing independence of these is at threat of being compromised through the support, maintenance and upgrade phases of the safety lifecycle. Like other safety requirements, the implementation of functional safety separation must be fully identified in the Safety Case and maintenance documentation. This must be revisited on a regular basis to ensure no unauthorised modifications have been made when changes to the system are made to ensure effective functional safety separation is maintained.

4 Practical Examples in Simulator Systems

4.1 Simulation domain specific safety issues

In simulator training devices, the combination of safety and non-safety related systems is an inevitable consequence of the systems involved and the direct interface to trainees through the simulation cues of visual, motion, aural and force-feedback.

One often-identified risk of training simulators is negative training indirectly leading to bad practices on the real platform. To mitigate this risk, full-flight simulators are accredited to standards prior to being placed into service and training credits being claimed. Fidelity checks are based on many factors, including model checks with real aircraft data and cues associated with key training competencies.

Taking the example of the 747 aircraft tail scrape in this paper's introduction, one of the findings of the TSB was: "The first officer's recent simulator training did not include an aircraft out-of-trim or out-of-balance take-off". The safety functional boundary for this scenario could have been extended beyond the operation of the aircraft to the specific training task and cues on the simulator. If this was considered then, so long as the simulator faithfully represented the aircraft and controls under these conditions, then the simulator has fulfilled its requirements.

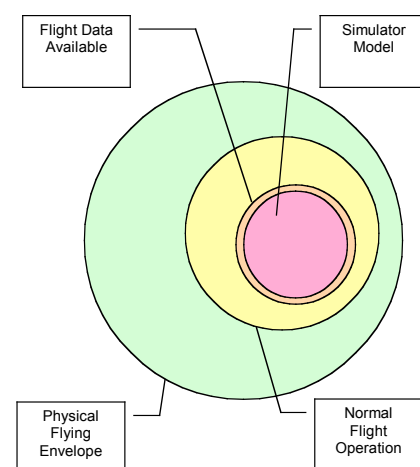


Figure 4 Simulator Modelled Space

The nature of general simulator architectures and the modelling of the aircraft operation do cause real problems in assigning an overall safety integrity level. Figure 4 graphically illustrates this limitation with the impracticality of simulating the complete behaviour of all platform systems in all conditions.

4.2 Generic simulator hazards



© Thales, used by permission

Simulators are different from real aircraft in several important areas: they are meant to crash without injury to the occupants; and they only simulate key areas of the aircraft functions. The key hazards associated with simulators are the integrated human interfaces associated with training cues as shown in Table 5.

Hazardous Cues	Dangerous failure impacts
Aural Cues	Issues of occupational deafness if sustained excessive volume
Motion Cues	Issues of crushing, falling and hitting
Visual Cues	No direct hazard other than motion sickness
Control Loading Cues	Feedback Cues – issues of entrapment, crush and strike.
Combination	Simulator motion sickness due to the concentration and limited accuracy possible in the simulated environment.

Table 5 Simulator Hazards and Impact Issues

4.3 Example of motion System Safety Integrity

One of the key cues associated with full-flight simulator systems is the “feel” of motion associated with aircraft movement and attitude. The motion system of the simulator takes the acceleration vectors and aircraft attitude from the simulated model and typically applies these to a six-degrees of freedom hydraulic motion platform.

The motion system is considered a safety related system due to the large excursions of movement. The safety boundary of this system encloses all the necessary controls to ensure safe operation and shutdown of the motion system as shown in Figure 5.

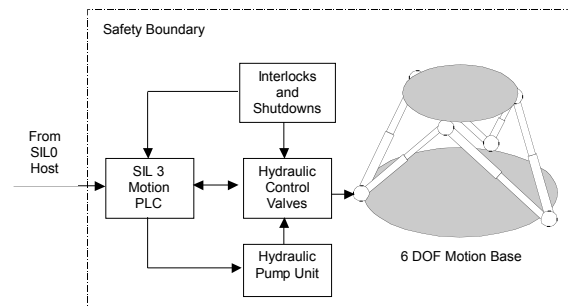
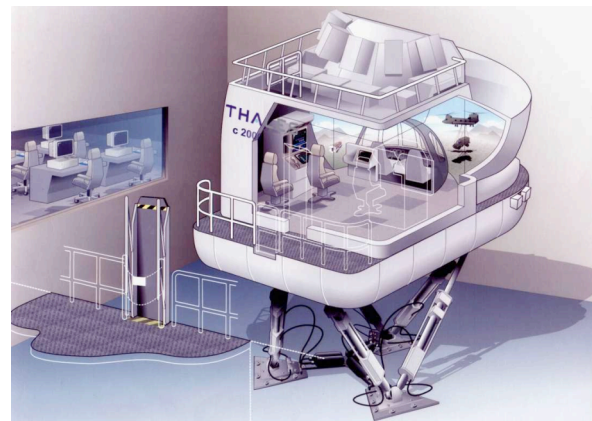


Figure 5 Simulator Motion System Overview

Data and control across the Functional Safety Boundary in this case is limited to acceleration and direction information. Local control is applied between the SIL 3 Safety PLC and the Motion hydraulic controls complete with integrated safety interlocks and emergency shutdowns. While ever the motion requests are within the bounds of acceptable limits the motion system will respond once the instructor gives consent and interlocks remain inactive (data and control independence).



© Thales, used by permission

Figure 6 C2000X Full Flight Simulator Cut-away

The motion systems is physically separated from the rest of the system and independently developed and implemented with different technology (physical independence and diversity). Separation across this safety boundary can be claimed to meet equivalent SSL 3 requirements as per Table 4 proposal and maintained to meet the accepted risk profile. This would then maintain the motion system PLC as SIL3 without degrading by connection to the host system, which cannot be substantiated as anything more than SIL0.

5 Conclusion

This paper has proposed a technique to quantify and implement separation of safety-related systems from other systems by recognising safety boundaries and interaction across those boundaries and their effect on the separation. This technique re-uses methods from existing standards to measure, implement and maintain separation based on the concept of Safety Separation Level with similar criteria to Safety Integrity Level and Functional Safety Boundaries. This allows the safety case to be established for complex systems by applying quantifiable separation requirements to systems where a SIL is difficult, if not impossible, to obtain at the overall system level.

6 Acknowledgements

The author gratefully acknowledges the assistance and support of Philip Swadling and Stephen Carey of Thales Training & Simulation and the independent reviewers for

this conference whose valuable contribution has helped me complete this paper.

7 References

- IEC 61508 (1998) Parts 1 to 4
Functional safety of electrical/ electronic/ programmable electronic safety-related systems.
- DEF (AUST) 5679 (1998) Land Engineering Agency, Department of Defence. *The procurement of Computer-Based Safety-Critical Systems.*
- Transport Safety Bureau of Canada Report Number A96O0030, *Control Difficulty Tail Strike, Air Canada Boeing 747-433 Combi C-GAGL Toronto/Lester B. Pearson, International Airport, Ontario, 19 February 1996.*
- Conmy, P., Nicholson, M., Purwanto, Y.,M., and McDermid, J. (2002) *Safety Analysis and Certification of Open Distributed Systems.*

Using Software Architecture Techniques to Support the Modular Certification of Safety-Critical Systems

Tim Kelly

Department of Computer Science
University of York
Heslington, York, YO10 5DD
United Kingdom
tim.kelly@cs.york.ac.uk

Abstract¹

In software engineering the role of *software architecture* as a means of managing complexity and achieving emergent qualities such as modifiability is increasingly well understood. In this paper we demonstrate how many principles from the field of software architecture can be brought across to the field of safety case management in order to help manage complex safety cases.

Traditional approaches to certification of modular systems as a statically defined configuration of components can result in a large certification overhead being associated with any module update or addition. A more promising approach is to attempt to establish a modular, compositional, approach to constructing safety cases that has a correspondence with the modular structure of the underlying architecture. This paper establishes the mechanisms for managing and representing safety cases as a composition of safety case ‘modules’. Having defined the concept of a modular safety case, the paper also describes principles for their definition and evaluation. An example generic modular safety case architecture for Integrated Modular Avionics (IMA) based systems is presented as a means of illustrating the concepts defined.

Keywords: safety case, modularity, certification, justification, composition, architecture

1 Introduction

Whilst the move towards modular systems and software architecture offers potential benefits of improved flexibility in function allocation, reduced development costs and improved maintainability, it can pose significant problems in certification. The traditional approach to certification relies heavily upon a system being statically defined as a complete entity and the corresponding (bespoke) system safety case being constructed. However, a principal motivation behind modular systems construction is that there is through-life (and potentially run-time) flexibility in the system configuration. For example, an Integrated Modular Avionics (IMA) system

can typically support many possible mappings of the avionics functionality required to the underlying computing platform.

In constructing a safety case for modular systems an attempt could be made to enumerate and justify all possible configurations within the architecture. However, this approach is unfeasibly expensive for all but a small number of processing units and functions. Another approach is to establish the safety case for a specific configuration of the architecture. However, this nullifies the benefit of flexibility in using a modular solution and will necessitate the development of completely new safety cases for future modifications or additions to the architecture.

A more promising approach is to attempt to establish a modular, compositional, approach to constructing safety cases that has a correspondence with the modular structure of the underlying architecture. As with software architecture it would need to be possible to establish interfaces between the modular elements of the safety justification such that safety case elements may be safely composed, removed and replaced. Similarly, as with software architecture, it will be necessary to establish the safety argument infrastructure required in order to support such modular reasoning (e.g. an infrastructure argument regarding partitioning being required in order to enable independent reasoning concerning the safety of two system elements).

By adopting a modular, compositional, approach to safety case construction it may be possible to:

- Justifiably limit the extent of safety case modification and revalidation required following anticipated system changes
- Support (and justify) extensions and modifications to a ‘baseline’ safety case
- Establish a family of safety case variants to justify the safety of a system in different configurations

2 Current Safety Case Development Practice

Safety case reports are often complex documents presenting complex arguments. Very rarely is it that safety cases are prepared by individuals. The reality is that the activity of establishing a safety case will be divided amongst a number of individuals, teams and, in some cases, organisations.

¹ Copyright © 2006, Australian Computer Society, Inc. This paper appeared at the *11th Australian Workshop on Safety-Related Programmable Systems (SCS'06)*, Melbourne. Conferences in Research and Practice in Information Technology, Vol. 69. Tony Cant, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included

To manage the complexity of safety case construction, system safety cases are often decomposed into subsystem safety cases. Many examples of this can be observed in current safety-critical systems: System safety cases incorporate software safety cases (a division advocated by Issue 2 of the U.K. Defence Standards 00-55 (MoD 1997) and 00-56 (MoD 1996)). A safety case concerned with the avionics of a complex military aircraft will be split into separate safety cases for separate systems (such as the navigation system, engine control and flight control). The implied safety case for UK rail operations is made up of separate safety cases for station operations, infrastructure and rolling stock. However, it is well understood that safety is a system property. Extreme care must therefore be taken to ensure that safety case boundaries are drawn correctly, that arguments don't "fall between the gaps", and that formalising boundaries (e.g. through contractual agreements between organisations) doesn't prevent the development of efficient safety solutions. Emergent safety properties, not dealt with by a "Divide and Conquer" approach to safety case construction, must also be addressed.

Although the above examples already exist in practice, the overall structure 'in-the-large' of these safety cases and the interdependencies that exist between them are often poorly managed. In the following sections, we show it is possible to map across principles already established in the field of software architecture to help address this problem.

3 Safety Case Architecture

Software architecture has been defined in the following terms (Bass et al. 1998):

"The structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them"

Safety case architecture can be defined in very similar terms:

The high level organisation of the safety case into components of arguments and evidence, the externally visible properties of these components, and the interdependencies that exist between them

Being clear of the externally visible properties of any safety case module allows us to appreciate its role within the overall safety case structure. The following can be regarded as the key 'interface' properties for any safety case module:

1. Objectives addressed by the module
2. Evidence presented within the module
3. Context defined within the module

It is important to note that the definition of safety case architecture must give equal importance to the dependencies between safety case modules (or 'components') as to the components themselves. This thinking must be at the heart of any attempt to decompose the safety case. Safety is not a "sum of parts" property.

Dependencies must therefore also be recorded as part of any interface definition, perhaps along the following lines:

4. Arguments requiring support from other modules
5. Reliance on objectives addressed elsewhere
6. Reliance on evidence presented elsewhere
7. Reliance on context defined elsewhere

Safety case modules can be usefully composed if their objectives and arguments complement each other – i.e. one or more of the objectives supported by a module match one or more of the arguments requiring support in the other. For example, the software safety argument is usefully composed with the system safety argument if the software argument supports one or more of objectives set by the system argument. At the same time, an important side-condition is that the collective evidence and assumed context of one module is consistent with that presented in the other. For example, the operational usage context assumed within the software safety argument must be consistent with that put forward within the system level argument.

4 Representing Modular Safety Cases in GSN

The Goal Structuring Notation (GSN) (Kelly 1997) - a graphical argumentation notation - explicitly represents the individual elements of any safety argument (requirements, claims, evidence and context) and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific claims, how claims are supported by evidence and the assumed context that is defined for the argument). The principal symbols of the notation are shown in Figure 1 (with example instances of each concept).

The principal purpose of a goal structure is to show how goals (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence (solutions). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (assumptions, justifications) and the context in which goals are stated (e.g. the system scope or the assumed operational role). For further details on GSN see (Kelly 1997).

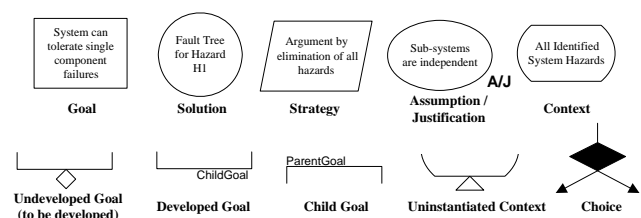


Figure 1: Principal Elements of the Goal Structuring Notation

GSN has been widely adopted by safety-critical industries for the presentation of safety arguments within safety

cases. However, to date GSN has largely been used for arguments that can be defined ‘stand-alone’ as a single artefact rather than as a series of modularised interconnected arguments. In order to make the GSN support the concepts of modular safety case construction it has been necessary to make a number of extensions to the core notation.

The first extension to GSN is an explicit representation of modules themselves. This is required, for example, in order to be able to represent a module as providing the solution for a goal. For this purpose, the package notation from the Unified Modelling Language (UML) standard has been adopted. The new GSN symbol for a safety case module is shown in Figure 2 (Right Hand Side).

As has already been discussed, in presenting a modularised argument it is necessary to be able to refer to goals (claims) defined within other modules. Figure 2 (left hand side) introduces a new element to the GSN for this purpose – the “Away Goal”. An away goal is a goal that is not defined (and supported) within the module where it is presented but is instead defined (and supported) in another module. The Module Identifier (shown at the bottom of the away goal next to the module symbol) should show the unique reference to the module where the goal can be found.

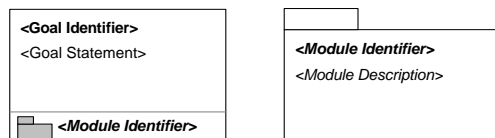


Figure 2: GSN Elements Introduced to Handle Modularity

Away goals can be used to provide *support* for the argument within a module, e.g. supporting a goal or supporting an argument strategy. Away goals can also be used to provide contextual backing for goals, strategies and solutions.

Representation of away goals and modules within a safety argument is illustrated within Figure 3. The annotation of the top goal within this figure “SysAccSafe” with a module icon in the top right corner denotes that this is a ‘public’ goal that would be visible as part of the published interface for the entire argument shown in Figure 3 as one of the “objectives addressed”.

The strategy presented within Figure 3 to address the top goal “SysAccSafe” is to argue the safety of each individual safety-related function in turn, as shown in the decomposed goals “FnASafe”, “FnBSafe” and “FnCSafe”. Underlying the viability of this strategy is the assumed claim that all the system functions are independent. However, this argument is not expanded within this “module” of argument. Instead, the strategy makes reference to this claim being addressed within another module called “IndependenceArg” – as shown at the bottom of the away goal symbol. The claim “FnASafe” is similarly not expanded within this module of argument. Instead, the structure shows the goal being supported by another argument module called “FnAArgument”. The “FnBSafe” claim is similarly shown to be supported by

means of an Away Goal reference to the “FnBArgument” module. The final claim, “FnCSafe”, remains undeveloped (and therefore requiring support) – as denoted by the diamond attached to the bottom of the goal.

In the same way that in can it be useful to represent the aggregated dependencies between software modules in order to gain an appreciation of how modules interrelate “in-the-large” (e.g. as described in the “Module View” of Software Architecture proposed by Hofmeister et al. in (Hofmeister et al., 1999)) it can also be useful to express a module view between safety case modules.

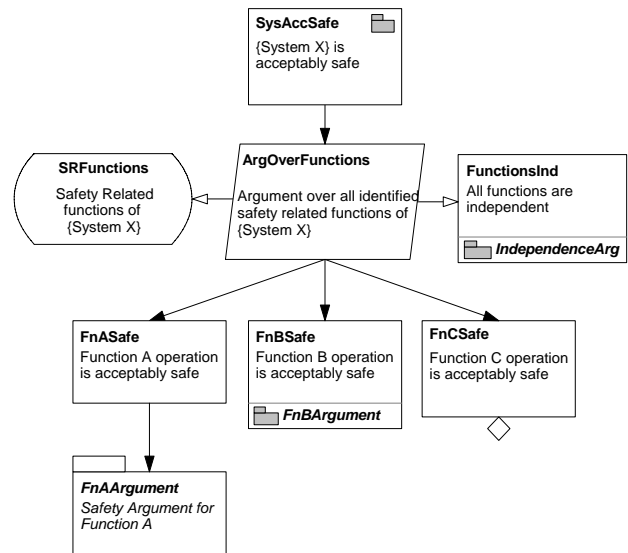


Figure 3: Representing Safety Case Modules and Module References in GSN

In the same way that in can it be useful to represent the aggregated dependencies between software modules in order to gain an appreciation of how modules interrelate ‘in-the-large’ (e.g. as described in the ‘Module View’ of Software Architecture proposed by Hofmeister et al. in (Hofmeister et al. 1999)) it can also be useful to express a module view between safety case modules.

If the argument presented within Figure 3 was packaged as the “TopLevelArg” Module, Figure 4 represents the module view that can be used to summarise the dependencies that exist between modules. Because the “FnAArgument” and “FnBArgument” modules are used to support claims within the “TopLevelArg” module a supporting role is communicated. Because the “IndependenceArg” module supports a claim assumed as context to the arguments presented in “TopLevelArg” a contextual link between these modules is shown.

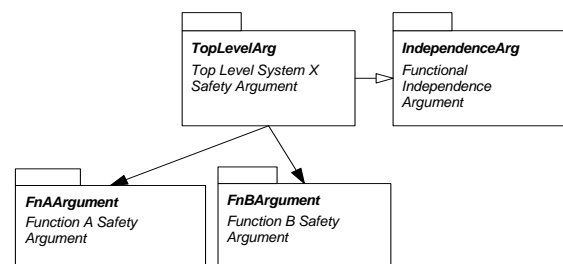


Figure 4 – Example Safety Argument Module View

In a safety case module view, such as that illustrated in Figure 4, it is important to recognise that the presence of a *SolvedBy* relationship between modules A & B implies that there exists at least goal within module A that is supported by an argument within module B. Similarly, the existence of an *InContextOf* relationship between modules A & B implies that there exists at least one contextual reference within module A to an element of the argument within module B.

Alongside the extensions to the graphical notation of GSN, the following items of supporting documentation are required:

Interface declaration for each safety case module – along the lines outlined in section 3, the external visible properties of any safety case module must be recorded – e.g. the goals it supports, the evidence (solutions) it presents, the cross-references ('Away Goal' references) made to / dependencies upon other modules of argument. Figure 5 depicts the items to be defined on the boundary of a safety case module expressed using the GSN.

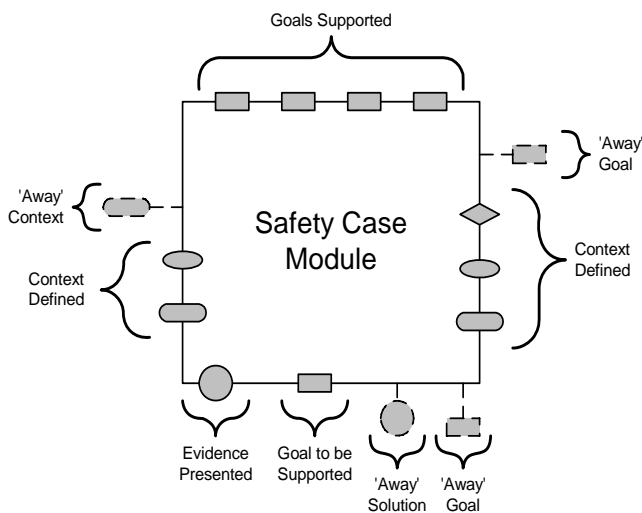


Figure 5 – The Published Interface of a GSN Safety Case Module

Contracts for composed modules – where co-dependent safety case modules are used together within a system safety case a contract must be recorded of the dependencies resolved between the separate arguments. This is discussed further in section 6.

5 Module Composition with GSN Modules

The following three steps must be undertaken when attempting to usefully compose two safety case modules A & B with interfaces as defined in the previous section:

Step 1 – Goal Matching

- Assess whether any of the goals requiring support in Module A (i.e. those listed under item 4 of the declared interface for Module A) match the goals addressed by Module B (i.e. those listed under item 1 of the interface for Module B).
- Conversely, assess whether any of the goals requiring support in Module B (i.e. those listed under item 4 of the declared interface for Module

B) match the goals addressed by Module A (i.e. those listed under item 1 of the interface for Module A).

Step 2 – Consistency Checks

If matched goals are found as a result of Step 1, assess whether the context and solutions defined by Module B (i.e. those listed under items 2 and 3 of the declared interface for Module B) are consistent with the context and solutions defined by Module A (i.e. those listed under items 2 and 3 of the declared interface for Module A).

Step 3 – Handling Cross-References

- Where cross-references are made by Module A to Module B (i.e. Away Goal, Context and Solution references listed under items 5-7 of the declared interface for Module A) check that the entities referenced do indeed exist within Module B.
- Conversely, Where cross-references are made by Module B to Module A (i.e. Away Goal, Context and Solution references listed under items 5-7 of the declared interface for Module B) check that the entities referenced do indeed exist within Module A.

It may seem strange to include both steps 1a and 1b – i.e. admitting the possibility that whilst Module B supports Module A, Module A may also support Module B. However, circularity of 'SupportedBy' relationships between modules does not automatically imply circularity of argument (cross-references may be to separate legs of the argument within a module).

The defined context of one module may also conflict with the evidence presented in another. For example, implicit within a piece of evidence within one module may be the simplifying assumption of independence between two system elements. This assumption may be contradicted by the model of the system (clearly identifying dependency between these two system elements) defined as context in another module. There may also simply be a problem of consistency between the system models (defined in GSN as context) defined within multiple modules. For example, assuming a conventional system safety argument / software safety argument decomposition – as defined in Issue 2 of the U.K. Defence Standards 00-56 (MoD 1996) and 00-55 (MoD 1997) – the consistency between the state machine model of the software (which, in addition to modelling the internal state changes of the software will almost inevitably model the external – system – triggers to state changes) and the system level view of the external stimuli. As with checking the consistency of safety analyses, the problem of checking the consistency of multiple, diversely represented, models is also a significant challenge in its own right.

6 Safety Case Module 'Contracts'

Where a successful match (composition) can be made of two or more modules, a contract should be recorded of the agreed relationship between the modules. This contract aids in assessing whether the relationship continues to hold and the (combined) argument continues to be sustained if

at a later stage one of the argument modules is modified or a replacement module substituted. This is a commonplace approach in component based software engineering where contracts are drawn up of the services a software component *requires* of, and *provides* to, its peer components, e.g. as in Meyer's Smalltalk contracts (Meyer 1992) and contracts in object-oriented reuse (Helm 1990).

In software component contracts, if a component continues to fulfil its side of the contract with its peer components (regardless of internal component implementation detail or change) the overall system functionality is expected to be maintained. Similarly, contracts between safety case modules allow the overall argument to be sustained whilst the internal details of module arguments (including use of evidence) are changed or entirely substituted for alternative arguments provided that the guarantees of the module contract continue to be upheld.

A contract between safety case modules must record the participants of the contract and an account of the match achieved between the goals addressed by and required by each module. In addition the contract must record the collective context and evidence agreed as consistent between the participant modules. Finally, away goal context and solution references that have been resolved amongst the participants of the contract should be declared.

7 Principles of Safety Case Architecture Definition

In this paper safety case architecture is defined as the high level organisation of the safety case into modules of argument and the interdependencies that exist between them. In deciding upon the partitioning of the safety case, many of the same principles apply as for software architecture definition, for example:

- **High Cohesion / Low Coupling** – each safety case module should address a logically cohesive set of objectives and (to improve maintainability) should minimise the amount of cross-referencing to, and dependency on, other modules.
- **Supporting Work Division & Contractual Boundaries** – module boundaries should be defined to correspond with the division of labour and organisational / contractual boundaries such that interfaces and responsibilities are clearly identified and documented.
- **Supporting Future Expansion** – module boundaries should be drawn and interfaces described in order to define explicit 'connect' points for future additions to the overall safety case argument (e.g. additional safety arguments for added functionality).
- **Isolating Change** – arguments that are expected to change (e.g. when making anticipated additions to system functionality) should ideally be located in modules separate from those modules where change to the argument is less likely (e.g. safety arguments concerning operating system integrity).

The principal aim in attempting to adopt a modular safety case architecture for modular systems architecture is for the modular structure of the safety case to correspond as far as is possible with the modular partitioning of the hardware and software of the actual system. Arguments of functional (application) safety would ideally be contained in modules separate from those for the underlying infrastructure (e.g. for specific processing nodes of the architecture). Additionally, cross-references from application arguments to claims regarding the underlying infrastructure need to be expressed in non-vendor (non-solution) specific terms as far as is possible. For example, part of the argument with the safety case module for an application may depend upon the provision of a specific property (e.g. memory partitioning) by the underlying infrastructure. It is desirable that the cross-reference is made to the claim of the property being *achieved* rather than *how* the property has been achieved. In line with the principles of module interfaces and contracts as defined in the previous two sections, this allows alternative solutions to achieving this property to be substituted without undermining the application level argument. From this example, it is possible to see that in addition to thoughtful division of the safety case into modules, care must be taken as to the nature of the cross-references made between modules.

8 Patterns in Safety Case Architecture?

Well-understood architectural patterns in software architecture (such as the use of indirection and abstraction layers) can be seen to have immediate analogues in safety case architecture. Figure 6 illustrates this point with a simple three-tier 'layered' safety case architecture. The top tier (the Top System Level Argument module) sets out objectives in a form (e.g. Defence Standard 00-55 (MoD 1997) System Integrity Level requirements) that cannot immediately be satisfied by the objectives supported (e.g. Civil Aerospace Guidance DO178B (RTCA 1992) Development Assurance Level claims) in the bottom tier (the Software Safety Argument module). To solve this problem, an indirection layer (the DAL to SIL Mapping Argument module) is inserted between the top and bottom tiers. This module makes the read-across argument from the DAL regime to the SIL regime. (If sufficiently well defined, such a read-across argument may be usefully reused in future safety cases).

Figure 7 illustrates the possible internal structure of the read-across argument contained within the middle tier of the safety case architecture shown in Figure 6. The published goal of the read-across argument is the claim expressed in the form required by the target application context (i.e. in this case in terms of a Defence Standard 00-55 SIL Claim). This claim is then decomposed into the specific claims regarding the key process and product requirements required (according to 00-55) in order to satisfy the SIL requirement (e.g. testing claims, claims regarding coding standards, language choice etc.). At the bottom of the argument shown in Figure 7 is an (undeveloped) goal regarding compliance to a Development Assurance Level that is known to be supportable from the available evidence (i.e. from the bottom tier of the architecture shown in Figure 6).

Working bottom-up, the read-across argument then infers that in order to support this DAL claim the individual requirements dictated by DO178B for this DAL must also have been satisfied. The argument therefore draws out (above the DAL claim) these individual implicated sub-claims. The challenge in creating the read-across argument now lies in relating the specific claims required in order to support the SIL claim to the specific claims required in order to support the DAL claim. This approach attempts to read-across from one claim to another by deconstructing each claim in to its constituent parts and then relating these parts. It should be noted that the interrelation of SIL and DAL subclaims depicted in Figure 7 (where claims of one type are shown to be directly supportable by claims of the other) is a simplification. In reality, more complex chains of argument should be expected between the claims of each type.

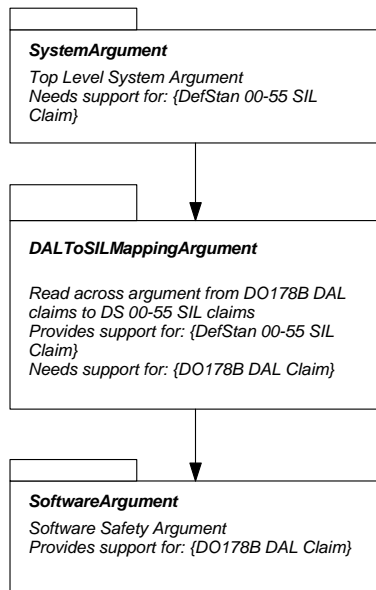


Figure 6 – Safety Case Architecture employing an abstraction layer

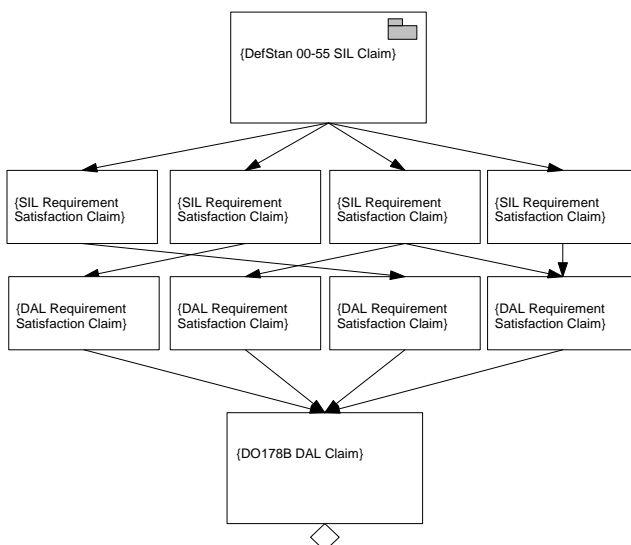


Figure 7 – Illustration of argument structure of DAL to SIL 'Abstraction' Layer Module

9 Managing Changes to a Modular Safety Case

Maintainability is one of the principle objectives in attempting to partition a safety case into separate modules. When change occurs that impacts traditional safety cases (defined as total entities for a specific configuration of system elements) reassessment of the 'whole' case is often necessary in order to have confidence in a continuing argument of safety. In such situations it will often be the case that for certain forms of change large parts of the safety required no reassessment. However, without having formally partitioned these parts of the case behind well-defined interfaces and guarantees defined by contracts it is difficult to justify non re-examination of their arguments.

When changes occur that impact a modular safety case it is desirable that these changes can be isolated (as far as is possible) to a specific set of modules whilst leaving others undisturbed. The definition of interfaces and the agreement of contracts mean that the impact path of change can be halted at these boundaries (providing interfaces are sustained and contracts continue to be upheld).

In extremis for a modular system it is desirable that when entire modules of the system are replaced, applications removed or added, or when the hardware of part of the system is substituted for that of a different vendor correspondingly entire modules of the safety case can be removed and replaced for those that continue to sustain the same safety properties. However, in order to achieve this flexibility, the following considerations need to be made for both the definition of context and the nature of cross-references made between modules:

- **Avoid unnecessary restriction of context** – It was highlighted in section 5 that the significant 'side-condition' of composing two or more modules together is that their collective context must be consistent. Often, the more specialised or restricted context is defined the harder it becomes to satisfy this condition (through incompatibility between defined contexts being more likely). For example, one module of the safety case may assume for the purposes of its argument that the temperature operating range is 10-20°C (i.e. the safety argument holds assuming the operating temperature is no less than 10°C and not greater than 20°C) whilst another modules may assume that the operating temperature is 20-30°C. Both ranges would form part of the defined context for each module and would create an inconsistency upon composition of the modules.
- There will be specific occasions when it is necessary to restrict the assumed context of an module in order for the module argument argument to hold. However, narrowing of context should be avoided as far as is possible. An analogy can be made with the operating range of a conventional mains power adaptor. If the adaptor is qualified over the entire operating range 110-250 volts then it may be used in wider number of situations (e.g. for both 110-120V main supply and 230-240V mains supply). If the adaptor is qualified to

a narrower operating range then obviously its scope of applicability is more restricted.

- **Goals to be supported within modules should state limits rather than objectives** – Borrowing terminology from the ALARP (As Low as Reasonably Practicable) framework (HSE 2001), ‘limits’ refer to the boundary between tolerable and intolerable risks, whilst ‘objectives’ refer to the boundary between tolerable and negligible risks. In order to permit the widest range of possible solutions of combinations with other modules, unsupported goals within a module (i.e. goals that will have to be supported through composition of this module with another) should define acceptability criteria rather than ‘desirability criteria’. (More informally, this means stating “what you will accept” vs. “what you want”). It is easier for another module to exceed (i.e. improve upon) a limit than it is to fail to meet an objective that was too harshly defined. Wherever possible boundary goals should ideally communicate both of limit and objective aspects of any requirement (by means of defining clearly the acceptance context of any undeveloped goal).
- **Goals to be supported within modules and ‘Away’ Goals should refer to ‘ends’ rather than ‘means’** – This issue has already been briefly discussed in section 7. In a similar vein to the previous observation, if goals on the boundary of modules or cross-references to goals between modules refer to claims regarding outcomes (e.g. a claim of memory partitioning) rather than means of achieving these outcomes (e.g. the specific mechanisms that ensure memory partitioning) then this leaves flexibility as to how solutions (supporting arguments) are provided – i.e. many possible alternative argument modules may be composed with this reference rather than just one specific form of argument.

A true assessment of the modifiability of any proposed safety case architecture can only be achieved through consideration of realistic change scenarios and examination of their impact on the module structure of the architecture. This form of evaluation is discussed further in the following section.

10 Safety Case Architecture Evaluation

In the discipline of software architecture early lifecycle assessment of any proposed architecture is encouraged to gain an appreciation of how well the architecture supports required architectural quality attributes such as scalability, performance, extensibility and modifiability. To assess software architectures (particularly with regard to modifiability) a scenario based evaluation technique – SAAM (Software Architecture Analysis Method) (Kazman et al. 1996) – has been developed by Kazman et al. The activities performed in a SAAM assessment are discussed briefly below:

1. **Develop Scenarios** – Definition of scenarios that illustrate activities and changes that the architecture should ideally accommodate.
2. **Describe candidate architecture** – Definition of the candidate architecture or architectures in a suitably expressive architectural description language (ADL) that can be easily understood by all parties involved in the analysis.
3. **Classify Scenarios** – Classification of scenarios into the two categories of *direct* and *indirect* scenarios. Direct scenarios are those scenarios that an architecture is expected to accommodate without change. Indirect scenarios describe situations where change to elements within the architecture is anticipated.
4. **Perform Scenario Evaluations** – For each indirect scenario, identification of the changes to the architecture that are necessary for it to support the scenario, together with an estimation of the effort required to make these changes. For each direct scenario, a walkthrough should be conducted that shows clearly how the scenario is accommodated by the architecture.
5. **Reveal Scenario Interactions** – Identification of where two or more indirect scenarios involve change to the same element of the architecture. The interaction of semantically unrelated scenarios can indicate a lack of cohesion in how architectural elements are defined.
6. **Overall Evaluation** – Based upon the results of all the scenarios analysed, evaluation of whether the proposed architecture adequately supports the required quality attributes.

With little modification, this method of architecture evaluation can be read-across to the domain of safety case architecture. One of the overriding aims in defining a modular safety case architecture is improve maintainability and (as a subtype of maintainability) extensibility. However, it is difficult to determine a priori whether a proposed safety case architecture (such as that presented in section 11) will be maintainable. Adopting a similar approach to SAAM but for safety case architectures would suggest that a number of change ‘scenarios’ should be identified. These scenarios should attempt to anticipate all credible changes that could impact the safety case over its lifetime (e.g. a change of hardware manufacturer, addition of functionality). For each of these change scenarios (NB – by definition these scenarios would be classified as *indirect* in the SAAM methodology), a walkthrough should be conducted to assess the likely impact of the change upon the individual modules of the proposed safety case architecture.

In the SAAM method, the effects of indirect scenarios are classified according to the following three classes of change:

- **Local Change** – change isolated within a single module of the architecture.
- **Non-Local Change** – change forced to a number of modules within the architecture.
- **Architectural Change** – widespread change forced to a large proportion of modules within the architecture.

These ideas can also be usefully applied to the safety case architecture domain. Ideally, for a modular safety case partitioned and carefully cross referenced in accordance with the principles stated in this paper the effects of all credible scenarios would fall within the first of the categories listed above. To illustrate how the categories of change read-across to the concept of a modular safety case architecture consider a simple safety case architecture as shown in Figure 8 containing the following four modules:

SysArg	Safety case module containing the top level safety arguments for the overall system identifying top level claims for each application run as part of the system and a top level claim regarding the safety of the interactions between applications.
AppAArg	Safety case module containing the arguments of safety for Application A.
AppBArg	Safety case module containing the arguments of safety for Application B.
InteractionArg	Safety case module containing the arguments of safety for the interactions between Applications A and B.

The 'SysArg' module is supported by the 'AppAArg', 'AppBArg' and 'InteractionArg' modules. The 'AppAArg' module relies upon guarantees of safe interaction with Application B as defined by the claims contained within the 'InteractionArg' module (hence 'AppAArg' is shown making a contextual reference to 'InteractionArg'). Similarly, the safety argument for Application B ('AppBArg') relies upon guarantees of safe interaction with Application A as defined in the 'InteractionArg' module.

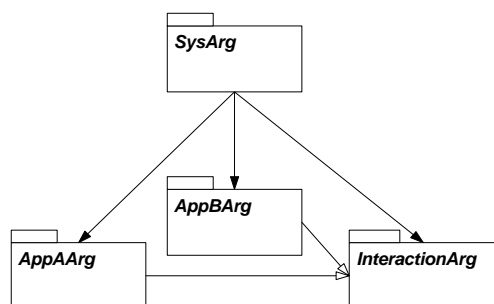


Figure 8 – A Simple Safety Case Architecture

The following are three possible change scenarios that could have an impact on the outlined safety case architecture

Scenario #1 Application A is rewritten (perhaps including some additional functionality) but still preserves the safety obligations as defined in the contract between AppAArg, SysArg and InteractionArg.

Scenario #2 Application A is rewritten and interacts with Application B differently from before.

Scenario #3 Change is made to the system memory management model that enables new means of possible (unintentional) interaction between applications.

The effect of scenario #1 would be that the safety argument for Application A ('AppAArg') would need revision to reflect the new implementation. However, provided that the safety obligations of the module to the other modules (as defined by the contracts between the module safety case interfaces) continue to be upheld no further change to other modules would be necessary. Figure 9 depicts the effects of this scenario (a cross over a module indicates that the module is 'challenged' by the change and revision is necessary). The effects of this scenario could be regarded as a *local* change.

The effect of scenario #2 would be that not only must the safety argument for Application A ('AppAArg') be revised but in addition the safety argument for the interaction between modules (contained in 'InteractionArg') would need to be reexamined in light of the altered interaction between applications A and B. If, however, the revised 'InteractionArg' could continue to support the same assurances to the Application B argument of the safety of interactions with Application A then the Application B safety arguments (contained in 'AppBArg') would be unaffected. Figure 10 depicts the effects of this scenario. The effects of this scenario could be regarded as a *non-local* change (owing to the fact that the change impact has spread across a number of modules).

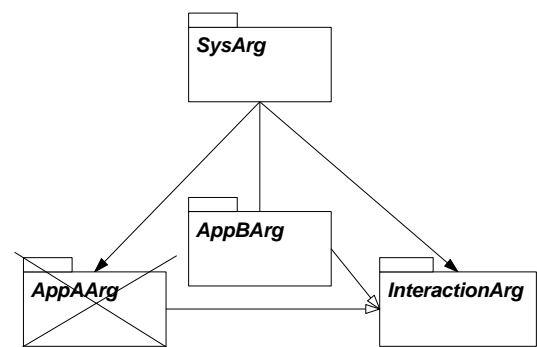


Figure 9 – Illustration of Local Change

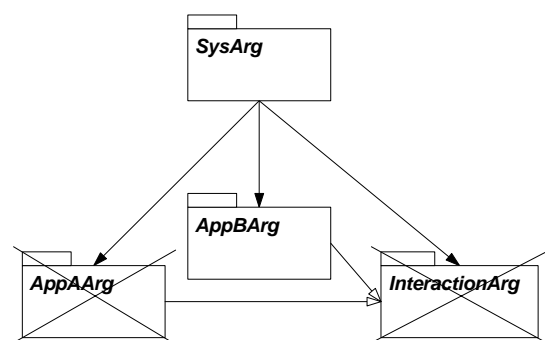


Figure 10 – Illustration of Non-Local Change

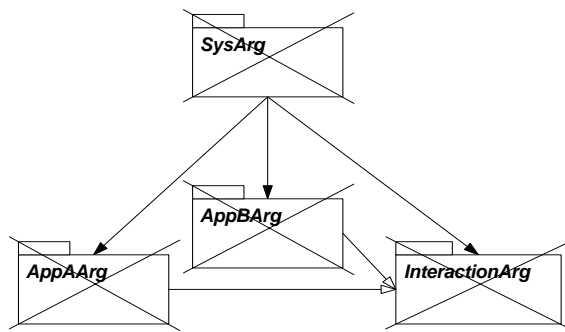


Figure 11 – Illustration of Architectural Change

The effect of scenario #3 is that it changes the nature of possible interactions between all applications. As such, the safety argument for the interaction between modules (contained in ‘InteractionArg’) would obviously need to be revised. It is likely that the nature of the assurances given by interaction argument to the safety arguments for applications A and B (as defined by the contracts between ‘InteractionArg’ and ‘AppAArg’, and between ‘InteractionArg’ and ‘AppBArg’) could be altered. Consequently both of these modules could be impacted. The change to the memory management model may even such that it alters the nature of the top level claim that needs to be made in the ‘SysArg’ module regarding the safety of application interactions (i.e. the ‘SysArg’ module may also be affected. Figure 11 depicts the effects of this scenario. The effects of this scenario could be regarded as *architectural* (owing to the fact that the change can potentially impact many modules). This is perhaps to be expected as this scenario describes modifying a fundamental services provided as part of the system infrastructure.

11 Example Modular Safety Case Architecture for IMA

The principles of modularising and evaluating safety case architecture have been applied in reworking a “Generic Avionics Safety Argument” developed by Pygott (Pygott 1999). The resultant safety case architecture is shown in Figure 13. (Note – for clarity not all of the dependencies between modules have been shown on this diagram). The role of each of the modules of the safety case architecture shown in Figure 13 is as follows:

AppInAArg: Specific argument for the safety of Application A (one required for each application within the configuration)

CompilationArg: Argument of the correctness of the compilation process. Ideally established once-for-all.

HardwareArg: Argument for the correct execution of software on target hardware. Ideally abstract argument

established once-for-all leading to support from SpecificHardwareArg modules for particular hardware choices.

ResourcingArg: Overall argument concerning the sufficiency of access to, and integrity of, resources (including time, memory, and communications)

ApplnInteractionArg: Argument addressing the interactions between applications, split into two legs: one concerning intentional interactions, the second concerning non-intentional interactions (leading to the NonInterfArg Module)

InteractionIntArg: Argument addressing the integrity of mechanism used for intentional interaction between applications. Supporting module for ApplnInteractionArg. Ideally defined once-for-all.

NonInterfArg: Argument addressing non-intentional interactions (e.g. corruption of shared memory) between applications. Supporting module for ApplnInteractionArg. Ideally defined once-for-all

PlatFaultMgtArg: Argument concerning the platform fault management strategy (e.g. addressing the general mechanisms of detecting value and timing faults, locking out faulty resources). Ideally established once-for-all. (NB Platform fault management can be augmented by additional management at the application level).

ModeChangeArg: Argument concerning the ability of the platform to dynamically reconfigure applications (e.g. move application from one processing unit to another) either due to a mode change or as requested as part of the platform fault management strategy. This argument will address state preservation and recovery.

SpecificConfigArg: Module arguing the safety of the specific configuration of applications running on the platform. Module supported by once-for-all argument concerning the safety of configuration rules and specific modules addressing application safety.

TopLevelArg: The top level (once-for-all) argument of the safety of the platform (in any of its possible configurations) that defines the top level safety case architecture (use of other modules as defined above).

ConfigurationRulesArg: Module arguing the safety of a defined set of rules governing the possible combinations and configurations of applications on the platform. Ideally defined once-for-all.

TransientArg: Module arguing the safety of the platform during transient phases (e.g. start-up and shut-down). Ideally generic arguments should be defined once-for-all that can then be augmented with arguments specifically addressing transient behaviour of applications.

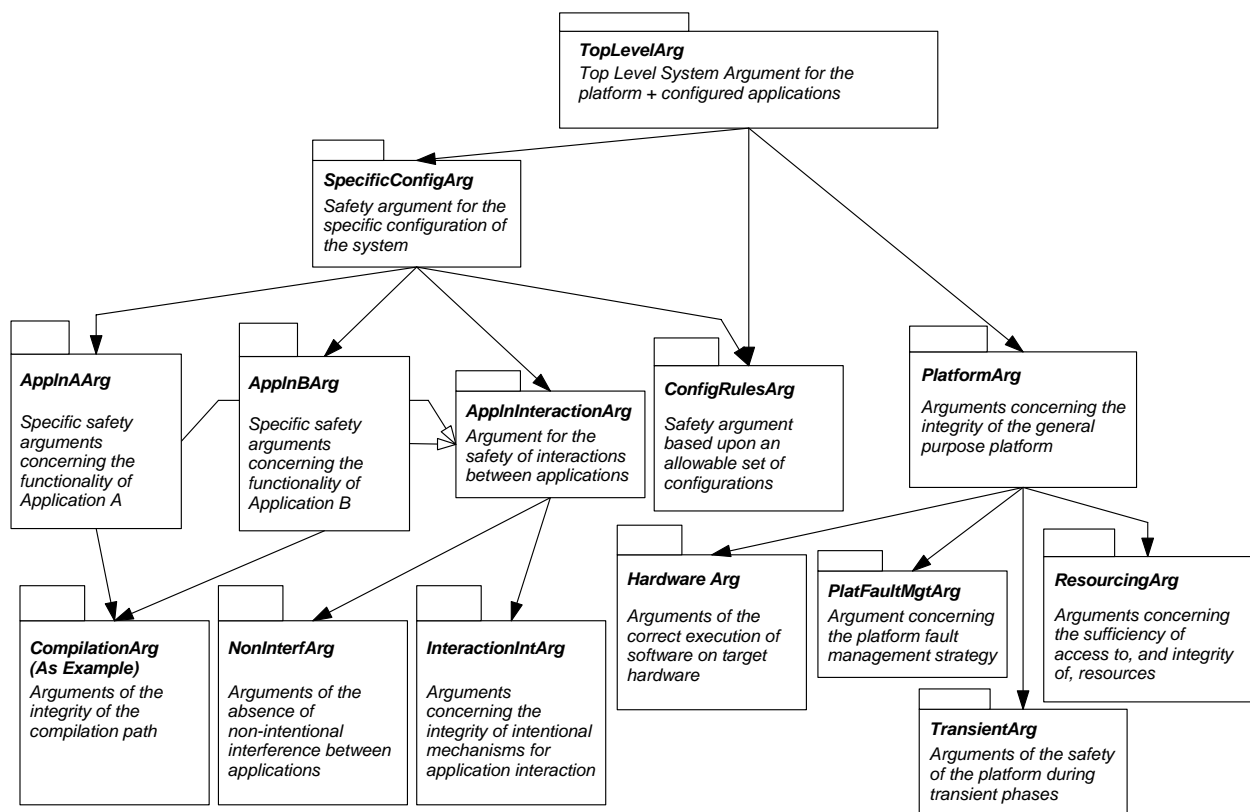


Figure 13 – Safety Case Architecture of Modularised IMA Safety Argument

An important distinction is drawn between those arguments that ideally can be established as ‘once-for-all’ arguments that hold regardless of the specific applications placed on the architecture (and should therefore be unaffected by application change) and those that are configuration dependent. Examples of application configuration specific modules include the ‘ApplnAArg’, ‘ApplnBArg’ and ‘ApplnInteractionsArg’ modules. Examples of the argument models established ‘once for all’ include the ‘NonInterfArg’ and ‘InteractionIntArg’ modules. Table 1 summarises the modules falling under each category.

Argument Modules Established ‘Once-for-all’	Configuration Dependent Argument Modules
ApplnAArg CompilationArg HardwareArg ResourcingArg InteractionIntArg NonInterfArg PlatFaultMgtArg ModeChangeArg TopLevelArg ConfigurationRulesArg TransientArg	ApplnAArg ApplnInteractionArg SpecificConfigArg

Table 1 – Categorisation of Safety Case Architecture Modules

In the same way as there is an infrastructure and backbone to the IMA system itself the safety case modules that are established once for all possible application configurations form the infrastructure of this particular safety case architecture. These modules (e.g. NonInterfArg) establish core safety claims such as non-interference between applications by appeal to properties of the underlying system infrastructures. These properties can then be relied upon by the application level arguments.

12 Minimising the Impact of Change

The intention of the partitioning of the safety case, such as in the approach described in the example above, is to maximise the number of arguments (modules) that are stable in the presence of change. As explained in section 10 in order to evaluate the success of the argument in this respect it is necessary to identify a number of credible change scenarios for the IMA-based system. Credible scenarios could include:

- Hardware Vendor Change
- Addition of a single application
- Removal of a single application
- Modification of existing application
- Addition of extra processing nodes
- Remove of processing nodes
- Change of Databus

Some of these scenarios may be accommodated easily by the proposed safety case architecture. For example, if the applications in a configuration change, although individual application arguments (e.g. “ApplnAArg” module) and application interaction arguments (i.e. those within the “ApplnInteractionArg” module) must be updated, argument of interaction integrity and non-interference (the “InteractionIntArg” and “NonInterfArg” modules) may well be able to stay unaltered. Other scenarios, such as change of hardware vendor may have a wider impact across the modules of the safety case (e.g. impacting compilation arguments as well as the more obvious hardware arguments).

For the architecture proposed, Table 2 provides illustrative examples of the modules affected by credible change scenarios.

Change Scenario	Impact on Safety Case Modules
Application A modified	ApplnAArg must be updated All other (13) modules unaffected provided that interface of ApplnAArg is preserved
Application C added	ApplnCArg must be established SpecificConfigArg must be updated ApplnInteractionArg must be updated All other (12) modules unaffected provided that interface of ApplnInteractionArg is preserved
Introduction of new hardware type	HardwareArg and other arguments that specifically address the hardware of the system (such as InteractionIntArg) must be updated. All other modules unaffected provided that the interface of the updated modules can be preserved (i.e. the same ‘guarantees’ can be made for the properties of the new hardware as for the old).

Table 2 - Example Change Scenario Impact Summaries

It is through the ability to leave many of the modules of the safety case undisturbed in the presence of change (as illustrated in Table 2) that the benefits of IMA can be carried through to the certification process.

13 Reasoning about Partitioning and Independence

One of the main impediments to reasoning separately about individual applications running on an IMA based architecture separately is the degree to which applications interact or interfere with one another. DO178B (RTCA 1992), in discussing partitioning between software elements developed to differing Development Assurance Levels identifies that there are a number of possible routes through which interference is possible:

- **Hardware Resources** – processors, memory, Input Output devices, timers etc.
- **Control Coupling** – vulnerability to external access
- **Data Coupling** – shared data, including processor stacks and registers.
- **Hardware Failure Modes**

For example, partitioning must be provided to ensure that one process cannot overwrite the memory space of another process. Similarly, a process should not be unintentionally allowed to overrun its allotted schedule such that it deprives another process of processor time.

The European railway safety standard EN 50129 (CENELEC 2001) makes an interesting distinction between those interactions between system components that are intentional (e.g. component X is meant to communicate with component Y) are those that are unintentional (e.g. the impact of electromagnetic interference generated by one component on another).

Unintentional interactions are typically the result of an error (whether random or systematic). For example, the unintentional interaction of one process overwriting the memory space of another is a fault condition. A further observation made in EN 50129 is that there are a class of interactions that are non-intentional but created through intentional connections. An example of this form of interaction is the influence of a failed processing node that is ‘babbling’ and interfering with another node through the intentional connection of a shared databus.

The safety case architecture promotes (in the “NonInterfArg” module) the ideal that ‘once-for-all’ arguments are established by appeal to the properties of the IMA infrastructure to address unintentional interactions. For example, a “non interference through shared memory space” argument could be established by appeal to the segregation offered by a Memory Management Unit (MMU). An argument of “non-interference through shared scheduler” could be established by appeal to the priority-based scheduling scheme offered by the scheduler. Although the particular forms of interference between applications will need to be drawn out (within the “ApplnInteractionArg” module) it is expected that these *specific* arguments can be addressed through the *general* infrastructure arguments provided by the “NonInterfArg” module.

It is not possible to provide “once-for-all” arguments for the intentional interactions between components – as these can only be determined for a given configuration of components. However, it is desirable to separate those arguments addressing the logical intent of the interaction from those addressing the integrity of the *medium* of interaction. For example, if application A passes a data value to application B across a data bus it would be desirable to partition those arguments that address the possibility of A sending to wrong value to B from the arguments that address the possible corruption of the data value on the data bus. Both issues must be clearly identified and reasoned about (within the

“ApplnInteractionArg” module). However, the supporting arguments concerning the integrity of the medium of interaction can be established “once-for-all” within the “InteractionIntArg” module.

14 Implications for Certification Processes

A modular approach to safety case construction has implications for the acceptance process. Whereas, traditionally certification has involved accepting, at a single point in time, a single monolithic safety case for an entire system for the benefits of a modular safety case approach to be realised requires a certification process that acknowledges the structure of a partitioned safety case that can be extended and modified without instantly requiring re-evaluation of the entire case. The guidance document ARINC 651 (ARINC 1991) recognises this fact for suggests that for IMA-based systems the certification tasks are comprised of the following three distinct efforts:

- Confirmation of the general environment provided by the cabinet
- Confirmation of the operational behaviour of each function (application) intended to reside within a cabinet
- Confirmation of the resultant composite of the functions

ARINC 651 also recognises that conventional safety standards (such as DO178B (RTCA 1992)) may need to be updated to reflect these new distinct tasks. These observations can be clearly related to the example IMA safety case architecture presented within this paper. Confirmation of the “general environment” involves qualification of both the hardware and software infrastructure (e.g. operating system) and relates to those modules shown within the proposed architecture that should ideally be established once for all possible application configurations (e.g. the ‘HardwareArg’ module). Confirmation of the operational behaviour of each function relates to the specific application argument modules (e.g. the ‘ApplnAArg’ module) shown within the proposed architecture. Confirmation of the composite operation of functions relates to those arguments, specific to a configuration of applications, that address the interaction of applications (e.g. the ‘ApplnInteractionArg’ and ‘SpecificConfigArg’ modules).

ARINC 651 talks explicitly of the need for “building block qualification” whereby it is possible to “separately qualify certain building blocks of an IMA architecture in order to reduce the certification effort required for any particular IMA-hosted function”. Example building blocks listed include specific arguments relating to the (ARINC 629) global data bus, the ARINC 659 backplane bus, the robust partitioning environment and the cabinet hardware / software environment. Again, it is easy to see a correspondance with the IMA safety case architecture proposed within this paper (e.g. the ‘NonInterfArg’ module addressing robust partitioning and the ‘InteractionIntArg’ module addressing the integrity of bus communication). However, no detail regarding how these

building block arguments are to be represented and managed is presented within ARINC 651.

In order to design and validate the various building blocks involved in IMA, ARINC 651 identifies the need for “rules which govern how the building blocks work together”. It additionally describes that, “a feature of these rules of application is that they can be used to limit the work associated with certifying and re-certifying an IMA function to proof of compliance with the rules, and qualification of the function itself. Regulatory agency discussion is encouraged to establish how certification credit may be granted for adherence to these rules”. This concept of defining rules between building blocks relates strongly to the principles of establishing well-defined module interfaces and contracts between safety case modules put forward within this paper. As the quote above clearly highlights, a necessary part of a new certification process based upon modular safety cases is to clearly give credit (i.e. limit the required re-certification) where contracts between safety case modules are upheld in the light of change to, or reconfiguration of, modules within the overall safety case.

15 Summary

In order to reap the potential benefits of modular construction of safety critical and safety related systems (e.g. ease of later addition or replacement of functionality, or through-life flexibility of hardware vendors) a modular approach to safety case construction and acceptance is also required. This paper has explained some of the key concepts and principles of a modular safety case approach, including safety case module interface definition, cross-referencing between safety case modules and the steps involved in composition of one or more safety case modules. Specifically, the paper has described how the Goal Structuring Notation (GSN) may be extended to include and support these concepts. Use of these extensions has been illustrated by means of an example modular safety case architecture for IMA-based systems.

This paper has attempted to illustrate how concepts established in the field of software architecture – such as design-by-contract, scenario-based evaluation and architectural patterns – can be seen to have obvious analogues in the safety case architecture domain.

16 Acknowledgements

The author would like to acknowledge the financial support given by QinetiQ for the work reported in this paper.

17 References

- Kelly, T. P. (1997) *A Six-Step Method for the Development of Goal Structures*, York Software Engineering.
- Jones, C. (1983) Specification and design (parallel) programs, *Proc. IFIP Information Processing 83*, 1983.
- Hofmeister, C., Nord, R., Soni, D. (1999) *Applied Software Architecture*, Addison-Wesley

- MoD (1996) *Defence Standard 00-56 Safety Management Requirements for Defence Systems*, U.K. Ministry of Defence
- MoD (1997) *Defence Standard 00-55 Requirements of Safety Related Software in Defence Equipment*, U.K. Ministry of Defence
- Meyer, B. (1992) Applying Design by Contract, *Computer*, **25**:40-52, IEEE Press
- Helm, R., Holland, I. M., Gangopadhyay, D. (1990) Contracts: Specifying Behavioural Compositions in Object-Oriented Systems, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 169-180, October 1990
- RTCA (1992), *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Washington D.C., December 1 1992.
- HSE (2001), *Reducing Risk, Protecting People – HSE's Decision Making Process*, HSE Books
- Kazman, R., Abowd, G., Bass, L., Clements, P. (1996) Scenario-based analysis of software architecture, *Software*, **13**:47-55, IEEE Press
- CENELEC (2001), *EN 50159:2001; Railway Applications - Communication, signalling and processing systems - Safety-related communication in closed transmission systems*, CENELEC
- ARINC (1991) *Design Guidance for Integrated Modular Avionics*, Aeronautical Radio, Inc., Annapolis, Maryland Report 651, November 9
- Pygott, C. (1999) *Certification Analysis Techniques for an IMA Architecture*, Internal DERA report DERA/CIS/CIS3/CR990865, December 1999

Formal Modelling and Analysis of Mission-Critical Software in Military Avionics Systems

Zahid H. Qureshi

Systems Engineering and Evaluation Centre
University of South Australia
Mawson Lakes Campus, Mawson Lakes 5095, South Australia
zahid.qureshi@unisa.edu.au

Abstract

A typical avionics mission system of a military aircraft is a complex real-time system consisting of a mission control computer, different kinds of sensors, navigation and communication subsystems, and various displays and stores; all interconnected by a number of serial data buses. The mission capability is increasingly implemented in the mission-critical software and the robustness of this software is vital for mission success. The complexity and real-time requirements of mission systems represent major challenges to the Australian Defence Force during new acquisitions, upgrades and maintenance. This paper describes the experiences on a joint research project between the University of South Australia and Australia's Defence Science and Technology Organisation into the modelling and analysis of avionics mission systems. The paper provides a summary of the key aspects of our previous research work on the modelling of a generic mission system using Coloured Petri Nets and the analysis of task scheduling on the mission computer. Finally, the paper briefly discusses the extension of the generic model to obtain a formal model of the mission system of the AP-3C Orion maritime surveillance aircraft.

Keywords: Avionics mission systems, formal methods, mission-critical software.

1 Introduction

The complexity of military avionics mission systems is continually increasing to meet the requirements of missions and changing operational environment. The mission capability is increasingly implemented in the mission-critical software and the robustness of this software is vital for mission success. The Australian Defence Force has experienced problems in the acquisition, upgrades and through-life support of airborne electronic mission systems, leading to cost and schedule overruns (CoA 2001). Major problems concern the integration of a large number of relatively different components and subsystems, such as radar, electronic support measures, navigation, communication and

mission data processing, and that of achieving an overall optimised and operationally effective mission system.

The probable cause of the loss of the Mars Polar Lander has been traced to premature shutdown of the descent engines, resulting from a vulnerability of the software to transient signals (CIT 2000). The F/A-22 avionics have failed or shut down during numerous tests of the aircraft due to software problems. The shutdown occurs when the pilot attempts to use the radar, communication, navigation, identification, and electronic warfare systems concurrently (GAO 2003a); this has led to delays in the avionics software development and flight-testing, and to an increase in avionics development costs by over US\$80 million. One of the major challenges in the F-35 Joint Strike Fighter (JSF) program includes the integration of highly advanced sensors with the avionics systems. This has contributed to the increase in the 1996 estimated cost and schedule for the JSF development phase by 56 percent and 40 per cent respectively (GAO 2003b), and an increase of an additional US\$10.3 billion since the start of the system development and demonstration phase (GAO 2004).

The key issue for the Australian Defence Force is to reduce the cost of procurement and upgrades of avionics mission systems in a way that provides sufficient assurance of the system architecture and the behaviour and performance of the software-intensive mission-critical system to meet the operational requirements of the aircraft. This has motivated research into the development of a framework for analysing mission system functionality and upgrade scenarios, and validating overall system performance for future procurements and upgrades.

The aim of this paper is to describe the experiences on a joint research project between the University of South Australia and Australia's Defence Science and Technology Organisation into the modelling and analysis of avionics mission systems, conducted over a 3 year period; key aspects of this research work have been published in the public domain (Kristensen et al. 2001, Kristensen et al. 2002, Petrucci et al. 2002, Petrucci et al. 2003).

In the following section, we describe the generic mission system architecture, and in Section 3 we provide an overview of formal methods for system modelling and introduce the basic concepts of Coloured Petri Nets. In Section 4, we describe the development of a modelling framework for a generic mission system using Coloured Petri Nets, and in Section 5 we present our analysis

approach which shows how state space methods can be used in the framework of Coloured Petri Nets to reason about system properties. In Section 6, we describe how the modelling framework was used to obtain a Coloured Petri Net model for the mission system of the AP-3C Orion maritime surveillance aircraft. Finally, in Section 7 we provide conclusions on the modelling approach and discuss ideas for future work.

2 Avionics Mission Systems

The mission system for a typical combat aircraft, such as an F/A-18, is composed of many discrete avionics subsystems including radar, navigation, and mission computers. Each of these subsystems may contain further subsystems and components. Since the mission system is a very complex collection of subsystems and components, we shall employ abstraction as a technique for managing the complexity to enable the construction of models at higher levels. The generic architecture of an avionics mission system (AMS) for a combat aircraft (Locke et al. 1990) is shown in Figure 1. The AMS consists of a number of subsystems connected via a serial data bus (SDB). The control of devices, displays, and pilot controls is handled by a collection of software tasks executing on the mission control computer (MCC) which also acts as the SDB controller. The subsystems communicate by the exchange of data/messages across the SDB.

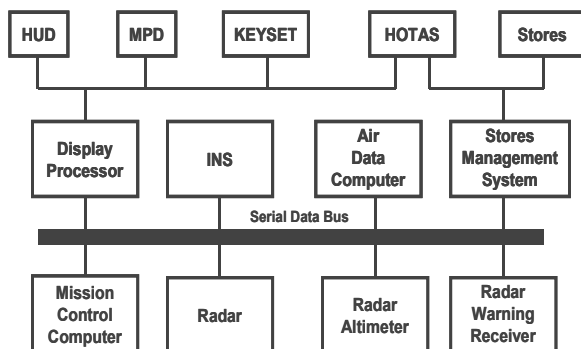


Figure 1: Generic Mission System Architecture

The controls and displays of the AMS consist of the head-up display (HUD), the multi-purpose display (MPD), the crew keyset (KEYSET), and the hands-on throttle and stick (HOTAS). These components form the human-machine interface of the AMS. The human-machine interface is controlled by the Display Process subsystem. The sensors of the AMS consist of the Air Data Computer (ADC), the Radar, the Inertial Navigation System (INS), the Radar Altimeter (RALT), and the Radar Warning Receiver (RWR). The stores contain a number of weapons such as missiles and bombs and are controlled by the Stores Management Subsystem.

Typical tasks performed by a mission control computer system may include data collection from various sensors, fusion of collected data, display of information to pilots, and controlling devices in response to inputs from the aircraft crew. One of the critical aspects of the proper functional and performance behaviour of the mission system is that the tasks must be scheduled in a way that guarantees that hard deadlines are met under all

circumstances. This real-time requirement is critical to the operational performance of the mission system and hence to the success of a particular mission. Thus major concerns, when upgrading and maintaining mission systems, are the scheduling of tasks and the impact of delays associated with data transfer across the bus connecting the mission control computer and the various devices.

The key issues discussed in this paper are the scheduling of tasks on the mission control computer, and the data transfer across the data buses connecting the mission control computer and the various avionics subsystems. A typical application software task scheduling mechanism, such as for the F/A-18 and F-111 aircraft, is based on a cyclic executive (Rockwell 1992). The cyclic executive executes an application that is divided into a sequence of non pre-emptive tasks, invoking each task in a pre-determined order throughout the execution history of the application (Locke 1992). One can distinguish two types of tasks, namely, rategroup and background tasks. The rategroup tasks are periodic and have higher priority than the background tasks, which may be considered as aperiodic. The cyclic executive repeats its task list at a rate that is known as a major cycle. The major cycle is further divided into periods known as minor cycles. The major cycles have a set of tasks scheduled that must meet the required deadline in order to maintain the integrity of the mission system.

3 Formal Modelling and Analysis

3.1 Formal Methods

Formal methods are mathematically based techniques, often supported by reasoning tools, that can offer a rigorous and effective way to model, design and analyse computer systems (Bjorner and Druffel 1990). A formal method has a sound mathematical basis, typically given by a formal specification language. This basis provides the means of precisely defining notations like consistency and completeness and more relevantly, specification, implementation and correctness. It provides the means of proving that a specification is realisable, proving that a system has been implemented correctly, and proving properties of a system without necessarily running it to determine its behaviour. There are comprehensive accounts of experience on the use of formal methods in industry and research (e.g., Hinchey and Bowen 1995).

The design and validation of complex computer-based systems, for example, military avionics and space missions, should ensure the correctness of a design at the earliest stage possible. The performance of an avionics mission system is critical during flight, thereby making it a good candidate for more rigorous design and verification methods. Havelund and Lowry (2001) discuss an application of the model checker SPIN to formally analyse a software-based multithreaded plan execution module of a NASA space-craft control system. The formal verification effort had a major impact: locating errors that would probably not have been located otherwise and identifying a major design flaw.

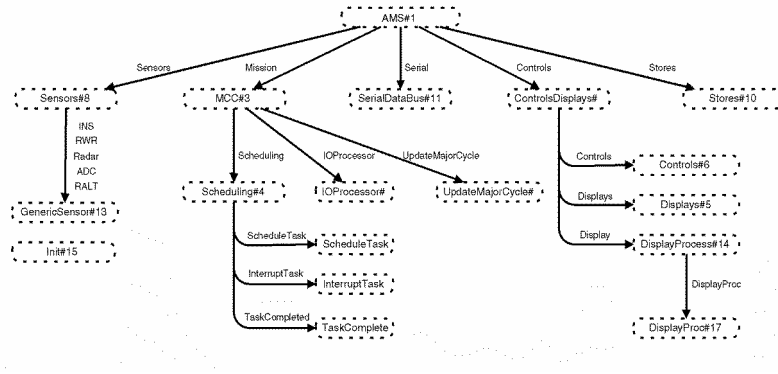


Figure 2. CPN model overview - Hierarchy page

3.2 Coloured Petri Nets

Coloured Petri Nets (Jensen 1997) are a graphically oriented modelling language for the design, specification, and verification of concurrent and distributed systems. CPNs are based on Petri Nets (Desel and Reisig 1998) and the functional programming language Standard ML (SML) (Ullman 1998). Petri Nets provide the primitives for modelling concurrency and synchronization, whereas SML provides the primitives for modelling data manipulation in systems and for creating compact and parameterisable CPN models. A CPN model of a system describes the states that the system may be in and the transitions between these states. CPN models are executable, which means that it is possible to investigate the behaviour of the system by simulations. CPN models can also be used for formal verification of systems based on state space analysis and model-checking (Jensen 1997). CPNs and the Design/CPN tool (Design/CPN Online) have been successfully applied in a wide range of application areas and many projects have been carried out in industry (Jensen 1997).

Scheduling of tasks in real time systems has traditionally been conducted using a purely algorithmic approach (Liu 2000). Recently, there has been an increasing interest in applying timed automata and model checking techniques to scheduling problems; the basic idea is to turn the scheduling problem into a reachability problem that can be solved by analysis tools using a state space search (Petrucchi et al. 2002). The advantages of formal modelling and state space methods in this setting is that the same model of the system can be used to analyse scheduling as well as other properties, such as functional correctness. Hence, it represents an integrated approach to the analysis of the system.

4 CPN Model of a Generic Mission System

In this section, we describe our modelling framework and approach by providing an overview of the CPN model of the generic mission system and give some representative examples of modelling at the different levels of abstraction in the CPN model; the reader is referred to Kristensen et al. (2001) for details.

4.1 Modelling Framework Overview

A CPN model can be structured into a number of hierarchically related modules (in CPN terminology called pages) with well-defined interfaces between them. The *hierarchy page* giving the overall structure of the CPN AMS model is depicted in Figure 2. Each node in Fig. 2 represents a page (module) of the CPN AMS model, and is named according to the page in the CPN model that it represents. An arc leading from one node to another node means that the latter is a subpage (submodule) of the former. The page *AMS* is the top-most page in the CPN model.

The CPN model consists of five main parts which correspond to the five immediate subpages of the *AMS* page: the *MCC* page and its subpages models the Mission Control Computer, the *Sensors* page and its subpages models the sensors, the *ControlsDisplays* page and its subpages models the man-machine interface, the *Stores* page models the stores and the Stores Management System, and the *SerialDataBus* page models the Serial Data Bus.

Figure 3 depicts the *AMS* page and provides the most abstract view of the model. This page consists of five *substitution transitions* (drawn as rectangles with an HS tag in the lower right corner) and two *places* (drawn as ellipses). The substitution transitions *Mission Control Computer*, *Sensors*, *ControlsandDisplays*, *StoresManagementSystem*, and *SerialDataBus* correspond to the five main parts of the AMS system. Each of the

sesubstitution transitions (and its surrounding places) is related to a *subpage*. The subpage of a substitution transition provides a more detailed description of the compound activity/component represented by a substitution transition. Place *SerialDataBus* is used to model the data transfer across the serial data bus. Place *CDS* represents the interface between the controls and the storage management system. Each of the places in Fig. 3 is so-called *socket places* used to link the subpage of the substitution transition and the *AMS* page. Socket places are assigned (linked) to *port places* on the subpage of the substitution transition.

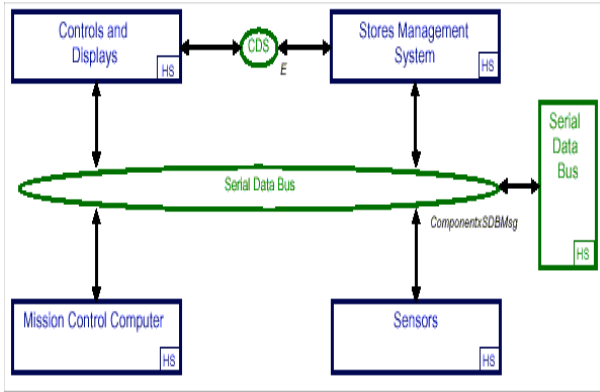


Figure 3: The AMS page

An addressing scheme has been developed to capture the AMS architecture, identify the components of the AMS, and to model the interaction between the components. Components can be hardware devices (e.g., sensors) as well as software processes/tasks (e.g., the tasks executed by the MCC). In addition to making it possible to identify components, the virtue of this addressing scheme is that new components can easily be added to the CPN model without having to make global modifications to it. This means that new tasks on the MCC as well as new hardware devices can be incorporated into the CPN

```

color MCCTaskName = with
  AircraftFlightData | Steering
  | RadarControl      | TargetDesignation
  | TargetTracking    | WeaponSelection
  | WeaponTrajectory  | WeaponRelease
  | HUDDisplay        | MPDHUDDisplay
  | MPDTacticalDisplay | MPDStoresDisplay
  | MPDStatusDisplay  | KEYSET_HOTAS
  | RWRControl        | RWRThreatControl
  | BuiltInTest ;

color BackgroundTask =
  record name : MCCTaskName *
        size : Int *
        left : Int ;

color RategroupTask =
  record name : MCCTaskName *
        rate : Int *
        next : Int *
        size : Int *
        left : Int ;

color MCCTask = union
  Background : BackgroundTask +
  Rategroup : RategroupTask ;

```

Figure 4: Colour set definition for tasks

model with only local modifications. A sample of the colour set definitions used to realise the addressing scheme are shown in Figure 4. The colour set definitions are written in the Standard ML programming language and are similar to type definition found in programming languages. The colour set *MCCTaskName* identifies the different tasks executing on the mission control computer. The colour set *BackgroundTask* contains the attributes:

- *name* - the name of the task;
- *size* - total size of the task when executed (measured in time units);
- *left* - how much of the task remains to be executed

In the colour set *RategroupTask*, the *rate* attribute specifies the frequency of a rategroup task (as the number of minor cycles between execution of the task), and the *next* attribute keeps track of the next minor cycle in which the task should be executed.

4.2 Mission Control Computer

Figure 5 depicts the MCC page which is the most abstract part modelling the mission control computer. The MCC is the subpage associated with the substitution transition Mission Control Computer from Fig. 3. The page has two substitution transitions: *Scheduling* represents the scheduling mechanism on the mission control computer and *IOProcessor* represents the IO processor of the mission control computer.

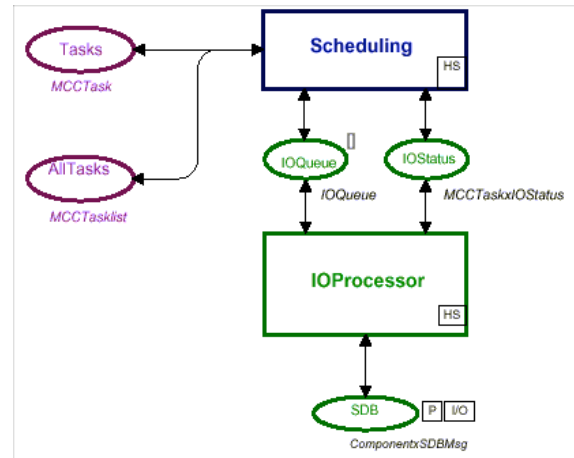


Figure 5: Mission Control Computer (MCC) page

A marking (state) of a CP-net is represented by a distribution of tokens on the places of the CPN model. The kind of tokens that can reside on a place is determined by the colour set of the place. The colour set of a place is by convention written below the place. The tokens initially present on a place are specified by the initial marking of the place. The initial marking of a place is by convention written above the place and omitted if the place is initially empty (i.e., contains no tokens).

The input socket places *Tasks* and *AllTasks* of the substitution transition *Scheduling* are used to represent the task of the mission control computer. Place *Tasks* has the colour set *MCCTask*, and each task on the mission

control computer is represented as a token of colour set *MCCTask* on the place *Tasks*.

The place *AllTasks* contains a list of all the tasks on the mission control computer. This list is used to access all tasks on the mission control computer and determine which task will be executed next. The *Tasks* place ensures that if there is a choice of the next task to execute on the mission control computer, the CPN model represents all possible such choices.

The socket places, *IOQueue* and *IOStatus*, represent the interface between the mission control computer and the IO processor. The place *IOQueue* is used to model a queue in which tasks can make requests for data to be transferred across the serial data bus. The place *IOStatus* keeps track of the input/output status of tasks, and to signal (using an interrupt) that data requested by a given task is now available. The place *SDB* represents the interface between the serial data bus and the IO processor.

4.3 Task Scheduling and Execution

The *Scheduling* page, shown in Figure 6, models the general scheduling mechanism on the mission control computer (MCC). This page is the subpage of the substitution transition *Scheduling* from Fig. 6. It consists of seven places and three substitution transitions. The port places *IOStatus*, *IOQueue*, *Tasks* and *AllTasks* are assigned to the identically named socket places on page *MCC*.

The place *MCCCPU* is used to model the state of the processor (CPU) in the mission control computer. Figure 7 shows the colour set declarations used to model the state of the CPU. The state of the CPU is modelled by the colour set *CPUState*. The CPU may either be *Idle* or *Busy*, i.e., executing a task. The colour set *BusyState* is a product where the first component is used to specify the task that the CPU is busy executing. The second component is used to record the time at which the task started executing. This information is used to compute

how much of the task was completed in case the current task is interrupted by a higher priority task. The colour set *CPUState* is timed. This means that tokens of this colour set will carry *time stamps*. These time stamps will be used to specify the time at which the task has run to completion. The initial marking of place *MCCCPU* is a token with colour *Idle* corresponding to the CPU initially being idle.

The place, *Minorcycle*, is used to keep track of the current minor cycle. As indicated by the initial marking of this place, the system starts in minor cycle 1. The place *MinTime* represents the minimum amount of time a task should spend on the CPU before it can be pre-empted by a task with a higher priority.

The substitution transition *InterruptTask* models how a task executing on the CPU can be interrupted by a task with a higher priority, and the substitution transition *TaskCompleted* describes the completion of a task. The pages associated with these substitution transitions are described in detail in Petrucci et al. (2002).

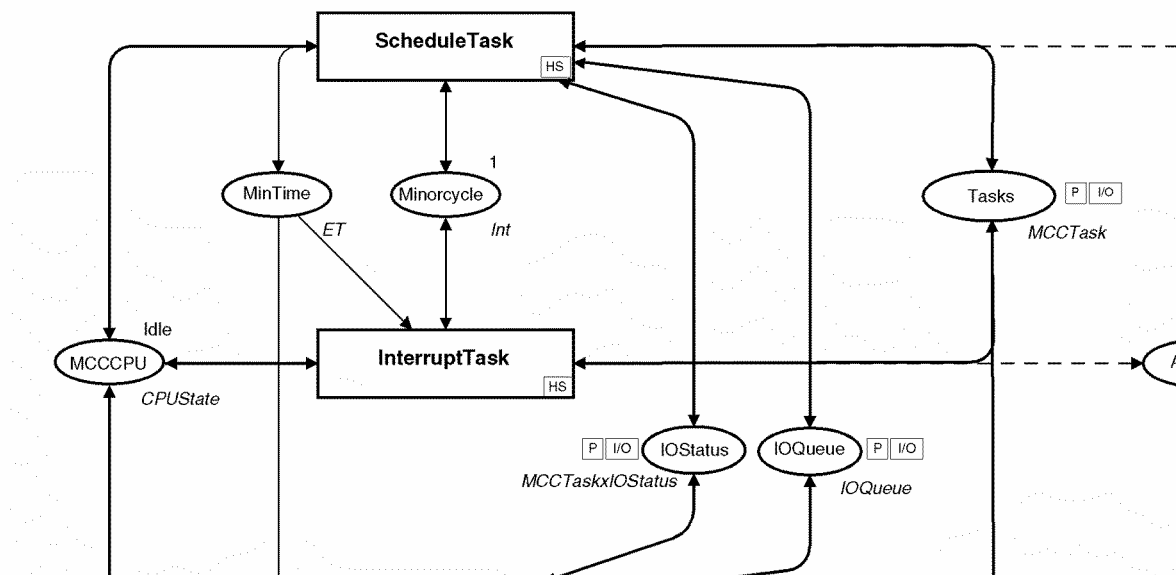
```
color CPUState = union
  Idle +
  Busy : BusyState timed;
```

Figure 7: Colour set definitions for CPU state

4.4 Serial Data Bus

The *Serial Data Bus* page shown in Figure 8 comprises three places and two transitions. There are two places *Idle* and *Busy* describe the state of the *Serial Data Bus*. The third place, *SDB*, represents the information transiting on the *Serial Data Bus*.

The *Serial Data Bus* is initially in the *Idle* state. When a request arrives on the *SDB* place from the I/O processor, it starts transmitting the request to the appropriate device (transition *Start Transmit* occurs). The serial data bus will then change its state from idle to busy by placing a token



in the place *Busy*. When the request has been transmitted, the *Serial Data Bus* becomes *Idle* again and signals that the request has been completed so that the I/O processor handles the next request (transition *Transmit Complete* fires). The places *Idle* and *Busy* ensure that a transfer on the data bus can only start if no other transfer is already being processed.

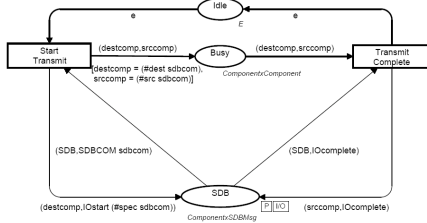


Figure 8: Serial Data Bus page

4.5 Sensors

The information necessary for tasks is gathered by various sensors such as radars. The execution of tasks causes data to be transferred between sensors, control, displays, and stores. The interaction between the *MCC* and the sensors consists of data transfers. Since we model the mission system at a high level of abstraction, the various sensors can be modelled as shown in the *GenericSensor* page (Figure 9).

When a request arrives on the *Serial Data Bus*, the relevant *Sensor* is activated and the data transfer starts (transition *Start Transfer* occurs). Each sensor takes a certain amount of time to operate and process information, and thus must remain in the *Transfer* state during this time. This is achieved by the time stamp given to the token created in place *Transfer* when transition

the requested information, it signals the *Serial Data Bus* that the request has been completed.

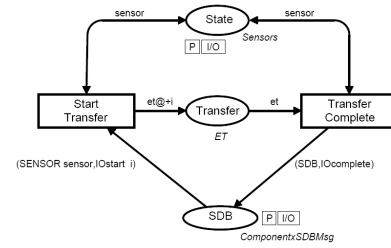


Figure 9: GenericSensor page

5 Analysis of the CPN Model

The analysis of the CPN AMS model is based on the state space method of Coloured Petri Nets as supported by the Design/CPN tool. The primary focus of the analysis has been to determine whether the tasks to be scheduled on the mission control computer are completed in time, and if this is the case, provide a schedule for the set of tasks. In addition to this, the size of the input/output queues of the I/O processor has been considered. The basic idea behind state space methods is to construct a directed graph (called the state space) with a node for each reachable state of the CPN model and an arc for each transition between states. Since the state space contains all reachable states it represents all possible executions of the CPN model. In this section, we provide a summary of our analysis approach and results which have been reported in detail in Petrucci et al. (2002).

The problem of finding a schedule can be formulated as finding a path in the state space leading from the initial state to a state where the major cycle has ended and all

Task Set	Tasks	RG	BG
S1	Displays and Controls HUDDisplay, MPDHUDDisplay, MPDTacticalDisplay, MPDButtonResponse, ChangeDisplayMode, MPFStoresDisplay, MPDStatusDisplay, KeysetResponse, HOTASDesignation, HotasBombButton	6	4
S2	S1 + Built-In Test PeriodicBIT, BITFailureWarning, InitiatedBIT	7	6
S3	S2 + Radar Control RadarSearch, radarTracking, RadarInitiateTracking	9	7
S4	S3 + Targeting DesignateTarget,ConfirmDesignation, TargetTracking, TargetSweetening	10	10
S5	S2 + Threat Response PollRWR, ThreatResponseDisplay	8	7
S6	S5 + RWR Control RWRProgramInput, RWRProgramming	9	8
S7	S6 + Weapon Control (except WeaponRelease) InputWeaponSelection, WeaponSelectionProc, AutoCCIPtoggle, WeaponTrajectory, ReinitiateTrajectory	10	12
S8	S7 + Targeting Designatetarget, ConfirmDesignation, TargetTracking, TargetSweetening	11	15

Table 1: Set of Tasks used for Analysis

Start Transition occurs. The token in place *Transfer* cannot be consumed before this amount of time has elapsed. When the sensor has terminated, i.e., transmitted

tasks were completed in time. To make state space analysis feasible, we started out by selecting a small set of tasks and gradually introduced additional tasks. Also, we experimented with different priority policies for tasks

accessing the CPU and for input/output. Table 1 lists the different sets of tasks taken from Locke et al. (1990) that are used for the analysis. The RG column gives the number of rategroup (periodic) tasks in a given set. The BG column specifies the number of background tasks in the set.

Table 2 gives the size of the state space for the different sets of tasks listed in Table 1. The Nodes column gives the number of nodes in the state space, and the Arcs column gives the number of arcs in the state space. The IOSS column gives the maximum number of requests in the I/O requests queue at the I/O processor observed in the state space. The IOSP column gives the maximum number of requests in the I/O queue along a path in the state space corresponding to a schedule for the tasks. The considered pre-emption and queuing policy allowed requests from rategroup tasks to overtake requests from background tasks in the IO queue, and both rategroup and background tasks had assigned priorities.

Set	Nodes	Arcs	IOSS	IOSP
S1	77,982	127,316	6	4
S2	78,734	128,715	7	6
S3	485,054	811,734	9	7
S5	144,780	235,769	10	10
S6	142,022	234,257	8	7
S7	409,888	702,831	9	8

Table 2: Standard State Space Generation

Various state space analysis techniques were considered, for example, the use of depth-first state space generation allowed the S4 and S8 task sets to be analysed, as this led to significantly fewer states to be considered (Petrucci et al. 2002).

Our analysis focuses on task scheduling of the mission control computer and only considers a single major cycle. This is sufficient because all tasks are required to be executed at the end of a major cycle. Analysis results showed that, for example, the rategroup task *WeaponRelease* cannot execute in time i.e. it fails to meet its deadline. We have demonstrated how analysis of scheduling and input/output queue of an avionics mission system can be done using state spaces.

6 AP-3C Orion Mission System Modelling

The AP-3C aircraft mission system upgrade provides enhanced mission capabilities and extends the P-3C life-of-type to 2015 (DMO 2002). The AP-3C aircraft is operated by the Maritime Patrol Group of the Royal Australian Air Force, and its mission roles include anti-subsurface and anti-surface warfare, surveillance, search and rescue, and maritime strike.

A high-level block diagram of the AP-3C avionics mission system architecture is shown in Figure 10 (RAAF 1996). It consists of a Mission Equipment Bus (MEB) that provides inter-communication channels between the following sub-systems: Navigation, Acoustics, Magnetic Anomaly Detection (MAD) and Radar. The Data Management System (DMS) provides specialised

interfaces to the following sub-systems: Armament/Ordinance (ARM/ORD), Electronic Support Measures (ESM), and Infrared Detection System (IRDS). The operators are provided with a high-resolution display and entry panel directly from the DMS. The Communication (Comm) sub-system interfaces with these sub-systems via the Avionics Equipment Bus (AEB), which is connected to the MEB via the Navigation sub-system. The MEB and the AEB are dual redundant serial data busses based on the MIL-STD-1553B (DOD 1993). The DMS of the AP-3C plays the same role as the mission control computer in the generic mission system.

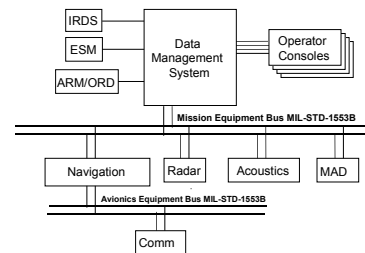


Figure 10: AP-3C AMS Architecture

The DMS is a centralised mission control and management sub-system. It is a complex multiprocessor system consisting of several Enhanced General Purpose Controllers (EGPC), custom computing devices and supporting software. Two input/output (I/O) processor cards provide the interface between the EGPCs and the MEB. The DMS provides the overall AMS management and normally acts as the MEB bus controller (RAAF, 1996).

The DMS software consists of a set of software components, which process sensor data and input from controls, performs necessary mission-oriented computations and provides outputs to the displays and other avionics equipment (RAAF 1999). Typical software components include the executive, navigation, stores management, display control, and data management. The runtime executive schedules and dispatches the execution of control tasks and services interrupts during various operations. The runtime executive component is typically responsible for the following functions: application software (task) scheduling, interrupt management, input/output scheduling and error management. The DMS executive software is based on a commercial Ada runtime kernel (RAAF 1999). The scheduling policy is pre-emptive and is executed by priority in a round-robin fashion (Rational 1995).

An EGPC sends a number of messages to the I/O processor for transmission to the addressed remote terminal (RT) (RAAF 1996), e.g. Navigation sub-system or Radar subsystem, via the MEB. The MEB minor frame rate (described in the next section) sets a real-time clock interrupt to the I/O software. At the beginning of each new minor frame, an interrupt occurs, and the I/O processor starts issuing the messages for that frame.

The AP-3C CPN model has been constructed based on the framework represented by the CPN AMS model briefly described in Section 4. From the two architectures

depicted in Figures 1 (generic) and 10, it follows that although the components in the two systems are basically the same, the structure is quite different. There are two buses in the AP-3C instead of one in the generic AMS. This is easy to handle in our model as they are identical, allowing us to use two instances of the same bus representation. We just need to connect the appropriate components to each bus instance to ensure that the model correctly reflects the topology of the system. In this section, we provide an overview of the model only, and the reader is referred to Petrucci et al. (2003) for details.

Finally, we need to consider two sorts of subsystems (or *device*) in the AP-3C mission system, rather than one. Because of the level of abstraction chosen for the model, we refer to subsystems or devices that have just a single connection to a bus as a *simple device*, such as, Magnetic Anomaly Detection, the Acoustic Subsystem, and the Communication Subsystem. We need a more complex model of the Navigation Subsystem because it is connected to both busses. Therefore, the simple devices can be modelled in the same way as the devices of the generic mission system (the Generic Sensor), while the navigation subsystem requires an enhancement to this model.

The hierarchy page of the AP-3C CPN model is depicted in Figure 11; a parallel should be drawn between this page and the generic AMS model hierarchy in Figure 2. The four main differences between the CPN models of the generic AMS and the AP-3C, as reflected in the hierarchy pages, are as follows:

- The *Sensors* and *Generic Sensor* pages from the generic AMS model have been combined and then split into two: the simple devices (in the Device page) and the navigation subsystem.
- The CPN model of the AP-3C AMS contains a refined model of the input/output processing on the mission control computer. In the generic CPN model, input/output processing was modeled by page *IOProcessor*. In the AP-3C model, input/output processing is modelled by page *IOProcessorCard* and its three subpages. The Scheduling page of the generic model becomes the *EGPC* page for the AP-3C.
- The timing regarding task execution has been moved from the mission control computer level to the EGPC level, and the page *UpdateMajorCycle* is now a subpage of the *EGPC* page.

The differences between the AP-3C and the generic AMS architectures are easily recognised by examining the CPN models' hierarchy pages. The transformation of the generic model into the AP-3C model was greatly facilitated by its initial hierarchical design. The transformation mainly consists of: re-arranging the hierarchy by moving some pages; creating new ones when refinement is required; and deleting pages not relevant to the specific architecture or the purpose of the model.

One purpose of the CPN model is to formally specify the transmission of messages between subsystems across the mission equipment and avionics bus. As usual we model

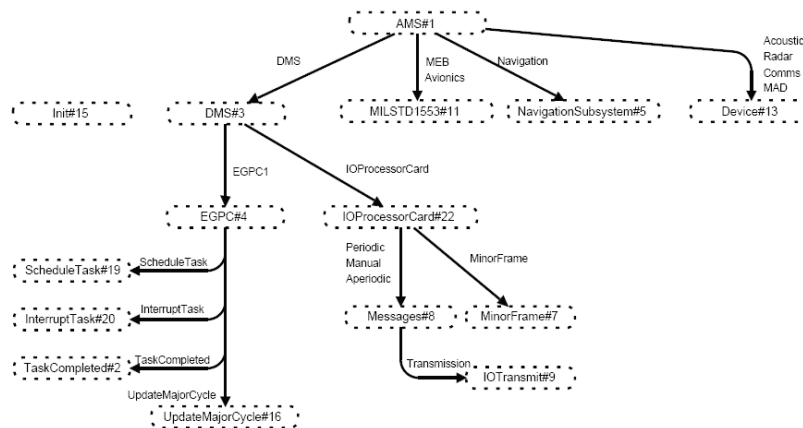


Figure 11: Hierarchy page of the AP-3C model

- The *Stores* page and the *ControlsDisplays* page and its subpages have been removed from the model. The reason is that we are initially concerned with scheduling problems associated with the DMS that are not related to the displays and controls.

the messages being transferred as tokens in the CPN model. When a subsystem transmits a message across the mission equipment bus to another subsystem, it will put a token on place *Mission Equipment Bus*. The subpage of the substitution transition *Mission Equipment Bus* will

then model the details in transferring the message. Eventually the message will be transmitted and the destination subsystem will consume the token representing the message from place *Mission Equipment Bus*. To model this data transfer in a flexible way that makes it easy to add/remove components, a general addressing scheme was developed as part of the CPN model of the generic AMS.

The top-level AMS page is shown in Figure 12 and corresponds to the most abstract level in the AP-3C CPN model. This page provides a high-level architectural view of the AP-3C AMS similar to the informal block diagram in Figure 12. All the transitions (rectangles) are substitution transitions, indicated by the HS (hierarchical substitution) tag in their lower right corner. The substitution transition *Data Management System* represents the main part of the system. The *Communication Subsystem*, *Navigation Subsystem*, *Acoustic Subsystem*, *Magnetic Anomaly Detection* and *Radar* correspond to the different AMS subsystems. The other two substitution transitions, *Mission Equipment Bus* and *Avionics Bus*, are used to model data transfer across the MIL-STD-1553B serial data busses. Places *Mission Equipment Bus* and *Avionics Bus* represent the interfaces for each bus. Finally, the other places (e.g. *ACS*) allow subsystems (such as the acoustics subsystem) to be identified.

Figure 13 depicts the DMS page which is the most abstract part modelling the data management system. The page has two substitution transitions: *EGPC1* represents the Enhanced General Purpose Controller on which the tasks execute, and the *1553B I/O Processor Card* handles all the input and output related to tasks. The modelling

EGPC4). Thus, several instances of the *EGPC* page may be used concurrently.

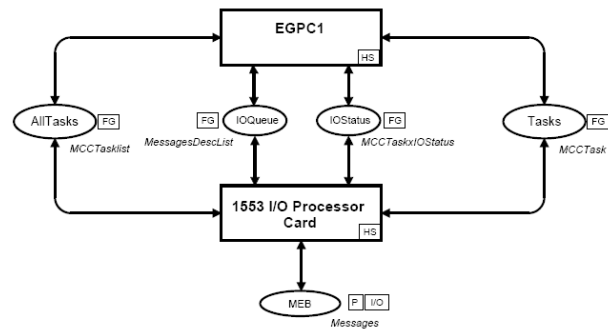


Figure 13: DMS page of the AP-3C model

7 Conclusions and Way Ahead

This paper has provided an overview of the experiences on a joint research project between the University of South Australia and Australia's Defence Science and Technology Organisation into the modelling and analysis of avionics mission systems, conducted during the period 2000-03.

We have described the development of a formal model of a generic mission system using Coloured Petri Nets. The main outcome is the capture of system domain knowledge, understanding of the mission system architecture and the interrelationships between the various avionics subsystems. We have demonstrated how analysis of task scheduling of an avionics mission system can be done using state spaces. A virtue of our modelling approach is that the CPN model is highly parametric,

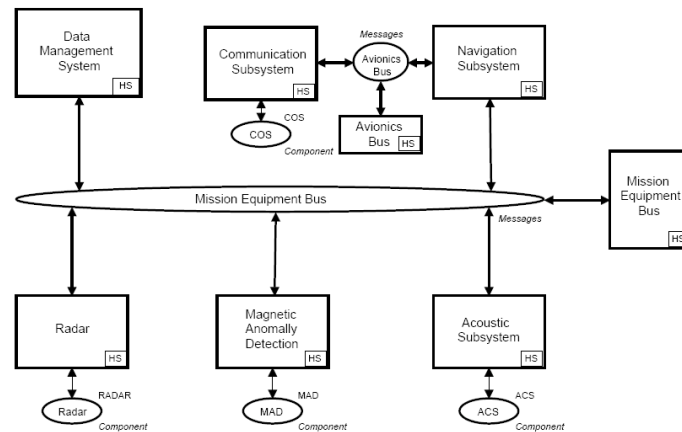


Figure 12: AMS page of the AP-3C model

details of processing and messages transmission on the IO Processor Card is described in detail in Petrucci et al, (2003). The AP-3C aircraft uses up to 4 EGPCs. Our model can easily cater for this by including the required number of EGPC substitution transitions (e.g. *EGPC1* to

which makes it easy to analyse different sets of tasks and in this way investigate the impact of adding tasks to the mission control computer. Another advantage of our modelling approach is that specific scheduling mechanisms can easily be changed and an analysis of their impact can be conducted.

We have shown how the hierarchical constructs of CP-nets and Design/CPN that were successfully used to model the generic airborne mission system could be readily applied to model the AP-3C mission system. The initial CPN model of AP-3C provides a basis to perform analysis of the mission system architecture and is focused on the Data Management System task scheduling and data transfer across the Mission Equipment Bus. The CPN model can serve as an unambiguous executable specification of the DMS. Since the CPN models are executable, the behaviour of the DMS can be observed by simulating the CPN DMS model. This can prove to be an efficient way of gaining and maintaining knowledge on the operation of the DMS. The CPN DMS model can be used to analyse what-if scenarios before the actual system integration. These what-if scenarios could be related to functional as well as performance aspects of the DMS.

A major problem is the state-space explosion, for example the state space of the CPN AMS model grows with the increase in the number of tasks. In order to perform more efficient state space analysis, the use of advanced methods such as the sweep-line methods (Christensen et al. 2001) should be investigated.

Safety-critical software for aviation is typically DO-178B (RTCA 1992) level A or B (BAE, 2004). Mission-critical software applications must also be developed using the guidelines of DO-178B. However, unlike safety-critical applications, mission-critical software is typically DO-178B level C or D (BAE, 2004). Formal methods can play an important role in the design evaluation of mission-critical software and systems.

Furthermore, there is a need to investigate other methods and techniques for the modelling and analysis of avionics mission systems, such as: Avionics Architecture Description Language (SAE-AADL) and associated MetaH tool (Vestal 1997), model based verification and lightweight formal methods (Gluch and Weinstock 1998), and formal verification techniques and tools, for example, SPIN model checker (Holtzmann 1997), PVS theorem prover (Crow et al. 1995), and another promising verification tool (under development) - the Hierarchical Verification Environment (HiVE) (Cant et al. 2005),

Finally, in addition to the research on modelling that has been presented in this paper, we recommend a number of critical research areas to complement the overall program for research on avionics mission systems, namely, advanced avionics architectures including safety, availability, fault-tolerance and growth issues, avionics data bus architectures and performance issues, real-time schedulability and timing analysis, and software and hardware design assurance.

8 Acknowledgements

This research was conducted under the Long Range Research Task scheme, Task No. LRR 00/061, during the author's association, as the task manager, with the Australian Defence Science and Technology Organisation. The author would like to acknowledge Professor Billington, University of South Australia on the research collaboration and guidance with the application

of Coloured Petri Nets. The author would also like to acknowledge Lars Kristensen, Aarhus University for the research on the development of the generic avionics mission system CPN modelling framework and Laure Petrucci, Laboratoire Spécification et Vérification, ENS de Cachan, for the formal analysis and refinement of the framework to the AP-3C aircraft, during their sabbatical at the Computer Systems Engineering Centre, University of South Australia. Finally, acknowledgements are due to Raymond Kiefer and Scott Simmonds from Tenix/RLM and Adacel Technologies respectively, for the technical discussions and design information on the AP-3C Data Management System, and to Flight Lieutenant Jon Postle, AP-3C DMS Manager, Royal Australian Air Force.

9 References

- BAE (2004): *Safety-critical vs. mission-critical: Understanding the difference*, CompactPCI and AdvancedTCA Systems, January, OpenSystems Publishing, <http://www.compactpci-systems.com/articles/id/?263>; Accessed 19 July 2006.
- Bjorner, D. and Druffel, L. (1990): Position statement: ICSE-12 Workshop on Industrial Experience Using Formal Methods, *Proceedings of the 12th International Conference on Software Engineering*, 264-266, March 26-30, Nice, France.
- Cant, T., Mahony, B., McCarthy, J., and Vu, L. (2005). Hierarchical Verification Environment, *Tenth Australian Workshop on Safety Critical Systems and Software*, SCS 2005, Cant, T. (Ed.), Conferences in Research and Practice in Information Technology, 55: 47-57, Australian Computer Society.
- Christensen, S., Kristensen, L.M. and Mailund, T. (2001): A Sweep-Line Method for State Space Exploration, *Proc. TASCAS'01, LNCS*, 2031:450-464, Springer Verlag.
- CIT (2000): Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions, Jet Propulsion Laboratory, California Institute of Technology.
- CoA (2001): Major Capital Equipment Project Delays or Cost Overruns, Additional Budget Estimates – Defence Portfolio, February, Defence White Paper Projects, Senate Foreign Affairs, Defence and Trade Legislation Committee, Canberra, Commonwealth of Australia.
- Crow, J., Owre, S., Rushby, J., Shankar, N. and Srivas. M. (1995): A Tutorial Introduction to PVS, *Proc. of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, USA, Computer Science Laboratory, SRI International.
- Desel, J. and Reisig, W. (1998): Place/Transition Petri Nets, *Lectures on Petri Nets I: Basic Models*, *Lecture Notes in Computer Science*, 1491:122-173, Springer-Verlag.
- Design/CPN Online: Computer Tool for Coloured Petri Nets, University of Aarhus, <http://www.daimi.au.dk/designCPN/>. Accessed 7 Mar 2000.

- DMO (2002): Projects, Air 5276 Phase 2A - P3 Update Implementation, Defence Materiel Organisation, <http://www.defence.gov.au/dmo/asd/air5276/air5276p2.cfm>, Accessed 25 July 2002.
- DOD (1993): MIL-STD-1553B Notice 3, 31 Jan 1993, Military Standard, Digital Time Division Command/Response Multiplex Data Bus, Department of Defense, Washington DC.
- GAO (2003a): DOD Should Reconsider Decision to Increase F/A-22 Production Rates While Development Risks Continue, Report to Congressional Committees, March, GAO-03-431, General Accounting Office, Washington, D.C.
- GAO (2003b): Defense Acquisitions: Assessments of Major Weapon Programs, Report to Congressional Committees, May, GAO-03-476, General Accounting Office, Washington, D.C.
- GAO (2004): Status of the F/A-22 and Joint Strike Fighter Programs, Testimony before the Subcommittee on Tactical Air and Land Forces, Committee on Armed Services, House of Representatives, March 25, GAO-04-597T, General Accounting Office, Washington, D.C.
- Gluch, D.P. and Weinstock, C.B. (1998): Model-Based Verification: A technology for Dependable System Upgrade, Technical Report CMU/SEI-98-TR-009, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
- Havelund, K. and Lowry, M. (2001): Formal Analysis of a Space-Craft using SPIN, *IEEE Transactions on Software Engineering*, 27(8):749-765.
- Hinchey, M.G. and Bowen, J. P. (Eds.), (1995): *Applications of Formal Methods*, International Series in Computer Science, UK, Prentice Hall.
- Holtzmann, G. (1997): The Model Checker SPIN, *IEEE Transactions on Software Engineering*, 23(5):279-295.
- Jensen, K. (1997): *Coloured Petri Nets. Basic Concepts, Analysis Method and Practical Use (Vol. 1-3)*, Monographs in Theoretical Computer Science, Second Edition, Springer-Verlag.
- Kristensen, L. M., Billington, J. and Qureshi, Z. H. (2001): Modeling Military Airborne Mission Systems for Functional Analysis, *Proc. 20th IEEE/AIAA Digital Avionics Systems Conference*, Daytona Beach, Florida, 14-18 October.
- Kristensen, L. M., Billington, J., Petrucci, L., Qureshi, Z. H. and Kiefer, R. (2002): Formal Specification and Analysis of Airborne Mission Systems, *Proc. 21st IEEE/AIAA Digital Avionics Systems Conference*, Irvine, California, 27-31 October.
- Liu, J. (2000): *Real-Time Systems*, New Jersey, Prentice-Hall.
- Locke, C.D., Vogel, D.R. and Goodenough, J. B. (1990): Generic Avionics Software Specification, Technical Report CMU/SEI-90-TR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
- Locke, C.D. (1992): Software Architecture for Hard Real-Time Applications: Cyclic Executive vs. Fixed Priority Executives, *The Journal of Real-Time Systems*, 4:37-53.
- Petrucci, L., Kristensen, L. M., Billington, J. and Qureshi, Z. H. (2002): Towards Formal Specification and Analysis of Avionics Mission Systems, In Lakos, C., Esser, R., Kristense, L. M., and Billington, J. (Eds.), *Proc. Workshop on Formal Methods Applied to Defence Systems*, June, Conferences in Research and Practice in Information Technology, 12:95-104, Australian Computer Society.
- Petrucci, L., Billington, J., Kristensen, L. M. and Qureshi, Z. H. (2003): Developing a Formal Specification for the Mission System of a Maritime Surveillance Aircraft, *Proc. 3rd International Conference on Application of Concurrency to System Design*, Guimaraes, Portugal, 18-20 June.
- RAAF (1996): Mission Equipment Bus (MEB) Protocol Specification. Report F6250.00.151, 11 March 1996, (CDRL-ENG-46-ACS-ICD-0074, Rev D 10 Jan 2000). Project Air-5276 Royal Australian Air Force, Department of Defence, Australia.
- RAAF (1999): Software Design Document for the Operating Systems CSCI, Report 7371902, Rev B, 26 February 1999, (SDRL-ENG-65D-03-03, CDRL-ENG-32-DMS-SDD-0423, 2 July 1999). Project Air-5276 Royal Australian Air Force, Department of Defence, Australia.
- Rational (1995): Runtime Systems Guide VADScross M68000, ver: 6.2.3.0, June, Rational Software Corporation, Santa Clara, California.
- Rockwell (1992): F/RF-111C Avionics Update Program, 1992, Software Design Document for the Mission Computer Operational Flight Program, Volume I, Rockwell International Corporation.
- RTCA (1992): *Software Considerations in Airborne Systems and Equipment Certification*, DO-178B, Washington DC, RTCA.
- SAE-AADL: SAE AADL Information Site, A Society of Automotive Engineering Standard, <http://www.aadl.info/>. Accessed 11 April 2006.
- Ullman, J. (1998): *Elements of ML Programming*, New Jersey, Prentice-Hall.
- Vestal, S. (1997): MetaH Support for Real-Time Multi-Processor Avionics, Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS/OORTS '97), IEEE Computer Society.

Certification Criteria for Emulation Technology in the Australian Defence Force Military Avionics Context

Flight Lieutenant Derek Reinhardt

Systems Certification and Integrity (SCI)
Directorate General Technical Airworthiness (DGTA)
Bldg L474 (B-2-STH), RAAF Williams, Laverton 3027, Victoria, Australia
derek.reinhardt@defence.gov.au

Abstract

Emulation technology promises to provide a means of addressing obsolescence issues in legacy computer processors in the military avionics domains. It has also been suggested that such technology might apply to safety critical and safety related systems in these domains. Numerous companies either have developed or are developing software components that are capable of emulating different legacy computing platforms. The emulators permit the execution of legacy code on newer computing platforms, without change to existing binary executables or data. Subsequent modifications to the legacy code in question may be made using either the legacy development environment and/or with some emulation technologies using a newer development environment.

The Defence Science and Technology Organisation (DSTO) is presently working with Northrop Grumman Space Technology (NGST) to develop a concept demonstrator utilising NGST's Reconfigurable Processor for Legacy Applications Code Execution (RePLACE) Emulation Technology for the Royal Australian Navy (RAN) Seahawk Display Generator Unit (DGU). To assess how the Australian Defence Force's (ADF's) Technical Airworthiness Authority (TAA) - the Directorate General Technical Airworthiness (DGTA) might accept emulation technology, DGTA has evaluated emulation architectures and specifically RePLACE in the context of the Seahawk DGU. The evaluation has considered the emulation architecture, including identification of risks largely unique to the technology; as well as application of ADF preferred avionics software assurance and software safety standards to this technology.

Evaluation of emulation technology, through exploration of emulation architectures and RePLACE as a case study, has allowed DGTA to define certification and regulatory guidance for the development of emulation technology within the ADF context.

Keywords: Certification, Emulation, Legacy Systems, Software Architecture, Software Safety, Safety Critical.

1 Introduction

The concept of emulation and emulators has been around for many years. Emulators permit the execution of legacy code on newer computing platforms, without changes to existing binary executables or data.

In recent years the power of modern microprocessors has evolved to such an extent that it is now possible to provide real-time software emulation of many legacy microprocessors that were widely used in the late 1970s, 1980s and early 1990s. These advances provide a tremendous opportunity to reuse much of the software developed for these earlier microprocessors without the penalty of having to rehost or translate the software to modern programming languages and microprocessor environments. Furthermore, emulation promises to solve the problem of hardware obsolescence among those legacy systems that are still in use today.

Emulation has been applied to many areas of computing already, including the emulation of older computer game consoles (Atari, Sega, Nintendo Entertainment System, Arcard Platforms, etc), emulation of earlier derivatives of PC, Macintosh, Unix and VaxVMS environments to permit execution of those old applications in a modern environment, and emulation of embedded systems to permit analysis, testing and simulation on development platforms to name but a few. Given the wide application to date of emulation, it is not surprising that emulation technology is now being suggested in the avionics domain, particularly the military avionics domain.

Emulation technology promises to provide a means of addressing many obsolescence issues in legacy computer processors in the military avionics domains, a domain where systems can be subject to comparably longer service lives than equipment in other domains. For example, it is not unusual for military aerospace systems to be in service for 30+ years, although some avionics systems would reasonably be expected to be upgraded over that time period. There have already been a number of programs that have used emulation in this sense. Numerous companies either have developed, or are developing, software components that are capable of emulating different legacy computing platforms for military avionics. One such company is Northrop Grumman Space Technology (NGST) who offer a product called RePLACE. RePLACE has already been

applied to numerous avionics systems and microprocessor instruction sets.

Furthermore, it has also been suggested that such technology might be further applied to safety critical and safety related systems in these domains. To date, there is a lack of regulatory guidance and certification criteria relating to how emulation might be applied to safety critical and safety-related systems.

The ADF's Technical Airworthiness Regulator (TAR) – also DGTA, is responsible for defining regulatory and certification criteria for modifications to Australian Military ('State') aircraft. This provides the ADF's TAA with the guidance from which to conduct design acceptance of such technologies. Note that DGTA has a dual responsibility, being both the TAR and TAA. Design acceptance is largely synonymous with type certification within the Federal Aviation Administration (FAA) airworthiness framework.

The Defence Science and Technology Organisation (DSTO) are presently working with NGST to develop a concept demonstrator utilising NGST's RePLACE Emulation Technology for the RAN Seahawk DGU. The DGU hosts several functions that are safety-related, and therefore warrants special consideration within the context of emulation. This development provides DGTA with an opportunity to develop certification criteria for emulation technology and to assess the effectiveness (technical, cost, schedule) of such certification criteria.

The remainder of this paper examines emulation technology, through exploration of emulation architectures and NGST's RePLACE as a case study, to allow DGTA to define certification and regulatory guidance for the development of emulation technology within the ADF context.

2 Examination of Emulation Architectures

In order to define certification criteria for emulation technology, it is firstly necessary to develop an understanding of those software architectures most relevant to emulators. This section introduces the simplest form of emulator architectures.

2.1 A Simple Emulation Architecture (Type 1)

A simple legacy emulation architecture (designated Type 1 for convenience of reference throughout this paper) is detailed in Figure 1 and Figure 2.

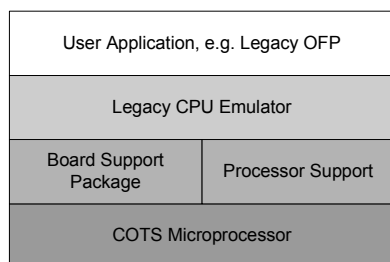


Figure 1: Simple Emulator Architecture (Logical Layers)

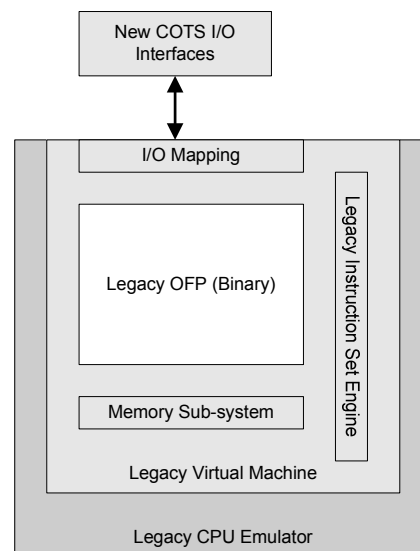


Figure 2: Simple Emulator Architecture (Sub-Elements)

The following paragraphs provide an overview of the emulator components detailed in Figure 2.

The main component of the emulator architecture is the *Legacy Virtual Machine*. The Legacy Virtual Machine consists of the Legacy Instruction Set Engine, Memory Sub-system, I/O Mapping, Legacy Operational Flight Program (OFP) (ie. the binary) and other underlying functionality necessary to emulate the legacy computer environment (eg. Interrupt/exception mechanisms). Encapsulating the Legacy Virtual Machine is the *Legacy CPU Emulator* which provides the interface for the Legacy Virtual Machine to execute in the native processor environment. It is included as a separate element in this architecture for consistency with some more complex architectures considered later in this paper.

The *Memory Sub-System* component's role is to model the memory of the legacy computer environment. This may include logical to physical address translation, memory protection mechanisms and memory regions (non-volatile regions, read-only regions, shared memory regions, etc.).

The *I/O Mapping* component's role is to match the data and control structures, as well as the interfaces of the new replacement I/O devices, to those that are representative of the legacy computer environment.

The *Legacy Instruction Set Engine* is a set of native machine code that fetches, decodes and executes the legacy instructions on the fly. Figure 3 describes the relevant data and information flows that might occur in one such implementation of the Legacy Instruction Set Engine. This example has been based on the MIPS processor, which is generally well understood across the computing domain, although the logical interpretation is easily extended to any type or class of microprocessor. Note that the MIPS processor is not used in the Seahawk DGU, which uses the AAMP processor.

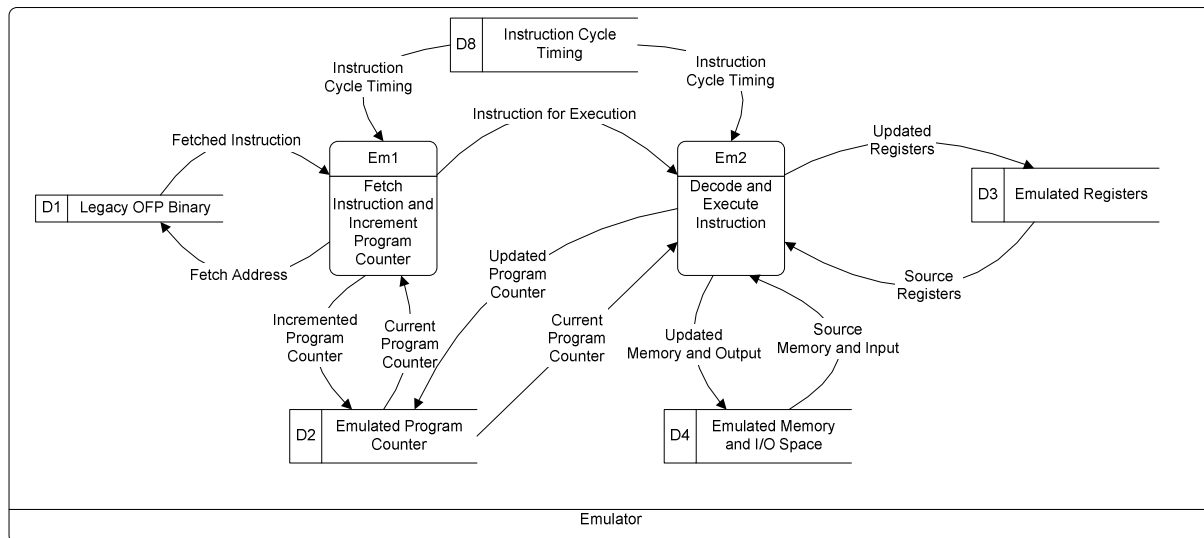


Figure 3: Legacy Instruction Set Engine Data/Information Flow Diagram

2.1.1 Incorporating an RTOS (Type 1A)

Rather than implementing the full suite of system related functions as part of the emulator, it is common for embedded applications of this type to incorporate some form of Real Time Operation System (RTOS). Figure 4 and Figure 5 show the logical layers and sub-elements that such an architecture might consist of. Aside from the incorporation of the RTOS between the emulator and lower level board/processor support firmware/software and the microprocessor and I/O interface, there is no significant change to the components, functional structure or relevant data and information flows within the emulator itself.

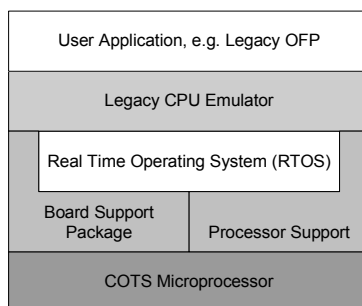


Figure 4: Emulator Architecture (Logical Layers) - Incorporating an RTOS

Of the emulators examined by DGTA, this architecture is the most widely adopted, and will form the starting point for analysis aimed at determining certification criteria for emulation technology.

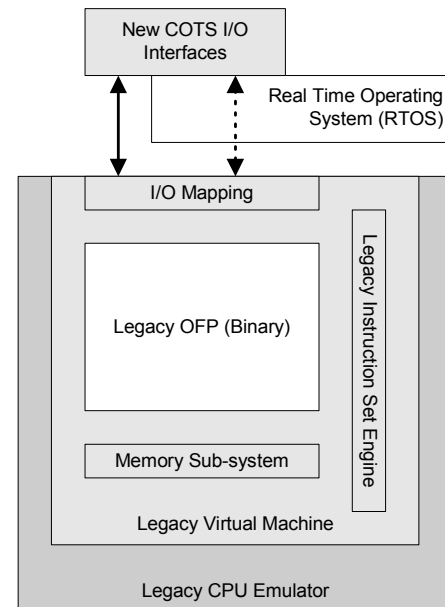


Figure 5: Emulator Architecture (Sub-Elements) - Incorporating an RTOS

3 Analysis of Type 1 Emulation Architecture

To provide an understanding of the software failure modes that might be relevant to the emulation architecture, and importantly what architectural considerations and software assurance activities are required to provide evidence of the absence or handling of these identified failure conditions, it is necessary to conduct some form of software safety analysis. There are numerous software safety analysis techniques that could be applied to such a system including Software Functional Failure Analysis (FFA), Software Fault Tree Analysis (FTA), Software FMEA (FMECA), Software HAZOP (DefStan 00-58 Computer HAZOP), Software Hazard Analysis and Resolution in Design (SHARD) - refinement of Software HAZOP, Markov Analysis and Data Flow Diagrams, Petri Net Analysis and Software Sneak Analysis (ADF 2006, and McKinlay 2001).

Guide Word	Deviation	Cause	Effect	Detection / Protection
Omission	Decode and execute instruction process fails to update memory or output	Programming error within decode and execute instruction process	Memory not updated with new contents Output not updated with new output	Memory and output post update verification
Commission	Decode and execute instruction process invalidly updates memory or output	Programming error within decode and execute instruction process	Memory is corrupted in specific location Output is corrupted	Entry point to decode and execute instruction process limited to following fetch instruction and increments counter process
Early	Decode and execute instruction process updates memory or output before valid processor cycle	Programming error causes instruction implementation to incorrectly replicate cycle synchronisation Cycle synchronisation incorrect	Memory updated out of sequence with other operations Output transitions early	Memory and output update to explicitly check cycle synchronisation Mappings to be established between legacy instruction and emulation implementation and mappings to be verified for functional and temporal equivalence.
Late	Decode and execute instruction process updates memory or output after valid processor cycle	As for early	Memory updated out of sequence with other operations Output transitions late	As for early
Value	Decode and execute instruction process updates memory or output with invalid value or updates wrong memory or output location	Programming error within decode and execute instruction process Incorrect instruction passed to decode and execute process	As for Omission, Commission, Early, Late	As for Omission, Commission, Early and Late.

Table 1: Extract from SHARD on Type 1 Emulator (Updated Memory and Output)

Guide Word	Deviation	Cause	Effect	Detection / Protection
Omission	Decode and execute instruction process fails to update program counter as result of jump instruction	Programming error within decode and execute instruction process	Program counter is not updated with correct value. Emulated program enters incorrect branch of instructions - possible program crash	Program counter is to be verified after operation. Mappings to be established between legacy instruction and emulation implementation and mappings to be verified for functional and temporal equivalence.
Commission	Decode and execute instruction process invalidly updates program counter	Programming error within decode and execute instruction process	Program counter is updated with corrupted value. Emulated program enters incorrect branch or instructions - probable program crash.	Mappings to be established between legacy instruction and emulation implementation and mappings to be verified for functional and temporal equivalence.
Early	Decode and execute instruction process updates program counter before valid processor cycle	Programming error causes instruction implementation to incorrectly replicate cycle synchronisation Cycle synchronisation incorrect	Program counter updated out of sequence with other operations. Emulated program enters incorrect branch or instructions - probable program crash.	Program counter update to explicitly check cycle synchronisation Mappings to be established between legacy instruction and emulation implementation and mappings to be verified for functional and temporal equivalence.
Late	Decode and execute instruction process updates program counter after valid processor cycle	As for early	As for early	As for early
Value	Decode and execute instruction process updates program counter with invalid value	Programming error within decode and execute instruction process Incorrect instruction passed to decode and execute process	As for Omission, Commission, Early, Late	As for Omission, Commission, Early and Late.

Table 2: Extract from SHARD on Type 1 Emulator (Updated Program Counter)

While it is possible to apply aspects of each of these techniques to analyse emulation architectures, and indeed the measured application of a number of these techniques would probably be necessary for the developer of such technologies to provide sufficient evidence as part of a safety case, it is not necessary for defining certification criteria. For the sake of defining certification criteria it is only necessary to develop an understanding of how emulation technology might fail and what might be done to either ensure it can't or doesn't fail; or if it can, then verify that it is sufficiently unlikely to fail. Such understanding should then provide insight into what evidence is required to provide sufficient confidence in these aforementioned properties. The SHARD technique is particularly relevant to developing this understanding as it considers failure modes, their causes, effects, and potential detection or protection means.

SHARD employs a series of guidewords to classify how the information flows and associated communication

events (and associated services) might deviate from their intended forms. These are as follows:

- Omission - Service not delivered.
- Commission - Service delivered when not required.
- Early - Service delivered, but early.
- Late - Service delivered, but late.
- Value - Service delivered, but with incorrect value.

SHARD requires that the system be analysed “backwards” from the outputs (ie. identify the system level effects first) back towards the inputs. The internal and input deviations are expressed in terms of how they cause or contribute to deviations in downstream items already investigated. Further information on the SHARD technique can be found in Pumfrey (1999).

SHARD analysis was conducted using the Legacy Instruction Set Engine Data/Information Flow Diagram

(Figure 3) as a reference for information flows that might exist in the emulator, and services that are required from a functional perspective. An extract from the SHARD is presented in Table 1 and Table 2 for the updated memory and output, and updated program counter information flows respectively.

Both Table 1 and Table 2 refer to the term ‘temporal equivalence’. ‘Temporal equivalence’ is used in a general sense here for convenience as each emulator implementation will need to define, within high and low level requirements, this property in the context of the relevant instruction set; legacy CPU architecture, including consideration for pipelining and parallel execution paths; and the configured application set. It is used to capture timing considerations at two levels of abstraction. The first is at the instruction level, which deals with the timing constraints placed on individual instructions, or sequences of instructions for some parallel architectures. The second is at the application level, which deals with ensuring that assigned tasks complete within their scheduled execution time, and that implementation quirks, such as processor cycle based synchronisation schemes (as opposed to interrupt timer based schemes) and the use of No Operations (NOPs) for timing synchronisations do not result in undesired effects (e.g. speed up) when emulated. This second level of abstraction is mostly applicable to those architectures considered in Section 4, however it cannot be ruled out in this context due to potential for synchronisation dependencies (e.g. those resident in timing sensitive executives and I/O). This implies that inspection and analysis of the legacy binary will be required to determine if these schemes are part of the implementation.

Having developed an understanding of the types of failure modes, their causes, effects, and detection/protection means, it is then possible to define architectural or verification requirements relative to those failure modes. The ADF preferred standard for software assurance of airborne software is RTCA/DO-178B (ADF 2005). For the purposes of consistency and clarity, verification requirements shall be defined based on those activities documented in RTCA/DO-178B (RTCA 1992). Readers should refer to DO-178B and related information (DO-248B (RTCA 2001), Order 8110.49 (FAA 2003), CAST 5 (CAST 2000)) for further definitions of software assurance activities described in this paper.

However, to understand the logic behind the approach used to define those architectural and verification requirements and associated DO-178B objectives, it is firstly necessary understand some key aspects of the DO-178B software assurance model. According to DO-178B, verification of airborne software has two complementary objectives. One objective is to demonstrate that software satisfies its requirements. The second objective is to demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions have been removed. Noting that the prescription of activities against these two objectives is scaled based on software level within the standard, it is worth considering the approach in general terms.

The first objective is largely supported through definition of and verification against high-level requirements. Where insufficient disclosure or ambiguities exist within the high level requirements, then refinement and further definition of and verification against is required in translation to low-level requirements. Therefore, it follows that provided the developer can adequately disclose the requirements at the prescribed level of detail, then this objective is relatively straight forward to satisfy.

The second objective, however, is not quite as intuitive. It deals with eliciting properties about the software which don’t necessarily follow from the set of already defined high and low level requirements, with focus on those properties that could potentially lead to unacceptable failure conditions. Eliciting these properties permits one of two outcomes: either the behaviour is appropriate, in which case it should be captured in the high and/or low level requirements; or the behaviour is inappropriate, in which case the software design and implementation should be changed to remove the behaviour. DO-178B approaches this through prescribing requirements coverage analysis and software structural coverage analysis. Furthermore, establishing requirements traceability from low-level requirements to source code, and to object code, supports providing an understanding of software properties commensurate with this second objective. While there are arguably other ways to elicit such properties, this paper, for the reasons previously documented, will restrict discussion to those called out by DO-178B.

For an emulator, the high and low level requirements would generally need to capture the extent of the instruction set, as well as all other supporting functional and non-functional properties of the machine being emulated. For most legacy CPUs, much of this information would need to be extracted from whatever technical documentation is still available. For CPU manufacturers that have been out of business or have since been subsumed into other businesses, it may no longer be possible to elicit much documentation beyond what is already held by the in-service support organisation and associated technical library. Unlike the commercial world where the internet often becomes a repository for obsolete information, rarely does proprietary information relating to obsolete military specific equipment find its way into the public domain. As it is not possible to guarantee that the documentation adequately captures all functional and non-functional properties of the legacy CPU, then it follows that the initial set of high and low level requirements assembled for the emulator might well be incomplete. Therefore the activities, and resultant outcomes of the second aforementioned DO-178B verification objective become especially important as one means of eliciting a complete set of high and low level requirements, not only in the context of the emulator itself, but also in the context of the extant legacy application – in particular, where the legacy application relies on undocumented legacy processor properties. An inspection of DO-178B reveals that activities supporting this objective only start to become applicable at Level C or better, with Level B providing the bulk of necessary activities.

Further to ensuring a complete set of high and low level requirements, the activities and resultant outcomes of the second aforementioned DO-178B verification objective are also necessary to explore properties associated with any underlying components of the emulator that might not directly relate to the execution of instructions, or management of I/O and memory. For example, any monitoring or mapping functions, or failure thereof, should not result in any unacceptable failure conditions.

An extract from the architectural and verification requirement assignment against identified failure modes is presented in Table 3. Full details have not been included on each specific assignment. However, it follows that they are appropriate based on the argument presented earlier in this section.

Detection/Protection	Architectural or Verification Requirement
Mappings to be established between legacy instruction and emulation implementation, and mappings to be verified for functional and temporal equivalence.	<p>Software high level requirements comply with system requirements</p> <p>High-level requirements are accurate and consistent</p> <p>High-level requirements are compatible with target computer</p> <p>Low-level requirements comply with high-level requirements</p> <p>Low-level requirements are accurate and consistent</p> <p>Low-level requirements are compatible with target computer</p> <p>Source Code complies with low-level requirements</p> <p>Executable Object-Code complies with low-level requirements</p> <p>Test coverage of software structure (decision coverage) is achieved</p>
<p>Memory and output post update verification. Memory and output update to explicitly check cycle synchronisation</p> <p>Entry point to decode and execute instruction process limited to following fetch instruction and increments counter process</p> <p>Program counter is to be verified after operation. Program counter update to explicitly check cycle synchronisation</p> <p>Registers are to be verified after operation. Registers update to explicitly check cycle synchronisation</p> <p>Monitoring of pass instruction to decode and execute instruction process to ensure graceful recovery from failure mode</p> <p>Program counter is to be verified after each increment operation. Program counter increment to be explicitly synchronised to fetch instruction.</p> <p>Fetch instruction and increment program counter process to be synchronised with decode and execute instruction process to ensure one decode and execute for each instruction fetched.</p>	<p>Software high level requirements comply with system requirements</p> <p>High-level requirements are accurate and consistent</p> <p>High-level requirements are compatible with target computer</p> <p>Low-level requirements comply with high-level requirements</p> <p>Low-level requirements are accurate and consistent</p> <p>Low-level requirements are compatible with target computer</p> <p>Source Code complies with low-level requirements</p>

Table 3: Determination of Architectural or Verification Requirements for Type 1 Emulator

An inspection of the software assurance activities called out in Table 3, considered in the context of the previous discussion on the DO-178B software assurance model, reveals that these objectives come largely from the set of objectives core to DO-178B Level B. Therefore, it follows that for most safety related systems, the most

appropriate software assurance level will be DO-178B Level B. This is further addressed, later in this paper.

4 Further Examination of Emulation Architectures

This section introduces an extension of the emulator architecture that permits changes to be made to the functionality of the legacy binary using a new development environment with code hosted directly into the native environment.

4.1 Incorporating New Functions Developed in Native Code (Type 2)

A legacy emulation architecture that permits the incorporation of new functions developed in native code (designated Type 2 for convenience of reference throughout this paper) is detailed in Figure 6 and Figure 7.

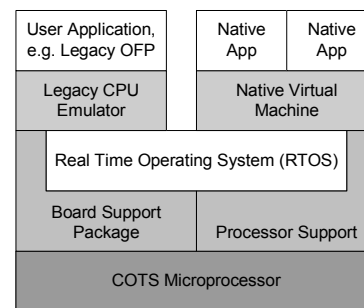


Figure 6: Emulator Architecture (Logical Layers) - Incorporating New Functions Developed in Native Code

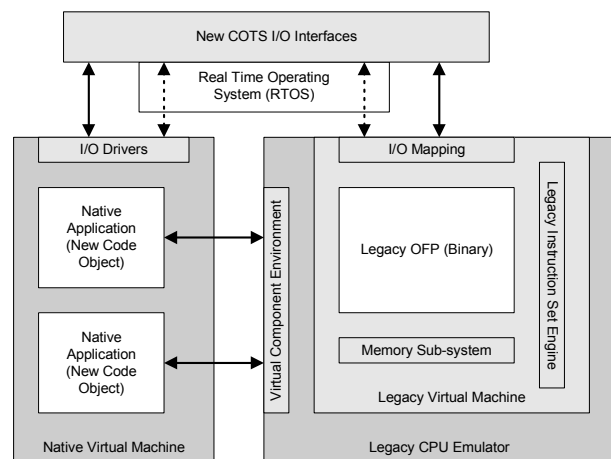


Figure 7: Emulator Architecture (Sub-Elements) - Incorporating New Functions Developed in Native Code

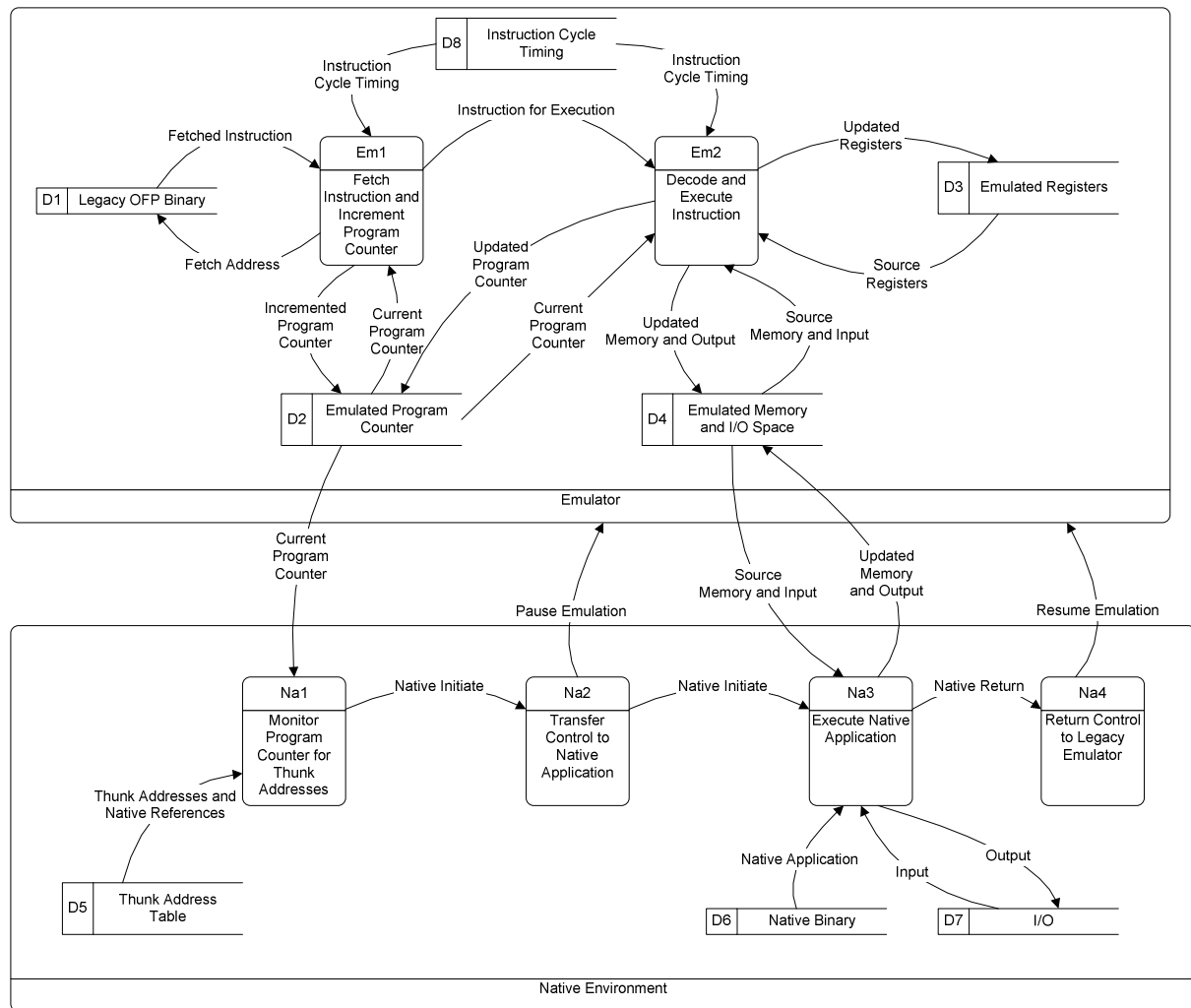


Figure 8: Legacy Instruction Set Engine and Virtual Component Environment Data/Information Flow Diagram

The following paragraphs provide an overview of the emulator components detailed in Figure 7.

Many components within the Legacy Virtual Machine that are common with the Type 1 architecture. As before, encapsulating the Legacy Virtual Machine is the *Legacy CPU Emulator* which provides the interface for the Legacy Virtual Machine to execute in the native processor environment.

However, in this architecture the Legacy CPU Emulator also includes a component labelled the Virtual Component Environment. The *Virtual Component Environment* provides the mechanisms to switch between legacy and new native code environments and share data between them.

Figure 8 describes the relevant data and information flows that might occur in one such implementation of this extended emulation architecture. This example has again been based on the MIPs processor although the logical interpretation is easily extended to any type or class of microprocessor. Figure 8 uses the term ‘thunk’, which is defined as a reference mapping of code addresses from one system specific form (i.e. legacy address space) to another (i.e. native environment).

5 Safety Analysis of the Legacy Emulation Architecture Incorporating New Functions Developed in Native Code

To provide an understanding of the software failure modes that might be relevant to the Type 2 emulation architecture, and importantly what architectural considerations and software assurance activities are required to provide evidence of the absence and handling of these identified failure conditions, it is again necessary to conduct some form of software safety analysis.

In line with the approach adopted for the Type 1 emulation architecture, a SHARD was conducted using the Legacy Instruction Set Engine and Virtual Component Environment Data/Information Flow Diagram (Figure 8) as a reference for information flows that might exist in the emulator, and services that are required from a functional perspective. An extract from the SHARD is presented in Table 4.

Guide Word	Deviation	Cause	Effect	Detection / Protection
Omission	Failure to pause emulation	Programming error causes a failure to recognise thunk address or to recognise need to pause emulation and transfer control to native function Wrong data in thunk table	Emulated program continues to execute beyond thunk address – synch problems with native functions, possible program crash	Establishment of thunk addresses and storage of thunk addresses within thunk table require level of integrity to process Verification of thunk address integrity, and relevant context to application code segment
Commission	Emulation paused when not required	Programming error causes transfer to native application when not required Wrong data in thunk table	Emulated program will pause, with transfer of control to wrong native function, or program halt	Emulated function to be an atomic operation to ensure interruption from thinking only between instructions. Virtual Component Environment shall be able to detect a native applications anticipated violation of the emulated applications real time constraints and deadlines, and be able to return operation to the emulated function gracefully. As for Omission
Early	Emulation paused earlier than required	Programmer error causes transfer to native application earlier than required	Emulated program will pause, with transfer of control to native function early resulting in state synch problems with emulated program	Emulated function to be an atomic operation to ensure interruption from thinking only between instructions. As for Commission
Late	Emulation paused later than required	Programmer error causes transfer to native application later than required	Emulated program may pause, with late transfer of control to native function resulting in state synchronisation problems with emulated program	As for Early
Value	Emulation pauses with wrong state	Programmer error causes program counter, register and memory/output state of emulator to be incorrectly captured	Native function accessing emulated state may perform operations on incorrect data. Return of execution of emulator likely to result in program crash, or operations on invalid data	Emulated function to be an atomic operation to ensure interruption from thinking only between instructions. Transfer control is not permitted access to emulator state unless otherwise justified.

Table 4: Extract from SHARD on Type 2 Emulator (Pause Emulation)

Detection/Protection	Architectural or Verification Requirement
Verification of thunk address integrity, and relevant context to application code segment	Software high level requirements comply with system requirements
Emulated function to be an atomic operation to ensure interruption from thinking only between instructions.	High-level requirements are accurate and consistent
Emulated function to be an atomic operation to ensure interruption from thinking only between instructions.	High-level requirements are compatible with target computer
Transfer control is not permitted access to emulator state unless otherwise justified.	Low-level requirements comply with high-level requirements
Virtual Component Environment must be able to detect a native application's anticipated violation of the emulated application's real time constraints and deadlines (to achieve temporal equivalence as previously defined), and be able to return operation to the emulated function gracefully.	Low-level requirements are accurate and consistent
	Low-level requirements are compatible with target computer
	Source Code complies with low-level requirements
	Test coverage of software structure (decision coverage) is achieved
Establishment of thunk addresses and storage of thunk addresses within thunk table require level of integrity to process	The development of native functions, their effect on the emulated systems state, and the integrity of the overall system are closely linked. Therefore, it may be necessary to apply more rigorous software assurance activities than associated with the severity of failure of the native function alone. Similar software assurance activities may be required as for emulator itself. This is dependant on the nature of the native function. Those that have significant effect on the state of the emulated system are likely to require additional assurance activities (i.e. equivalent to those defined for the emulator). Those functions that don't may be conducted at a software assurance level commensurate with the severity of failure of that function.
Transfer control is not permitted access to emulator state unless otherwise justified.	Protected Domain – Partitioned RTOS

Table 5: Determination of Architectural or Verification Requirements for Type 2 Emulator

Having developed an understanding of the types of failure modes, their causes, effects, and detection/protection means, it is then possible to define architectural or verification requirements relative to those failure modes. Section 3 has already discussed the relationship of the DO-178B software assurance model and the associated critical properties elicited from relevant activities. The

same logic is applied in this case. An extract from the architectural and verification requirement assignment against identified failure modes is presented in Table 5. Full details have not been included on each specific assignment. However, it follows that they are appropriate based on the argument presented earlier in this section.

An inspection of the software assurance activities called out in Table 5 reveals that these objectives again come largely from the set of objectives core to DO-178B Level B. Therefore in a general sense, it follows that for most safety related systems, the most appropriate software assurance level will be DO-178B Level B. This is addressed in greater detail later in this paper. It should also be noted now that a requirement is identified relating to the interaction between the emulator and native environment. A robust means of addressing this requirement is through a protected domain (partitioned) RTOS. A broader inspection of the SHARD analysis, beyond the extent of that presented in this paper, also dictates a requirement for complete isolation of the emulator from the new COTS hardware (including I/O) by means such as the protected domain (partitioned) RTOS.

5.1 Incorporating a Protected Domain RTOS (Type 3)

A legacy emulation architecture that extends the Type 1 architecture to incorporate a protected domain RTOS (designated Type 3 for convenience of reference throughout this paper) is detailed in Figure 9 and Figure 10.

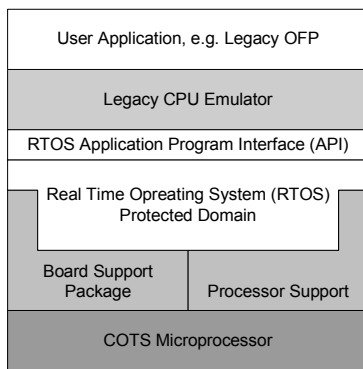


Figure 9: Emulator Architecture (Logical Layers) - Incorporating a Protected Domain RTOS

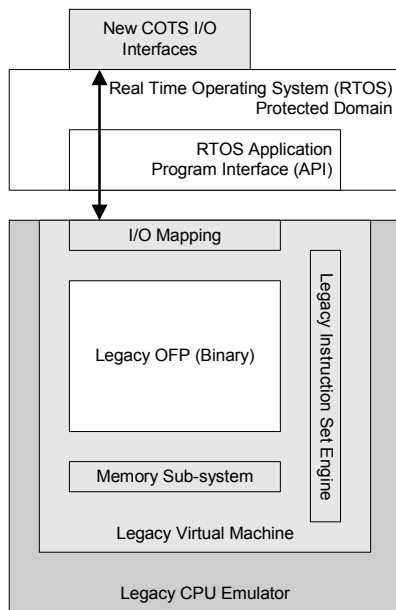


Figure 10: Emulator Architecture (Sub-Elements) - Incorporating a Protected Domain RTOS

The significant change compared with the Type 1 architecture is the complete isolation of the emulator from the new COTS hardware (including I/O) by the protected domain (partitioned) RTOS.

This approach is ideally suited to those emulator applications where there is no immediate requirement to introduce new functionality into the legacy OFP using the native environment (as described in the Type 2 emulator), but for which future capability introduction may be required. The introduction of the protected domain (partitioned) RTOS provides a future expansion capability that ensures it is possible to later introduce new functionality in the native environment, without significant rework of the emulator. For example, changes to the emulator would likely be restricted to the addition of a virtual component environment.

5.2 Emulation Architecture Incorporating New Functions Developed in Native Code (Type 4)

A legacy emulation architecture that extends the Type 2 architecture to incorporate a protected domain RTOS (Type 3 features) that facilitates the incorporation of new

functions developed in native code (designated Type 4 for convenience of reference throughout this paper) is detailed in Figure 11, Figure 12, and Figure 13.

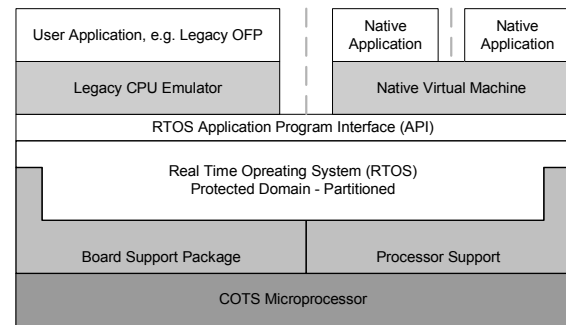


Figure 11: Emulator Architecture (Logical Layers) - Incorporating New Functions Developed in Native Code

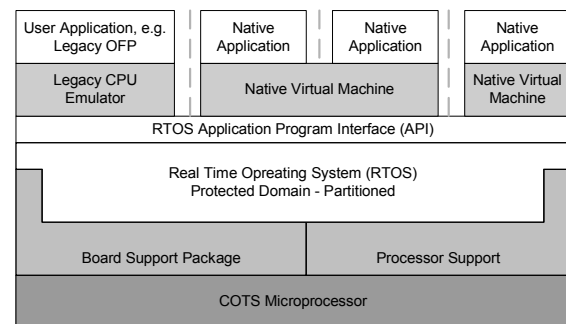


Figure 12: Emulator Architecture (Logical Layers) - Incorporating New Functions Developed in Native Code

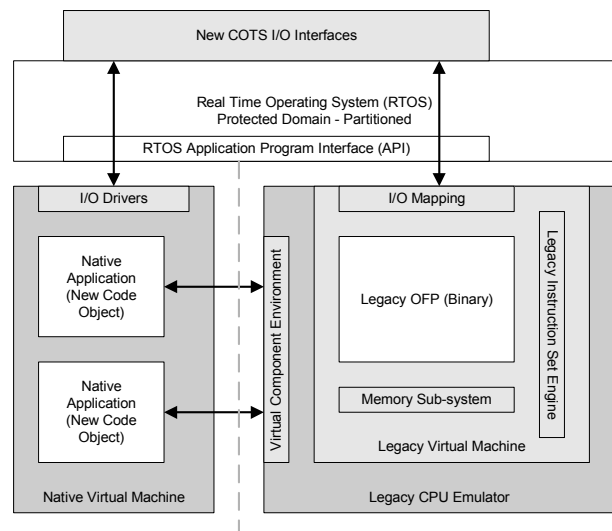


Figure 13: Emulator Architecture (Sub-Elements) - Incorporating New Functions Developed in Native Code

The following paragraphs provide an overview of the emulator components detailed in Figure 13.

The significant change compared with the Type 2 architecture is the complete isolation of the emulator and

native virtual machine from the new COTS hardware (including I/O) by the protected domain (partitioned) RTOS. Furthermore complete temporal and spatial partitioning is now provided by the protected domain (partitioned) RTOS of the legacy CPU emulator and the native virtual machine to ensure adequate separation of legacy and new native functions (represented by the gray dashed line)

While the temporal and spatial partitioning provided by the protected domain RTOS now ensures that the legacy application will not crash as a result of a problem with functions implemented in the native environment, there are some additional architectural issues that need to be addressed. For example, the virtual component environment must now exhibit safety properties to allow the legacy application to continue operating in event the native code fails to return control to the legacy application in a functionally appropriate or timely manner.

6 Recommendations Relating to Emulator Architectures

The analysis conducted in earlier sections of this paper has provided an appreciation of the failure modes that might be associated with the emulation architectures considered. This permits recommendations to be formed on the relevance of particular emulation architectures to the severity of various safety and mission failure conditions. Although this paper is primarily aimed at safety critical and safety related systems, recent guidance in AAP7001.054 Sect 2 Chap 7 (ADF 2005) has provided a framework through which those software assurance activities relevant to safety critical and safety related systems can be applied commensurately to mission systems. Table 6 details the emulation architecture types considered in this paper, and the safety or mission failure conditions for which they are recommended.

Failure Condition		Type1	Type2	Type3	Type4
Safety	Catastrophic	NR	NR	HR	R ¹
	Hazardous	NR	NR	HR	R ¹
	Major	R ¹	R ¹	HR	HR
	Minor	R	R	HR	HR
	No Effect	HR	HR	R ²	R ²
Mission	Critical	R	R	HR	HR
	Serious	R	R	HR	HR
	Important	HR	HR	R ²	R ²

NR=Not Recommended, R=Recommended, HR=Highly Recommended

Note 1: Recommended only if the sub-elements have been subjected to rigorous software safety analysis that shows the absence or handling of all potential failure modes.

Note 2: Recommended rather than Highly Recommended based on the cost associated with the purchase of a protected domain and partitioned RTOS.

Table 6: Emulation Architecture Recommendations

7 Issues with the Native Code Approach

Subsequent modifications to legacy code hosted on the emulator may be made using either the legacy development environment or a newer development environment.

There may be substantial risks associated with making any more than a small number of changes to the system using the newer development environment and native code. This is because of the difficulty of being able to demonstrate precise knowledge of the pre-conditions to modifications from exit points of the legacy code increases with each subsequent change. Similar difficulties might also exist for the post-conditions of modifications and entry points back into the legacy code. These problems are particularly pronounced for legacy software that has limited available documentation (often the case of legacy systems), or where developer's knowledge of the legacy software is no longer sufficient. The problems may be further exacerbated by poor control over the determination of entry and exit points to and from the legacy code, and the amount of coupling permitted between various native code elements. A robust Application Programming Interface (API) is therefore required to provide tight control of the entry and exit points.

Specific architectural considerations, including partitioning (spatial and temporal as provided by a partitioned RTOS), and related analysis would be required to demonstrate finite, well defined dependencies between subsequent new developments and legacy code. Such analysis would require a thorough understanding of the emulator, the legacy software and the legacy processor. Risks associated with adding new code can be mitigated largely by detailed analysis, as suggested throughout this paper, and planning of new features as part of an appropriately controlled and managed change process. Tool support would also be desirable to assist with providing an understanding of the legacy and native implementations.

Some emulators provide embedded real-time, non-intrusive monitoring and legacy code debugging services as part of the virtual component environment or lower-level CPU emulator. Such services may provide developers with tools necessary to mitigate aspects of the aforementioned problems by providing visibility into the entry and exit points across the boundaries between the legacy and native code elements.

One strategy that might also address aspects of this problem is to eventually translate the executive out of the legacy application into the native environment, with the legacy binary being used as a library of functions. NGST has successfully implemented this approach with some other avionics systems, although proprietary and US State Department restrictions prevent disclosure in the public domain. Specific software safety analysis would be required to provide an understanding of any risks with this approach.

The risks identified above must be weighed against the potential cost and schedule benefits offered by emulation, and the risks of alternative software approaches for upgrading systems.

8 Software Assurance Evidence Requirements for Emulation

The ADF preferred standard for software assurance of airborne software is RTCA/DO-178B (ADF 2005). Although it is acceptable to develop emulation within the framework of other relevant software assurance and software safety standards, this paper will restrict the provision of certification criteria to DO-178B. Comparisons to other standards may be developed through consideration of the critical software assurance activities identified in this paper.

Table 7 defines the DO-178B software levels relevant to emulation based on those critical software assurance activities identified in previous sections of this paper. The levels are determined by a comparison of those critical software assurance activities with those activities normally prescribed by DO-178B at the respective software levels defined in that standard.

Failure Condition		DO-178B Software Level	Software Level for Emulation
Safety	Catastrophic	Level A	Level A
	Hazardous	Level B	Level B
	Major	Level C	Level B
	Minor	Level D	Level C
	No Effect	Nil	Level D
Failure Condition		AAP7001.054 Guidance	Software Level for Emulation
Mission	Critical	Level C	Level B
	Serious	Level D	Level C+
	Important	Nil	Level D

Table 7: Software Levels for Emulation

Table 8 details other additional activities required for Level C+, over those activities required for Level C. These activities largely mirror those specific Level B activities identified in the earlier analysis that are critical to meeting and verifying detection/protection requirements, and meeting the desired level of integrity for the system. Where independence, as defined by DO-178B, is believed to provide further assurance to the satisfaction of the relevant DO-178B objective, then a requirement for it has also been documented. Similarly, where independence is not viewed as a key contributor to the outcome of the activity, then it is documented as not required.

DO-178 Reference	Objective
A-3-1 (6.3.1a)	Software high level requirements comply with system requirements (satisfied with independence)
A-3-2 (6.3.1b)	High-level requirements are accurate and consistent
A-3-3 (6.3.1c)	High-level requirements are compatible with target computer (satisfied with independence)
A-4-1 (6.3.2a)	Low-level requirements comply with high-level requirements (satisfied with independence)
A-4-2 (6.3.2b)	Low-level requirements are accurate and consistent (satisfied with independence)
A-4-3 (6.3.2c)	Low-level requirements are compatible with target computer
A-5-1 (6.3.4a)	Source Code complies with low-level requirements (satisfied with independence)
A-6-3 (6.4.2.1, 6.4.3)	Executable Object-Code complies with low-level requirements (satisfied with independence)
A-7-6 (6.4.4.2a, 6.4.4.2b)	Test coverage of software structure (decision coverage) is achieved (independence not required)

Table 8: Level C+ Additional Activities Over Level C

Although such prescription detailed in Table 7 departs from the traditional hazard severity / software level alignment of Aerospace Recommended Practice (ARP) 4754 and DO-178B, emulation technology presents specific architectural risks that require specific assurance activities to mitigate. Developers might argue that the increase in software assurance level for emulation will significantly increase the costs associated with the introduction of emulation technology. While there is an element of truth to this argument, there are a number of key points that provide an appropriate tradeoff against the cost increase. These are as follows:

- The size of the emulator (in terms of lines of code) will generally be only a small proportion of ‘real world’ legacy binary (lines of code) for military avionics equipment (e.g. emulator’s lines of code is less than 25% ‘real world’ legacy binary). A typical ‘real world’ legacy binary in currently operating Australian military aircraft is of the order of 150,000 lines of code, although future aircraft and systems will continue to see this figure increase. These figures are based on the RePLACE and DGU example, and are considered typical of such implementations.

Therefore, the number of lines of code to which the more stringent software level should apply is not substantial, and certainly less than the legacy binary. It is important to note that the guidance does pertain to the emulator only, and not to the legacy OFP (binary). This table does not imply that the legacy binary should be redeveloped to the prescribed software level.

- The service history of the legacy software will be yielding a perceived software failure rate or rate of problem occurrences. This will be interpreted by operators both in terms of the reliability or availability, and thus the capability integrity, of the associated system; and also the inherent level of safety currently provided by the system. Service history is one important attribute as it is unlikely that most legacy systems will have been developed with the requirements of most current software assurance or safety standards in mind. Reflecting on the software failure rate, it is generally argued by the software community that such a rate is not actually a reliability (ie. reliability normally being a measure of a systems susceptibility to random failure conditions, whereas this is more synonymous as a measure of the software's exposure to conditions that might uncover systematic errors). However, it does provide a baseline to operators as to the 'apparent reliability', and 'level of safety' of their avionics equipment. Importantly, it also provides technical support staff with an understanding of the software's contribution to any identified failure modes. Therefore, it should be the goal of any program addressing the equipment obsolescence to provide properties commensurate or better than those experienced on the original legacy systems. This places some specific integrity requirements on the emulator. For example, the emulator should not introduce any further failure modes that might reduce the 'apparent reliability' or 'level of safety' of the system. Furthermore, benign failures should remain benign, or be handled by the emulator. One means of achieving this is to apply a greater level of rigour, appropriately targeted, to the emulator than for that required of the original legacy binary, thus providing a greater level of integrity in the emulator software. An appropriate, targeted increase in the software level for the emulator therefore justifies the applicable cost increase.
- The software assurance level, and associated prescription/definition of activities for Minor, No Safety Effect, and Mission Important categories is not significantly greater than the level normally defined under normal circumstances for these systems. Therefore, these systems provide a suitable entry point for the technology into the military avionics domain.

9 RePLACE Dual Instruction Set Computer (DISC)

DSTO are presently working with NGST to develop a concept demonstrator utilising NGST's RePLACE Emulation Technology for the RAN Seahawk DGU. The

Dual Instruction Set Computer (DISC) variant of RePLACE, as distinct from other RePLACE variants (eg. X-Port and hybrid), has been identified by NGST as most applicable to emulating the DGU's AAMP processors. This identification is based on consideration of the AAMP processor's performance against a proposed native processor (ie. PowerPC), with due consideration for the RePLACE variant's computational overhead. It is therefore necessary to examine RePLACE DISC in the context of the guidance already formulated in this paper.

9.1 Overview of RePLACE Architecture

Figure 14 details the architecture of the RePLACE DISC. By inspection it is possible to determine that it closely represents the Type 2 architecture already covered in this paper.

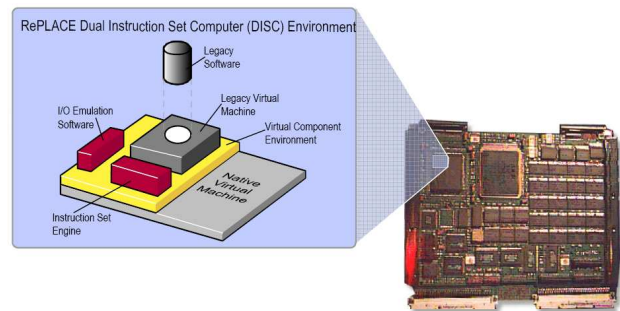


Figure 14: RePLACE Dual Instruction Set Computer (NGST 2005)

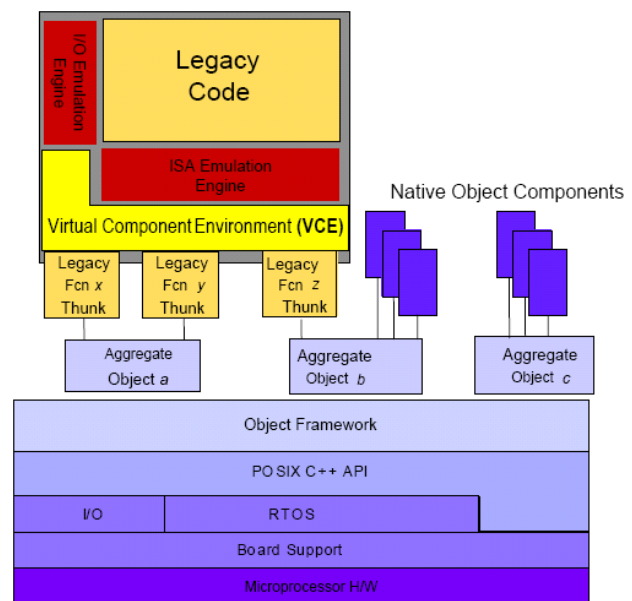


Figure 15: RePLACE DISC (Logical Layers) (NGST 2005)

9.2 Assessment of RePLACE

DGTA funded a US based company, Certification Services Inc (CSI), under a DGTA standing offer to conduct a DO-178B audit of the RePLACE program. The audit considered both DO-178B Level B and C, with a specific goal to identify the practicality of applying such objectives to the RePLACE development, and to assess

any issues (technical, cost, schedule) that might exist in transitioning an existing program within a framework that would meet the objectives identified throughout this paper. CSI are highly skilled in evaluating the application of DO-178B to avionics developments, and specifically Mike DeWalt, whom conducted this audit, is considered an authority on DO-178B.

The audit (CSI 2005) found that the RePLACE development does not yet satisfy all relevant DO-178B objectives, and although the RePLACE program in its current state is mostly close to satisfying both Level B and Level C, many of those objectives that it does not yet address are considered essential in this context. Those objectives not presently addressed predominantly relate to requirements traceability, verification and some software configuration management activities. It is important to note that RePLACE Seahawk DGU program is presently being conducted as a concept technology demonstrator, and therefore satisfaction of many of these objectives is beyond the scope of funding available in such programmes.

CSI's assessment is that there is little technical risk of the RePLACE program not being able to meet DGTA's expectations with respect to avionics software assurance. However, cost and schedule risk were identified relating to the generation of software assurance artefacts and the rework necessary to ensure that RePLACE fully meets the relevant DO-178B requirements. However, it was assessed that through some targeted certification risk reduction activities, it is possible to constrain cost and schedule risks to suitable levels.

Following the audit, DGTA published a series of papers on how the Commonwealth might accept RePLACE as part of the design acceptance process used for modifications to Australian State aircraft (DGTA 2005). These papers formed the starting point of further negotiation and development with NGST and DSTO. The guidance in these papers was principally based on the analysis which forms the background of the material presented in this paper.

Post-audit work conduct between DGTA, DSTO and NGST, which is still on-going, has recently resulted in NGST delivering a white paper that demonstrates a qualified understanding of cost and schedule risks. Furthermore, DGTA assesses that the identified cost and schedule reflect that emulation is a cost effective option for addressing legacy obsolescence in some safety related and mission systems. Further details relating to cost and schedule are commercially sensitive and cannot be discussed further in this paper.

RePLACE for the Seahawk DGU program is presently hosted on the Wind River VxWorks OS, a non-DO-178B compliant RTOS. VxWorks was selected for the DGU emulator demonstrations due to the high cost of other DO-178B compliant RTOS's, and the limited funds available for the Seahawk DGU emulation demonstrator. There is some work NGST would be required to undertake to modify any system calls and software structure within this implementation of the RePLACE application to accommodate a different RTOS. Other

RePLACE products have already been hosted on protected domain RTOS's, indicating that there is unlikely to be any technical barriers to moving to a protected domain RTOS (eg. Green Hills Integrity OS) for the Seahawk DGU RePLACE application.

10 Summary

Evaluation of emulation technology, through exploration of several emulation architectures and of RePLACE as a case study, has allowed DGTA to define certification and regulatory guidance for the development of emulation technology within the ADF context. The trial application of this certification guidance with the Seahawk DGU RePLACE concept technology demonstrator has permitted an evaluation of the effectiveness of the prescribed DGTA certification criteria. At this time DGTA is satisfied that this guidance will promote an acceptable level of safety for emulation on legacy military avionics while still ensuring emulation is a cost effective option for addressing legacy obsolescence.

11 Acknowledgments

I would like to thank Systems Certification and Integrity (SCI) – DGTA staff including Mark Wade, Squadron Leader Ben Musial and Flight Lieutenant Wendell Fox for their input to and review of all my work relating to emulation.

I would also like to thank Paul Vicen, Tamy Staub, Curt Pflasterer and other RePLACE staff at Northrop Grumman Space Technology (NGST) for their input to this paper relating to RePLACE and enthusiasm to explore certification criteria with DGTA.

Finally I would like to thank Dr Rob O'Dowd, Mark Davies and David Culpin of Air Operations Division – Defence Science and Technology Organisation for their coordination of RePLACE development activities with NGST, and their on-going liaison with DGTA.

12 References

The following documents, papers and publications are referenced throughout this paper. A number of these documents are not available in the public domain for propriety or confidentiality reasons. Readers wishing to seek further information should direct their queries to the author of this paper, or the relevant standards body.

- Aerospace Recommended Practice ARP4754 (1996) Certification Considerations for Highly Integrated or Complex Avionics Systems, Society of Automotive Engineers.
- Australian Defence Force (2005) Australian Air Publication (AAP) 7001.054 Airworthiness Design Requirements Manual AM1.
- Australian Defence Force (2006) Aircraft System Safety Engineering Course – Software Safety Course Notes developed jointly by Systems Certification and Integrity – DGTA and Ball Solutions Group.

- Certification Authorities Software Team (2000) Position Paper Cast 5 – Guidelines for Proposing Alternate Means of Compliance to DO-178B, Federal Aviation Authority.
- Certification Services Inc (2005) Evaluation of the NGST RePLACE Product, CSI Document 05-276-1246 Rev03.
- Directorate General Technical Airworthiness (2005) Paper on How the Commonwealth Might Accept RePLACE – Issue 3, Australian Defence Force.
- Federal Aviation Authority (FAA) Order 8110.49 (2003) Software Approval Guidelines, USA.
- McKinlay, A. (2001) Software Safety Course Notes, Aviation Safety, School of Engineering, University of Southern California, USA.
- Northrop Grumman Space Technology (2005) RePLACE Technology – Bringing 20th Century Systems into the 21st Century - Marketing Brief, Dayton Ohio, USA.
- Pumfrey, D. (1999) The Principled Design of Computer System Safety Analyses, PhD Thesis, Department of Computer Science, University of York, UK.
- RTCA Inc (1992) RTCA/DO-178B Software Considerations In Airborne Systems and Equipment Certification, Washington, D. C. USA.
- RTCA Inc (2001) RTCA/DO-248B Final Report for Clarification of DO-178B Software Considerations in Airborne Systems and Equipment Certification, Washington, D. C. USA.

Safety, Software Architecture and MIL-STD-1760

Matthew John Squair

Senior Safety Consultant
Jacobs Australia
GPO Box 1976, Canberra, ACT 2601
Matthew.Squair@defence.gov.au

Abstract

Integrating modern aircraft stores, particularly weapons, creates a complex system of systems challenge. The traditional approach to such integrations was for each to be a stand-alone program. For each program a unique interface would usually be implemented, usually also with a set of unique problems, such as the missile ‘ghosting’ problems experienced during the F-16 to AMRAAM integration (Ward 1993). In response to the problems of such an approach *MIL-STD-1760 an Interface Standard for Aircraft to Store Electrical Interconnection System* was released by the US DoD to standardise aircraft/store interfaces. This paper discusses the advantages and limitations of the architectural techniques of MIL-STD-1760. A hierarchical method for integrating the use of the standard into a safety case is also described.

Keywords: Safety, architecture, software, MIL-STD-1760.

1 Introduction

1.1 Architecture, bus design and integration

Unfortunately no singular agreed definition of what constitutes a software architecture exists, for example:

“From a safety viewpoint, the software architecture is where the basic safety strategy is developed for the software” (IEC 61508), or

“In avionics (an architecture is) a representation of the hardware and software components of a system and their interrelationships, considered from the viewpoint of the whole system” (STANAG 3908)

ARP 4574 goes further to relate architectural design patterns to measures of connectivity (Table 1-1). For this paper architecture is defined as the large scale description of system components, their interactions, connectivity and the principles and guidelines that ensure a balanced system design and evolution. Thus, although traditionally typified as a protocol or ‘bus’ design issue, any decision to select a bus protocol is also a decision that affects the central architectural principles of a distributed system (Rushby 2001).

Copyright © 2006, Australian Computer Society, Inc. This paper appeared at the 11th Australian Workshop on Safety Related Programmable Systems (SCS’06), Melbourne. Conferences in Research and Practice in Information Technology, Vol. 69. Tony Cant, Ed. Reproduction for academic not-for-profit purposes permitted provided this text is included.

Architecture design pattern	Connectivity Concept
Partitioned design	Decoupling
Dissimilar, independent designs implementing a function	Decoupling Independence Redundancy
Active/monitor parallel design	Hot redundancy
Backup parallel design	Cold redundancy

Table 1-1 ARP 4754 Architecture Examples

1.2 Software architecture and safety

The concept of architecture is useful from a software safety perspective as it can be used to impose a separation of concerns between decisions about the architecture of a software artifact versus the implementation. Partitioning the design space in this way supports the development of safety arguments in a modular and hierarchical fashion. Such partitioning can also clarify organisational interfaces between collaborating development teams; another traditional area where safety issues can arise.

Because of their abstract nature, software architectures can also support the re-use of well understood and proven architectural solutions. Such re-use allows the construction of safety arguments based upon the continuity of design rather than solely upon the unique attributes of a specific implementation. Thus the re-use of architectural design patterns moves software engineering closer to traditional engineering disciplines where safety arguments are based in large part upon the provenance of the design. Such arguments are similar to, but more archetypal, than the ‘product service history’ argument of DO-178B (RTCA/DO-178B 1992).

1.3 The MIL-STD-1760 interface standard

A modern store is intended to be highly interoperable¹ with multiple aircraft. Such a store is usually also an independent system evolving at a pace independent of the carriage aircraft. As such, a modern store (and its carriage aircraft) satisfies the majority of Maier’s criteria for what constitutes a system of systems (Maier 1996) and it is for this systems of systems domain that MIL-STD-1760 was developed.

¹ Two systems are interoperable if each can conveniently benefit from the resources (capabilities or information) of the other.

MIL-STD-1760 is an open, published and non-proprietary standard intended to maximise both interoperability and safety. While open in this sense, actual designs are typically closed with data transfers and associated deadlines determined during design rather than at run time. The standard applies an architecting strategy which is both collaborative (programs may re-use other programs efforts) and directive (by specifying minimal design requirements) in nature (Meyer 1998). An analogy would be a city planner enforcing a building code rather than an architect enforcing a particular design solution. As a result of this strategy, safety is an emergent attribute at both a system of systems and individual aircraft to store integration level.

2 Military aircraft and weapon system safety

2.1 The military aircraft domain

Military aircraft may launch or jettison stores, fly at supersonic speeds, execute high-g manoeuvres whilst carrying out safety critical functions including:

1. Store inventory,
2. Interruptive Built In Test (BIT),
3. Store rack unlocking,
4. Selective or emergency jettison firing,
5. Fire/release/launch sequencing,
6. Arming (both immediate and preset),
7. Fuzing (both delay or mode),
8. Weapon yield selection,
9. Output stage selection (store/pylon/station), or
10. Initiating irreversible functions (pyrotechnics).

In this operational environment achieving functional reliability is a significant technical challenge. Unfortunately achieving high reliability in these circumstances does not automatically equate to safety. In fact increasing reliability can decrease safety (Leveson 1995), requiring us to balance the need for reliability against safety. Other challenges of this domain include:

1. limited weapon operational histories,
2. hard real time requirements²,
3. irreversible safety critical processes,
4. dynamic interface and network topologies,
5. mode rich safety critical behaviour, and
6. multiple configurations of store and platforms.

To safely and reliably perform such functions aircraft and store must coordinate their actions dictating a dependable communication channel robust enough to withstand the harsh operational environment.

² In itself the requirement for hard real time behaviour can be a challenge to the MIL-STD-1553 data bus which has traditionally been used for soft real time system applications.

3 Stores management systems design

3.1 Architecture

3.1.1 General architecture

Most modern combat aircraft utilise a dedicated stores management system to control stores and the support and release equipment associated with them. Two basic architectural patterns are presently used in aircraft stores management systems, centralised or distributed. A centralised architecture is generally used in situations where the stations to be controlled are closely located and inter-station wiring is therefore a minimal system overhead. Distributed architectures, as Figure 3-1 illustrates, allocate stores management functions to physically separate Store Station Interface Units (SSIU) while minimising interconnects by multiplexing signals over a data bus. Since it forms the basis of most modern stores management systems, a distributed architecture will be used as the reference design for this paper.

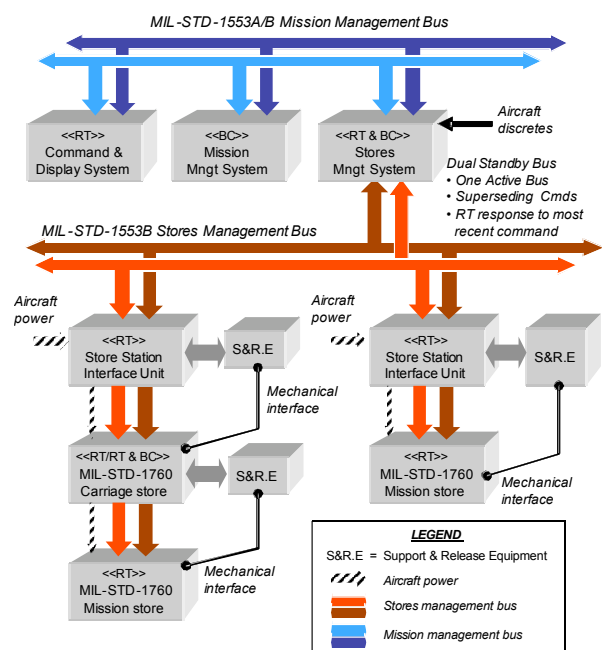


Figure 3-1 A distributed stores management architecture

3.1.2 Communication bus protocols

MIL-STD-1760 constrains the MIL-STD-1553B multiplex bus communication standard to a single-master polling protocol with only one node, the Bus Controller (BC), in charge of communication on the stores management bus, as well as solving and administering Remote Terminals (RT) access conflicts and errors that may arise on the bus. Due to its simplicity, this protocol makes analysis and monitoring of communication easier than split bus control (also supported by MIL-STD-1553B). The command response structure of the protocol is well suited to reactive system application and provides a bound on latency of communications which is important for real time systems. Disadvantages of the protocol are that a Single Point of Failure (SPOF) is introduced, identification of RTs during initialisation is required and communication overhead is increased by the need for a

command/response message pair for each set of data (Sivencrona 2001)³.

Communication protocols can be event or time-triggered, with event-triggered protocols usually applied when there are a large numbers of discrete signals transmitting messages in a pseudo-random or sparse time base form. Since stores management is inherently reactive and task driven in nature, event triggered protocols are obviously well suited to this application. Another argument for the event driven communication protocol (adopted by MIL-STD-1760) is that time triggered protocols introduce frame delays which in turn translate to errors in weapon delivery. A final advantage is that unlike a time triggered protocol there is no need to coordinate a schedule of transmissions amongst distributed components⁴.

MIL-STD-1553B is a relatively slow speed interface (1Mbps) compared to modern standards, such as FlexRay which runs at 10Mbps (FlexRay 2005) or Time Triggered Protocol (TTP/C), running at 25 Mbps (TTTech 1999). However the slower bus speed does make the bus more resistant to noise from signal reflections and ambient RF noise.

3.2 Safety coordination issues

3.2.1 The general coordination problem

Representations of data are consumed and produced by aircraft and store functions to perform the required mission. However, for this representational system to work, both the store and aircraft need a common first and higher order set of *expectations* about their own behavior as well as about the expectations and behavior of the other (Lewis 1969). Therefore, in order to coordinate safety critical behaviour there is a need to define and manage a common, unambiguous set of conventions (or protocol) about the production and consumption of safety critical data. Such conventions can also assist in constraining the use of error prone semantic constructs.

3.2.2 System of systems coordination

At the system of systems level, this coordination problem traditionally expresses itself as a problem of backward and forward compatibility across versions of particular data formats. For example MIL-STD-1760's definition of the Most Significant Bit (MSB) for 2's complement entities was changed between Revision B and C then subsequently changed back in Notice 1 of Revision C. These changes led to some developers interpreting the first bit as a 'signed' bit for a signed number while others used a 2 complement number format. More generally the options introduced in successive revisions have generated a family tree of possible implementations of the standard. To further complicate matters, standards invoked by MIL-STD-1760, such as MIL-STD-1553B, have themselves evolved over time. Other examples of

coordination issues at this level are the differing safety requirements of various weapons communities, i.e. nuclear versus non-nuclear safety.

At the system of systems level, a safety argument must therefore demonstrate that any differences between revisions of the standard do not introduce hazards or if they do, these interactions are identified, excluded or controlled.

3.2.3 Store integration coordination

At the store integration level the coordination problem expresses itself as false assumptions about behavior on the other side of the interface. For example in one aircraft 'initiate battery' commands were implemented as irreversible because traditional batteries were one shot thermal devices. However for the store being integrated the batteries were NiCad and the command needed to be reversible. Since this was an unstated assumption of the interface it was not identified during development, becoming evident only during integration testing. This example illustrates the necessity to explicitly state all behavioural expectations and consequently drives MIL-STD-1760 to be much more than a set of data-grams.

At the store integration level, a safety argument must be supported by a demonstrable claim that a common and well understood set of data conventions are being used on both sides of the interface.

3.2.4 Real time distributed control

The real time and distributed control nature of stores management systems also introduces the problem of how to ensure coordinated safe behaviour within such a system. For example, launching stores from stations on one side of an aircraft can induce highly hazardous asymmetric loads, whilst simultaneously releasing stores can cause hazardous g-jump effects; both of which can lead to overstressing the aircraft. For the communications bus, this problem devolves to how to send commands in a timely, safe and reliable fashion when faced with arbitrarily long communications delays⁵ over a communications channel.

At the level of real time distributed control, a safety argument must demonstrate (expressing the problem in functional integrity terms) that individual functions are safe and independent, or if they do interact their coordinated behaviour is safe.

4 The MIL-STD-1760 standard

4.1 The MIL-STD-1760 interface

MIL-STD-1760 defines implementation requirements for the Aircraft/Store Electrical Interconnection System (AEIS) in aircraft and stores. This interconnection system provides a common interfacing capability for stores on aircraft, and a hierarchical depiction of the:

1. electrical (and optical) signal interfaces,

³ In MIL-STD-1553 a 2 word command and 1 word status response sequence consist of 32 data bits and 53 overhead protocol bits. In CANBUS a nominal message of 32 data bits has only 43 protocol bits.

⁴ For example by implementing a global (distributed) clock.

⁵ For example delays from internal processing or file transfer events.

2. physical umbilical and connector interface, and
3. logical interface, comprising the:
 - a. communications architecture,
 - b. message content and formatting, and
 - c. data transfer protocols.

MIL-STD-1760 establishes the interrelationships between aircraft and store interfaces and the interfaces at different store stations on an aircraft. As Figure 4-1 illustrates, there is a dynamic hierarchical relationship between Aircraft Store Interface (ASI), carriage store Interface (CSI), Carriage Store-Store Interface (CSSI) and mission store (MSI) interfaces. The standard defines the requirements for both a primary and auxiliary interface, however this paper will only discuss the primary interface since it is the one most often used. Several interface classes of varying capability are also defined in MIL-STD-1760 for the ASI, however for clarity this distinction will be omitted because the safety critical signals are common to all classes.

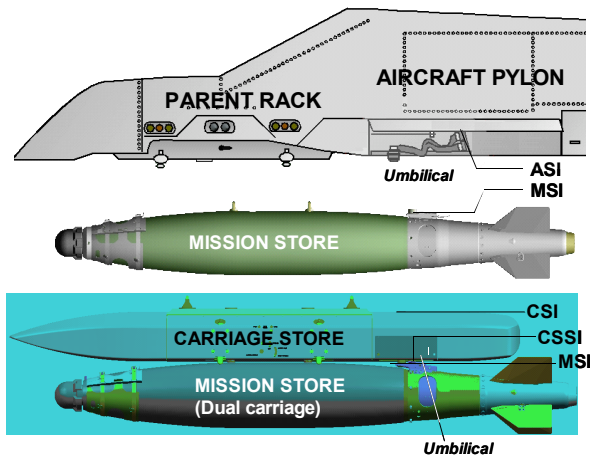


Figure 4-1 Physical interface hierarchy

4.2 The electrical interface

4.2.1 Primary signal set

The primary MIL-STD-1760 interface signal set, as shown in Figure 4-2 comprises redundant data bus signals, high and low bandwidth signals, dedicated discrete signals, fibre optic signals and aircraft power. MIL-STD-1760 nominates certain signals as safety critical signal interfaces and these are discussed in the following sections.

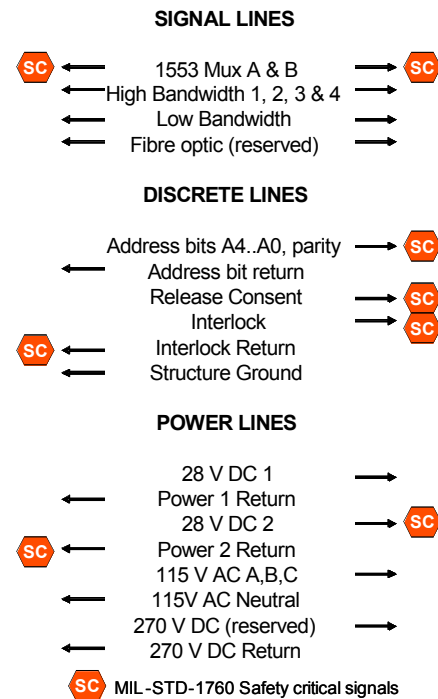


Figure 4-2 MIL-STD-1760 primary signal set

4.2.2 MIL-STD-1553B interface

The Aircraft Store Interface (ASI) MIL-STD-1553B interface consists of Mux buses A & B transformer coupled stubs, carrying the MIL-STD-1760 message set.

4.2.3 Store address interface

These fixed address and parity bit discrete signals are used to indicate the RT address that a store should use to identify itself to the BC.

4.2.4 Release consent interface

Perhaps better termed 'safety critical consent' this discrete release consent signal grants consent to the store to act on safety critical commands received over the MIL-STD-1553B data bus.

4.2.5 Interlock interface

The ASI interlock interface is used by the aircraft to monitor the electrically mated status of the interface connector between stores and aircraft so as to allow (as an example) determination whether a successful launch has occurred.

4.2.6 28V DC No. 2 power interface

The 28V DC No.2 interface is used to power safety critical functions, for example firing a squib or thermal battery.

4.3 The physical interface

As Figure 4-3 illustrates, the physical 1760 interface comprises the umbilical cable and connectors making up the electrical interconnect between aircraft and mission store. Harsh carriage environments and repeated

operational sequences of mating and de-mating connectors mean that hardware failures can be expected.

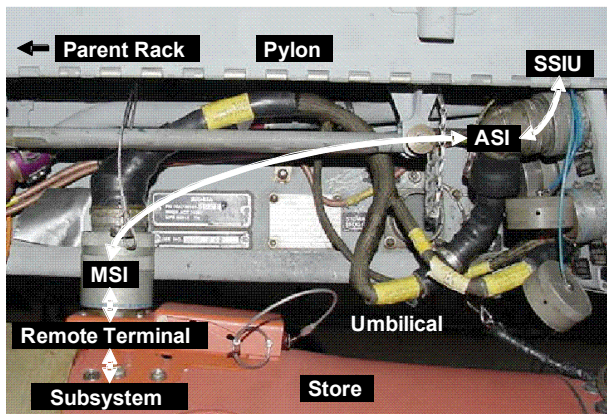


Figure 4-3 Physical MIL-STD-1760 interface

4.4 The Logical Interface

4.4.1 Message Data formats

MIL-STD-1760 invokes the MIL-STD-1553B digital communication bus standard (Revision B Notice 4) and for safety further constrains certain of MIL-STD-1553B's logical operations. MIL-STD-1760 also defines standard message formats, called sub-addresses (S/A), for store control and monitoring functions. Messages not specified can be developer defined, usually in an Interface Control Document (ICD), those currently specified are:

1. Store Description (1T),
2. Aircraft Description (1R),
3. Store Control (11R),
4. Store Monitor (11T),
5. Nuclear Stores Control (19R & 27R),
6. Nuclear Stores Monitor (19T and 27T),
7. Reserved (08) may be used for test,
8. Mass Data Transfer Control message (14R), and
9. Mass Data Transfer Monitor message (14T).

5 MIL-STD-1760 logical interface

5.1 Open System Interconnection Model

The MIL-STD-1760 logical interface can be mapped to a means/ends hierarchical arrangement of operations and mechanisms (SAE AS-1B3 2002) in a similar fashion to the OSI Basic Reference Model, (ISO/IEC 7498-1). Such a mapping provides a service layer architectural model of the MIL-STD-1760 logical interface and a convenient framework to assist in identifying the safety attributes of the protocol. As Figure 5-1 illustrates, the Real System Environment (RSE) encompasses the complete Aircraft-Store logical interface. Within the RSE, the aircraft and store application processes exchange data using the services of the MIL-STD-1760 Open System Interconnection Environment (OSIE). The resultant

layered MIL-STD-1760/1553B interfaces can be typified as forming a bi-directional and open architecture.

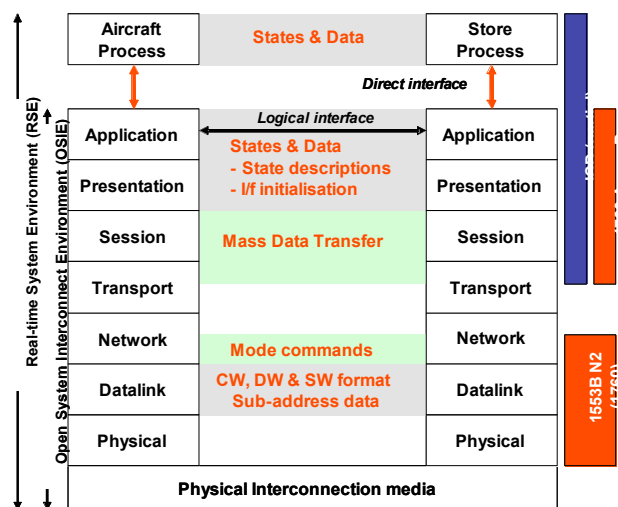


Figure 5-1 MIL-STD-1760 OSI map (SAE AS-1B3 2002)

5.2 Application process

MIL-STD-1760 lists a set of 173 standard data entities for use. These entities are used by the OSI application process while their defined syntax maps to the OSI presentation layer. While application processes are not specifically defined in MIL-STD-1760 generic applications can be inferred, such as those in Table 5-1. MIL-STD-1760 does not address all aircraft-store processes, some of which will be store unique, such as developmental test functions, and some which are aircraft housekeeping functions such as post launch cleanup.

Application User Process	MIL-STD-1760 Clause
Power application	5.2.12.2
Identification (Aircraft/Store)	B4.2.2.3, B4.2.2.6
Built In Test (BIT)	Table B- XXVI
Mission data transfer/initialisation	Table B- XXVI
GPS initialisation	Table B- XXVI
Transfer alignment/conditioning	Table B- XXVI
Release/launch/jettison sequence	Table B- XXVI
Control and monitor	Table B- XXVI

Table 5-1 MIL-STD-1760 Processes

5.3 MIL-STD-1760 OSI environment

5.3.1 Application layer

The application layer is normally defined by the particular store developer and documented in the ICD.

5.3.2 Presentation layer

MIL-STD-1760 defines the following presentation layer protocols for data transmission syntax and semantics:

1. standard store control, monitor and description message data words formats,
2. application specific data word formats, and
3. data entity syntax tables referenced by the Data Entity List.

Application specific message data word formats are normally documented in the store to aircraft ICD and can include time, alignment, GPS time marks, moment arm and non-critical store control/monitor messages.

5.3.3 Session layer

MIL-STD-1760 provides a connection-mode session layer service by exchanging Mass Data Transfer (MDT), Transfer Control and Transfer Monitor messages. Service can be further partitioned into classes of application specific data such as targeting, GPS almanac/ephemeris, weather or imagery. Lower level Protocol Control Information (PCI) is also represented in this layer (SAE AS-1B3 2002).

5.3.4 Transport layer

The transport layer provides a connection mode MDT service using Transfer Data messages which implement record and block numbering for file segmenting and sequence control. Above this layer data can be exchanged as files or messages; but below it data is only exchanged in message form (SAE AS-1B3 2002).

5.3.5 Network layer

There is little provision of network-services in MIL-STD-1760/1553B, the network layer PCI consists of header and identifier, address confirm, message sub-addresses, mode codes, word count, T/R flag and the RT status word. S/A 7⁶ is reserved by MIL-STD-1760 for message peeling to provide a growth path to connection or connectionless modes of message routing and relay. However, to comply fully with OSI network-service requirements further protocols would need to be added for a specific integration⁷ (SAE AS-1B3 2002).

5.3.6 Data-link layer

The MIL-STD-1553B protocol merges the lower part of the OSI Network Layer, the Data Link Layer and the Physical Layer together and does not subdivide into Network and Data Link functions (SAE AS-1B3 2002). The datalink layer provides connectionless services, i.e. messages are sent without establishing a dedicated connection between RT and BC. RT addresses define the data-link address space with RT 31 (broadcast) providing a common data-link address. Word parity, MIL-STD-1760 message checksum, control authority words and synch bits also comprise the data-link layer.

5.3.7 Physical layer

In MIL-STD-1760, the MIL-STD-1553B frames are represented by Manchester bi-phase waveforms with tailored source and receiver end characteristics.

6 MIL-STD-1760 and Safety

6.1 Safety requirements

MIL-STD-1760 imposes no global safety target for a store/aircraft interface and such requirements are usually defined in other documents, such as an aircraft or store system specification. For discussion purposes an example of such requirements is given in Table 6-1.

Failure event	Co-effect/Mode	Probability
Launch failure on command	Launch state (store ready awaiting launch command)	10^{-4} /cmd
Inadvertent launch	Normal (master arm switch safe)	1×10^{-7} / hr
Inadvertent launch	Tactical (master arm switch armed)	1×10^{-5} / hr
Inadvertent launch	Launch state	1×10^{-3} / hr

Table 6-1 MIL-HDB-244A safety requirements

The different inadvertent jettison probabilities of Table 6-1 reflect that safety constraints are progressively removed as the aircraft system approaches the launch point (MIL-HDB-244A 1990). This balances within a mission profile the competing requirements of safety and mission reliability.

MIL-STD-1760 also requires that the probability of inadvertent generation of a valid critical control word with a valid critical authority word and a data field requesting critical action, should not exceed 1×10^{-5} per flight hours per data field combination.

Since ideally the reliability of the communication bus should not be a dominant source of system failure, the reliability of the communication channel should also be set least two orders of magnitude lower than the probability of launch failure i.e. 1×10^{-6} per launch event.

The system integrator is responsible for developing a design which meets the overall safety requirements whilst ensuring that design requirements for the aircraft store interface do not exceed the safety claim limits of MIL-STD-1760. An integral part of this design must therefore be to ensure that software, hardware and environmental faults do not result in the top level events through a mixture of fault avoidance, elimination and tolerance.

6.2 Fault avoidance/elimination

Reducing the complexity of an interface can reduce the likelihood of unintended interactions and error propagation. By restricting MIL-STD-1553B to a master/slave command and response protocol MIL-STD-1760 enforces simple and deterministic behaviour across the interface. This also makes it easier

⁶ MIL-STD-1553B terminology for a specific message.

⁷ For example the International Space Station (ISS) 'boxcar' protocol that provide broadcast and individual asynchronous communication using a major/minor frame based on 1553 messages (Hyman 2003).

to verify safe behaviour, such as whether responses are bounded in the time domain. In comparison, a CANOpen safety protocol interface would require verification that its safety related data can be transmitted within a specific timeframe across a bus allowing both synchronous (time based) and asynchronous (event based) messages. The determinacy of the MIL-STD-1760 protocol is further enhanced by the elimination at design time of the use of ambiguous data.

The MIL-STD-1760 protocol also offsets, to some degree, the disadvantages of using an event-triggered protocol for a safety critical application. While detecting errors and building fault-tolerant mechanisms for time-triggered communication protocols is easier because more 'a priori' knowledge exists as to their behaviour, the master/slave protocol does provide a more predictable communication protocol than other event triggered protocols such as bus contention or token passing.

The simplicity of the protocol also supports self test and diagnosis functions which in other protocols can be more problematic. For example, should an RT respond with an incorrect address the BC will detect the error within one command/response cycle. Should the same problem occurs in a CANBUS network however, no inherent method exists that allows identification of the source of a message sent with the wrong identifier.

6.3 Fault tolerance

6.3.1 Fault tolerance strategies

To balance the competing requirements for both reliability and safety of section 6.1 MIL-STD-1760 adopts two parallel fault tolerance strategies:

1. to assure reliable service by a redundant fault tolerant design, and
2. to assure safe service by a 'fail silent' error recovery strategy.

It is important to note that requirements for fault tolerance may also introduce additional and complex asynchronous behaviour which may exhibit even higher proportions of requirements related design faults than mission functions (Lutz 1993, Mackall 1998). Again the simple MIL-STD-1760 protocol and MIL-STD-1553B 'cold' bus redundancy scheme reduces complexity and to some degree the likelihood of introducing subtle side effect hazards into the design.

6.3.2 The fault hypothesis

The next logical question is what assumptions can be made about the number and type of faults that the integrated system will be able to tolerate? This set of assumptions is termed the fault hypothesis⁸ for the system. A well formed fault hypothesis should also identify which faults are not covered and for which recovery strategies are needed. Developing a fault hypothesis as part of an integration program is important,

since this allows a systems integrator to evaluate existing fault tolerance mechanisms on both side of the interface for assumption coverage (i.e. verifying that hypothesis accords with reality). Although MIL-STD-1760 does not explicitly state a formal fault hypothesis its inclusion of fault tolerance mechanisms can assist the system integrator in developing such a hypothesis. For example, it is possible to derive from the standard a possible physical fault hypothesis as follows:

1. an RT forms a single Fault Containment Region (FCR) that can fail in an arbitrary way,
2. the physical network and BC form a single FCR that can fail to distribute messages or distribute messages in error,
3. RT hardware timeouts will translate babbling idiot temporal failures on the bus to fail silent,
4. Error detection is performed by both RT and BC, but fault recovery is managed by the BC,
5. The BC will detect a possible error of the RT within one command/response cycle and confirm in two cycles,
6. The BC translates detected RT failures to fail safe by powering down the station and transitioning to another, and
7. The system can recover from a single store failure within an application dependent time.

Another example is the fault hypothesis that can be derived from the assumptions made by MIL-STD-1553B as to the RF noise environment (impulsive noise) and the tests designed to simulate this environment (a worst case white Gaussian noise). This noise hypothesis is a key factor in evaluating the effectiveness of low level data redundancy, i.e. checksum and error detection codes.

6.3.3 Redundancy and independence

Fault tolerant behaviour depends in large measure upon redundancy, although not always upon component replication. Because of this, the independence of redundancy becomes a critical issue for fault tolerant design. If components are dependent in some way, then hazards can arise where a common attribute can cause the failure of supposedly independent components. As an example, dependencies can be introduced by shared hardware or data providing propagation channels for side effect type interactions, which can then lead to cascading failures (Jaffe 1999). Independence can also be violated by shared environments (common cause failures) or common component design and failure modes (common mode failures).

A complete architectural safety argument therefore needs to consider the direct failure of components, the independence of redundancy structures and whether common cause hazards are present. The difficulty is that these common cause hazards can occur in many different guises and the more complex the system the more difficult it is to identify them during design.

⁸ A subset of the general design hypothesis stating all assumptions upon which the design is based.

6.3.4 Error propagation and detection

A fault that propagates, either as an error in data or an incorrect control output, will violate the assumption of independence essential to redundancy. To protect against this situation MIL-STD-1760 requires both time and value error detection mechanisms. MIL-STD-1760's master/slave protocol ensures that a BC will detect an error of an RT within notionally one cycle. By excluding direct RT to RT communications the protocol also eliminates direct RT to RT error propagation channels. Babbling idiot style error propagation is dealt with by the MIL-STD-1553B requirement for an RT to shut itself down if it exceeds its maximum message length, a primitive form of distributed bus guardian in the time domain. The BC is independent of the RT usually also with internal dual bus redundancy and due to its separate development also a degree of design diversity. A layered set of protocol checks provide a defence in depth against message transmission corruption.

6.3.5 State restoration/error recovery

Having detected a data error there are two possible approaches to handling such invalid data:

1. store and tag the data as being invalid, or
2. discard the data completely.

The first option requires a subsequent decision as to the utility and safety of the data while the second option needs to address what effects data senescence will have upon reliability and safety. Where data that is detected as being invalid is used, the system can become vulnerable to propagating state erosion due to transient faults (i.e hardware, RF noise or Heisenbugs (Gray 1985)). MIL-STD-1760 applies the second policy, requiring that all error data be discarded. Upon detecting an error, the RT must then flag error by withholding the status word and letting the BC decide what to do next. This reduces the potential for state erosion, maintains bus timeliness and provides a predictably safe response to abnormal environmental events. However, a discard policy can introduce data staleness and data loss rates proportional to the rate of transient faults.

6.3.6 Diagnostics and fault tolerance

One of the problems introduced by fault tolerance schemes is that they may also mask symptoms required to identify faults. While diagnostic capabilities are supported by the protocol via RT subsystem (optional) and terminal (mandatory) status flags, one shortcoming of the standard is that MIL-STD-1553B as invoked requires RTs to ignore invalid commands⁹ thus preventing multiple concurrent responses¹⁰. However this also introduces a bus diagnostic shortcoming because the error

is not fed back to the BC. Similarly illegal commands¹¹ may be optionally detected by the RT but if that option is not implemented the error cannot be reported to the BC.

6.4 Safety architecture design patterns

6.4.1 Inoperability design pattern

Although not explicitly stated by MIL-STD-1760 one of the key design patterns of weapons systems safety is that of maintaining the store in an inoperable (unarmed) state that is incapable of carrying out the unsafe action. A store that is neither armed nor activated a store is significantly less hazardous, especially if it is exposed to abnormal environments such as a lightning strike (Spray 1994).

6.4.2 System level redundancy pattern

At the highest level of system architecture, an aircraft will usually carry redundant stores to perform a mission. This provides system level redundancy and allows a faulted store-station to be isolated and the task transitioned to a stand-by station. By providing this redundancy the overall probability of mission failure drops to 1×10^{-4N} per launch intent where N = the number of stations (neglecting common cause failures) and the system level effects of a store level fail silent strategy are minimised.

At the system level the response is to ensure that an affected node is safely stopped and reconfigure the system to provide similar or degraded service, rather than attempting to mask the error and continue mission functions with that particular node. However the use of redundancy comes at the price of increased complexity of behaviour. For example if an RT fails to respond the BC would need to switch between buses to confirm that both channels are silent before declaring the RT as failed.

6.4.3 Homogenous redundant design pattern

The MIL-STD-1553B bus is a dual redundant cold standby hardware architecture intended to ensure reliability of service in the presence of random hardware failures. Drawbacks to this pattern are recurring cost, the BC as a potential SPOF, vulnerability to common cause/mode failures and the consumption of additional system resources (such as power and cooling).

6.4.4 Dissimilar redundancy design pattern

Redundancy is implemented at each level of the MIL-STD-1760 system design from architecture (monitor/actuator pattern) through to the layers of the logical interface. This approach provides an inherent defence in depth approach with a high degree of dissimilarity and independence. Figure 6-1 illustrates the redundancy introduced into the logical interface by the combined use of MIL-STD-1553B and MIL-STD-1760 protocol checks including MIL-STD-1553B synch and parity bits, MIL-STD-1760 critical control flags, critical

⁹ An invalid command is one containing one or more invalid words (for example due to a parity bit failure) but with a valid address.

¹⁰ Compared to CANBUS (ISO 11898) where an error in the last-but-one bit of a CAN frame may cause inconsistent message duplicates or omissions.

¹¹ Illegal commands are commands that have passed the validation test but are not part of the RT's capability.

authority polynomial error codes and message checksums.

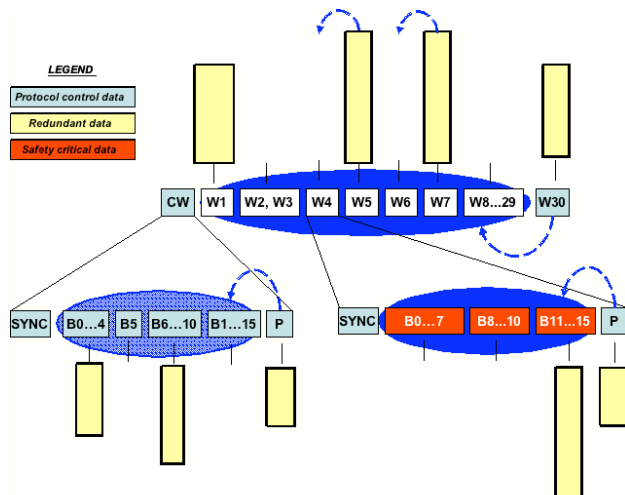


Figure 6-1 MIL-STD-1760/1553B combined protocol

One question that arises when implementing MIL-STD-1760, is how far back should redundant data be taken? For example, should a Commit to Separation command also be stored as a single bit in memory or should it be stored redundantly?¹². The issue of where to stop applying redundancy (an instance of the general stopping rule problem for modification programs) is a significant issue in legacy system integrations where processor and memory constraints may constrain the representation of data.

6.4.5 Monitor/actuator control channel pattern

The deliberate separation of command and consent signals by MIL-STD-1760 forms a diverse redundancy safety pattern (Douglass 1999). Here the control interface is divided into a control channel (the MIL-STD-1553B command interface) and a monitor channel (the release consent interface). As Figure 6-4 illustrates the control channel provides initialisation data and a 'launch' command for safety critical function while the monitor channel prevents firing except when independent consent is achieved. This pattern supports the fail silent safety strategy.

The monitor/actuator channel of MIL-STD-1760 offers the advantage of a lightweight dissimilar redundancy pattern that can protect against SPOF faults, environmental or common cause faults. Using a dissimilar redundancy pattern also makes safety analysis and verification easier by reducing the level of criticality of the control channel software from notionally safety critical¹³ to safety related¹⁴, thereby reducing verification requirements and cost (McDermid 2001).

¹² For example storing data normally and in ones complement form or utilising multi-bit representations to increase the hamming distance between safe and unsafe values.

¹³ A single member of a fault tree minimum cut set.

¹⁴ A member of a fault tree minimum cut set greater than one in size

6.4.6 Control & authority independence

MIL-STD-1760 specifies two pairs of MIL-STD-1553B words for the transfer of safety critical data, Critical Control 1 and 2. Each word has in turn an associated Critical Authority word. Critical Control words contain the actual safety critical information while the Critical Authority words contains a polynomial code check (i.e. redundant data) on the data bits of its associated Critical Control word.

To achieve a false software command rate of no greater than 1×10^{-5} per flight hour required by the standard independence must be demonstrated between control and authority word processing, or conversely the impossibility of a common cause failure. Figure 6-2 illustrates an architectural design pattern where critical signals are segregated and combined as late in the channel as possible to ensure their independence. In this pattern the BC derives the authority from an independent authority generator which is also interlocked to the launch consent switch (MIL-HDBK-1760). Additional assurance via dissimilar design could be achieved by implementing the authority table in firmware.

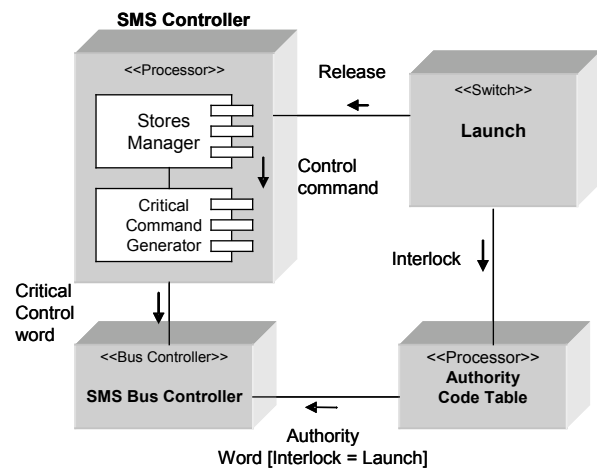


Figure 6-2 Independent control & authority generator

In comparison the CAN open safety protocol (DSP 304 2001) uses serial redundancy where safety critical data is transmitted in two independent messages. The data in the second message is bit-wise inverted and is crosschecked after re-inversion with the first message in the receiver. However, as DSP 304 invokes no requirements for independence of these messages it does not provide similar protection against common mode/cause failures.

6.4.7 Firewall (segregation) design pattern

MIL-STD-1760 applies an architectural design pattern of physically segregating critical signals from non critical ones in order to minimise the likelihood of hazardous interaction. Safety critical power, data and signals are physically, functionally and temporally separated from non critical mission power, data and signals to minimise inadvertent interactions that may invalidate an independence assumption. This segregation is carried out at both the architecture and implementation levels of the interface.

The pin layout of the primary interface connector surrounds the Release Consent pin with other pins that are at nominal 0 V potential. This ensures that during a fault condition where the Release Consent pin shorts to its neighbour, the neighbouring pin will not act as an enable signal. This physical segregation scheme should also be adopted within the aircraft internal connectors, which can be a significant stopping rule issue for legacy system integration.

Ideally a common set of messages would be used across all RTs on the bus thereby enforcing data segregation. But if non store equipment is on the same bus as MIL-STD-1760 stores then these equipments may use the particular safety critical words for non-critical information thereby degrading data segregation. In this case the design of the BC to meet MIL-STD-1760 becomes more complex. For this reason the standard discourages this implementation, unfortunately this can also be a significant program coordination issue when integrating MIL-STD-1760 stores to a legacy aircraft.

The standard also requires that release consent and safety critical power be functionally segregated i.e. not be generated or consumed by non safety critical functions. The resulting procedural cohesion of these two interfaces ensures that any interlocks or reasonableness checks performed need only deal with a small set of use states.

Both release consent and 28V DC are also segregated temporally from signals by strictly limiting the availability of release consent and power to the period immediately prior to the store receiving its safety critical command. Such temporal cohesion also simplifies the implementation of reasonableness checks.

6.4.8 Physical (spatial) proximity pattern

MIL-STD-1760 recommends that the supplier of safety critical signals be as physically close to the signal consumer as possible to minimise the potential for hazardous EMI effects in long cable runs. This safety pattern places the generation of critical signals as close to the actual interface as possible thereby reducing the likelihood of inadvertent interactions.

6.4.9 Signal complexity pattern

The MIL-STD-1553B digital communication standard invoked by MIL-STD-1760 introduces complex digital signals as a means of communication information. In an environment where excluding noise could only be achieved at prohibitive cost, the complexity of these signals reduces the likelihood that the environment might spontaneously generate them. By implementing such signals in a monitor/actuator control channel the likelihood of spontaneous generation of a critical signal by the environment is reduced again. While this reduces design and verification requirements it also requires that the system be able to discriminate signals from noise, have rejection logic immune to such environments, provide predictably safe responses to abnormal environments and be verifiable (Leveson 1995).

A tradeoff in this MIL-STD-1553B complex signal design pattern is that the higher the signal rejection

threshold is set then the higher the Signal to Noise Ratio (SNR) must also be achieved the required Bit Error Rate (BER). To reduce this effect a relatively high threshold can be set for non-message time to reject line noise then a reduced threshold for Manchester bit sampling time can be used to reduce the SNR and while achieving the required BER.

6.5 Address interface safety attributes

Stores are required to latch address line signals upon startup rather than continuously reading them as it is fairly common to experience interconnect wire or pin damage in flight; with the potential for an invalid RT address to be used by the store with hazardous consequences. As the store verifies terminal address values in the Control Message (S/A 11) using this signal, a failure of the address lines could result in the store responding to a message addressed to another RT. MIL-STD-1553B therefore requires that if the store detects a failure it refuse to respond to any commands to it or to another RT and to not set the terminals address incorrectly.

Because MIL-STD-1553B requires protection against a SPOF in the address interface an odd parity bit is implemented for detection purposes. While notionally a parity bit will only detect 50% of $N > 1$ bit errors this is based upon the assumption of symmetric errors (i.e. logic 1 or 0 errors are equally likely). In practice shorts to ground or to another wire are more likely (logic 1) than open circuit (logic 0). As this reduces the likelihood of combined logic 1 and logic 0 errors the performance of the single parity bit is improved.

To further protect against address faults, addresses can be selected so that there is at least a >2 bit difference between them to increase the hamming distance to 3. For legacy integrations this may be difficult to achieve as address lines may not include a full five bit plus parity set of signals and the carriage of multiple stores may require the use of RT addresses with closer hamming distances. Figure 6-3 illustrates this problem by showing the growth in address requirements across one aircraft's life.

Fixed addresses are used in the standard rather than variable addresses so as not to degrade the safety of the system when transferring safety critical information on the MIL-STD-1553B bus. One example of a hazard that variable addressing introduces is that of sending the message to the wrong RT because an address has been dynamically re-assigned.

Store address signals may also be used by a store to monitor for the aircraft's presence. However due to the potential for SPOF in the address lines, the absence of an address signal cannot be said to be logically equivalent to an Aircraft Not Present signal and must be considered ambiguous. For example a hung store could interpret the removal of the umbilical during the subsequent download as a separation leading to aero-surface deployment on the ground. MIL-STD-1760 therefore requires the address signal to be interlocked with another signal if used to trigger a safety critical transition.



Figure 6-3 F/A-18A/D load growth 1970's to 2005

6.6 Interlock interface safety attributes

The interlock signal is used by aircraft to verify the presence or absence of the store. However due to the potential for SPOF's in the interlock lines (e.g. broken wire, bent pin or contaminated/damaged contact) and the inherent ambiguity¹⁵ of the signals open value it is not of itself a safe indication of the state of the interface. MIL-STD-1760 therefore constrains the implementation such that an Open Interlock signal cannot be used as the sole trigger for safety critical functions. Two examples illustrate:

1. When an aircraft inventory function uses this signal the inventory function must also use another independent signal (i.e. rack pistons extended or hooks open); or
2. When an aircraft launch function imposes dead-facing of the connector prior to release it *cannot* use this signal as loss of interlock occurs during, not before, the de-mating sequence.

6.7 Release consent interface safety attributes

The standard recommends that when release consent is in use its actuation should be visible to the aircrew as a deliberate action. This constrains the design of the release consent signal to be 'not software generated, only steered' (MIL-HDBK-1760) as Figure 6-4 illustrates.

In accordance with the standard's architectural requirement to functionally segregate signals the store must not use the release consent signal to activate any internal store mode or function *except* those modes or functions required to accept or reject safety critical messages received by the store's RT. This enforces procedural cohesion of safety critical interface functions. Generation of the release consent will normally also be safety interlocked to ensure that operator error does not become a SPOF. These interlock can include manual master arm switches, environmental checks such as Weight Off Wheels signals or interlocks such as Bay Door open.

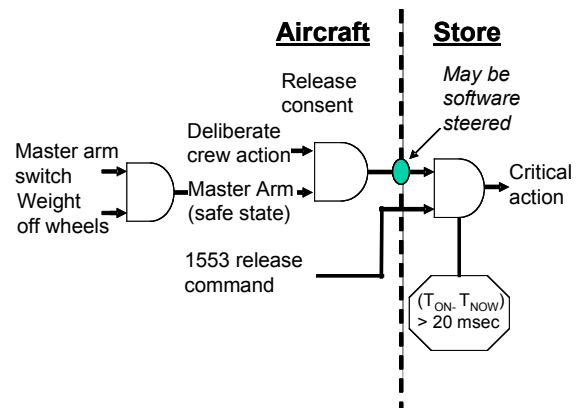


Figure 6-4 A notional release consent logic

Another architectural decision is the functional allocation of the release consent generator function. The function is usually distributed within the AEIS to place signal generation as close to the ASI as possible, so as to minimise the effects of EMI upon long cable runs. However this introduces the need to distribute a digital release consent signal to the SSIU and with distribution the issue of how this is transmitted across the stores management bus. In these circumstances a master arm interlock signal may be added downstream of the release consent steering scheme to provide additional assurance.

6.8 28V DC No. 2 interface safety attributes

In accordance with the standard's architectural requirement for functional segregation, 28 V DC No 2 must only be used for the powering of safety critical functions. Note that when power is applied, the hazard state of the store is increased because 28 V DC No. 2 is used as the *energy source* for carrying out safety critical functions received over the data bus not because it is used as a safety critical command.

To avoid inadvertent actuation the standard suggests that a store should interlock the internal use of 28V DC No. 2 power with the reception of release consent signal or other internal state. This is not a requirement of the standard but a recommended best practice (MIL-HDBK-1760). Similarly on the aircraft side, the standard recommends that 28V DC No. 2 should be interlocked with an aircrew operated switch (i.e. Master Arm) to prevent actuation unless there has been a positive action by the aircrew. As an example, in one incident, a dual rail launcher using a switched power bus was commanded to switch from the launched missile side to the remaining unlaunched missile side. However due to software timing error, the bus was not de-energised in time and the other rail mounted missile was accidentally fired. This interlock addresses the hazard of non-deterministic and potentially un-safe responses to an input received at un-expected times (Jaffe 1989).

Dead-facing of the 28 V DC No. 2 interfaces during disconnect was also initially recommended as a requirement for MIL-STD-1760C, but was eventually discarded because of concern that safety interlocks in some nuclear weapons could be dependent on power at the connector up to the instant of disconnect. This is illustrative of the general coordination problems when

¹⁵ Rail launched missiles, such as the AIM-120, where the umbilical is retracted prior to separation are an example of this inherent ambiguity.

trying to integrate stores with varying safety requirements at a system of systems level. Similar to the release consent signal the allocation of 28 V DC No. 2 power function final switching should be close to the ASI. Likewise if a master arm power relay interlock is implemented the interlock should also be placed as close to the ASI as practical.

6.9 MIL-STD-1553B databus dual redundancy

The use of dual redundant bus eliminates the MIL-STD-1553B bus as a SPOF (ignoring the BC) but as Figure 6-5 illustrates, if implemented as a linear bus, each bus remains vulnerable to common cause failure. For greater failure tolerance, the nodes can be connected in a star arrangement to a hub¹⁶. However this is still not perfect as the star topology takes up additional space and can still suffer common cause failures at choke points, such as connectors or umbilical's.

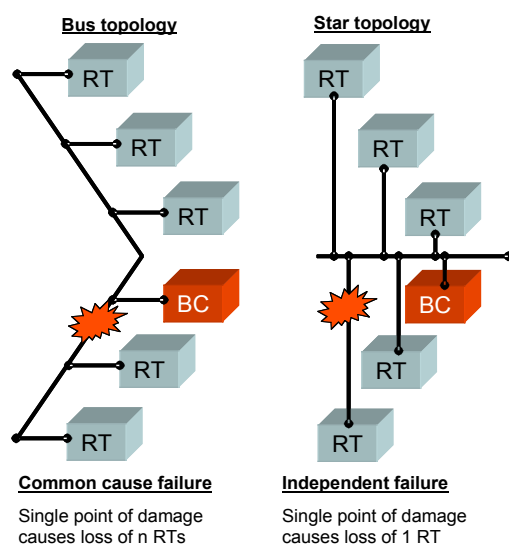


Figure 6-5 Bus versus star topology

6.10 Logical interface safety attributes

6.10.1 Application layer

Application processes are usually specified by the store designer and documented in an ICD in the form of states, functions and data exchanges. Integrator and store developers must implement fault tolerance for these application processes, typically (at this layer) for incorrect value and timing errors. For example in response to the detection of a failure to transition to the launch state (indicated by a store failing to provide a Commit to Store Separation (CTSS) signal) the aircraft could respond by declaring a fail against that store, removing aircrew display cuing, isolating power to the station and transitioning to the next priority station.

Clear standardised data formats with a defined syntax addresses the coordination hazards arising from system of system integration. However one of the drawbacks of MIL-STD-1760 is that these data formats are not derived from a generic set of completeness criteria such as that developed by Jaffe and Leveson (Jaffe 1989) as a result individual implementations may have more (or less) complete and consistent specifications.

Coordinate frames are a common area where data conventions may become inconsistent; as a result Appendix B of MIL-STD-1760 provides a standard set of definitions. However, these definitions should then also be applied consistently across aircraft internal interfaces. For example on one weapon integration program, the Mission Computer team used z-axis alignment coordinates reversed from that of the Station Interface Processor team. The error was not discovered until flight test and resulted in the missile failing transfer alignment before launch.

Another frame of reference problem arises when altitude data must be interpreted by aircrew. Because altitude can be referenced to either the earth geoid or ellipsoid of revolution reference frame different frames can apply simultaneously for target, waypoint or release altitudes depending on their derivation. When displaying altitude data an explicit reference frame should therefore be identified to avoid ambiguity. The consequences of not doing so are the potential for human operators planning a flight path that terminates in 'ground clobber' or loss of line of sight for data-linked weapons.

A traditional technique used to address the coordination of multiple stores and SSIU behaviour is to restrict the concurrent execution of safety critical activities. This scheduling technique takes advantage of the fact that bus traffic is time base sparse and tends to cluster around store state transition events such as store initialisation or release. To minimise inadvertent and potentially hazardous interactions, the multiplexing of store state changes is constrained to periods outside the safety critical sequence. This de-confliction eliminates the design problem of guaranteeing timing for a sequence of commands between store and SSIU when interleaved with other non-safety critical messages of arbitrary length. Non critical store state changes can also be time multiplexed to spread, and so restrict, the level of activity and as an additional benefit, peak load increases will be constrained as additional stores are added to the bus.

Another coordination problem arises from the differences between various aircraft architectures. One aircraft type may utilise the stores management bus to communicate with SSIU's (F-16 or F/A-18) and consume a significant amount of bandwidth while another may only have stores on the bus (F-15). This can be a significant issue with legacy aircraft integrations and means that the safety of one store integration cannot be directly inferred from the safety of an integration onto another aircraft.

In order to carry out certain store functions safely the store may need to know the aircraft type and/or carriage station on which it is located. One example of this need is when different wing unfold timelines are required for

¹⁶ These are not the only possible topologies, for example the IEEE 1394 Firewire protocol (used on the JSF to connect the Vehicle Management System (VMS) to Remote Data Concentrators (RDCs)) allows signals received on one node to be propagated to all others, allowing daisy chain and tree topologies.

different store stations, i.e. bomb-bay versus external pylon. For safety this data should be explicitly provided rather than the store relying on assumptions or side effects to establish the aircraft configuration. For example in one integration a weapon originally design for carriage on a single platform utilised that platforms RT address to determine whether it was carried on a left or right shoulder station and therefore what active separation algorithm to utilise. As RT addresses do not automatically map to specific stations across platforms utilising this scheme on another aircraft could have led to hazardous behaviour. Because of this potentially hazardous ambiguity, the aircraft ID message was added at Revision B of MIL-STD-1760.

Similarly store identity is provided to the aircraft so the aircraft can implement appropriate flight control laws¹⁷, perform automated inventory checks, display the stores load-out to aircrew and appropriately control the store. Stores must be accurately described in the Store Description message so that an aircraft will not misinterpret the store's identity and therefore use incorrect (and hazardous) parameters. This is such a safety critical issue that an inventory confirmation function is often implemented, usually using a heterogeneous set of input data that may include:

1. Store description message data,
2. mission planning data,
3. 'store aboard' discretes, and
4. aircrew entered weapon inventory codes.

The use of such dissimilar data provides greater safety by minimising the likelihood of common cause failures such as human error. MIL-STD-1760 supports this functionality through the store description message and the provision of an interlock signal that can be mapped to an aircraft's 'store aboard' discrete. The derived need to define a common set of weapon and country codes across multiple programs is another example of a coordination style safety issues.

MIL-STD-1760 also defines a set of generic weapon states, which are illustrated in Figure 6-6. These states provide a logical partitioning of store functions and associate functional availability with controlled state changes and their triggers. While the intent of this definition is to encourage logical and deterministic behaviour inadvertent functional interactions can be introduced when the store state requires functions not previously provided in a particular aircraft state, for example providing transfer alignment data to an air to ground store whilst the aircraft is in another mode.

Having defined such states and transitions, as a general safety strategy, the presence of many paths to safe states and few paths to hazardous states are preferred. Ideally there should also be both hard and soft failure modes¹⁸ on

paths to hazardous states, while for a fail safe system the paths to safe states should have no hard or soft failure modes (Leveson 1995). MIL-STD-1760 implements such a strategy, as Figure 6-6 illustrates, through the use of release consent and command authority words for safety critical commands and redundant signals when initiating critical sequences. To satisfy this general strategy a store designer must incorporate implementation specific safety constraints upon transitions, for example prohibiting transition out of the abort state if the preceding transition was from the launch state.

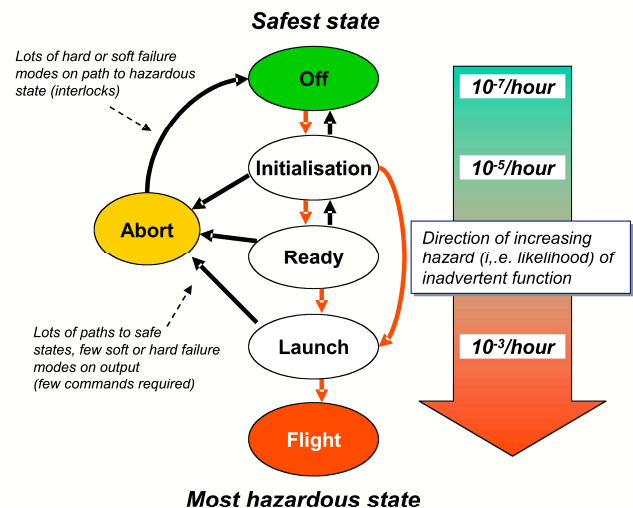


Figure 6-6 MIL-STD-1760 state chart

The standard also requires that store state changes occur only when the contents of a message command a state change and not simply because message receipt was detected. This rule enhances safety by outlawing reliance on 'shortcuts' that could lead to non-deterministic and potentially hazardous behaviour. While such shortcuts may be safe within the one integrations context, the underlying assumptions upon which they are based are not defined in the standard and therefore may prove false in another integration program.

The MIL-STD-1553B concept of illegal commands is also extended in MIL-STD-1760 to include those states where a command that the RT *can* implement could cause a hazardous condition. For example certain commands (such as the reset remote terminal or initiate self-test mode commands) may be legal (and safe) during ground or factory tests, but could cause hazardous effects in flight. Stores may use a 'weight-on-wheels' signal or other discrete hardware input to lockout these commands for incompatible states.

MIL-STD-1760 finally requires the use of redundant outputs to initiate safety critical store state transitions thereby eliminating the possibility of a SPOF triggering a hazardous state transition. This redundancy increases the state transition path robustness (Leveson 1995), i.e. loss of a Release Consent or Master Arm interlock signal will always prevent a store transitioning to launch. MIL-STD-1760 also specifies a 20 msec hysteresis delay

¹⁷ Store mass and drag properties can be extremely critical for dynamically astable high performance aircraft.

¹⁸ A soft failure mode is where Loss of ability to receive 'X' could inhibit 'transition to Y' output, while a hard failure mode is where loss of ability to receive 'X' will always inhibit 'transition to Y' output

to allow for the store to respond to a commanded state change. This delay ensures that the aircraft will not attempt an unsafe timing of a store command sequence.

6.10.2 Presentation layer

Although the standard's accompanying handbook dismisses data formats as having no direct effect on safety, they do play an important role in addressing the coordination type hazards associated with integrating aircraft and store. To safely do so a common understanding of each of the data variables passed is needed. The clear definition of data formats is a fundamental fault avoidance strategy that reduces the likelihood of both syntactic and semantic type design faults occurring.

MIL-STD-1760 defines the store control (11R) and Store Monitor (11T) messages which must be used for safety critical implementations. These messages maintain the fire-wall design pattern of separating safety critical data from other mission data requirements thereby minimising the possibility of inadvertent interaction hazards. The use of dedicated messages for safety critical data has also been applied in other protocols, see for example the CANopen protocol's Safety Related Data Objects (SRDO) concept (DSP 301 2001).

Two words (Critical Control 1 and 2) are used for the transfer of safety critical data as one bit flags. The segregation of safety critical data into these two words supports the firewall design pattern by the transmission of safety critical data in a highly cohesive form that decouples it from non-critical data. Each control word has an associated Critical Authority word containing a polynomial code check on its associated Control Word. For single bit safety critical flags, a polynomial code check included in the message increases the hamming distance between correct and incorrect messages. Using polynomial codes only for the critical control words is a trade-off of processing overheads associated with code calculation versus the error detection requirements for data represented as a single bit.

The Critical Control words also contains, an Identifier field (set to carriage or mission store type) and an Address Confirm field (set to match the address discretised logic) allowing BC detection of an invalid address response by an RT. Stores discard any message found to contain a critical control word that fails one or more of the protocol checks and only enable safety critical processes demanded by critical control words passing the control check.

Invalidity flags are used to indicate that some data should not be used temporarily. For example transfer alignment could be temporarily disregarded because of the launch of launcher rotation or because of a degraded navigation system. The application process handles the generation of the flag by the aircraft and how it is handled by the store. Invalidity flagging forms part of the general fault tolerance scheme of the interface allowing (for example) a mission critical function such as transfer alignment to be suspended and then restarted when the data is valid. Validity flagging allows the application process to coast

through short periods of bad inputs in a roll forward error recovery strategy. From a safety perspective invalidity flags are a lightweight mechanism that decouples data supplier and consumer by containing the propagation of a hazardous but transient error. While the standard requires the use of one bit per word, some store interfaces have used one bit flag per data entity. This semantic drift introduces ambiguity and the potential for hazardous omission style design errors during integration.

The mandated use of a Rotated¹⁹ Modulo 2 (XOR) 16 bit checksum for the standard data messages provides a minimum and consistent level of data redundancy within each message. The use of a checksum as an error code for each message represents a compromise between the ability to detect errors in the message, the vulnerability of the data/error code to bit errors, checksum processing and re-transmission overheads. In practice the checksum will detect all single bit errors, 93.95% of two bit errors and all error bursts of length 16 bits or fewer. Burst errors greater than 16 bits in length are detectable if they result in an odd number of actual bits being inverted or if the inverted bits do not align in the same bit position in the affected words. For applications where burst errors (such as those induced by impulse noise) are the dominant source of errors, Modulo 2²⁰ provide better error detection than one's complement addition, Fletcher and Adler checksums (Maxino 2006).

As an illustrative counter example if the message inversion technique of the CAN open safety protocol discussed in 6.4.4 was adopted error code size would be directly proportional to the size of data and would become proportionally more vulnerable to bit errors while imposing higher retransmit overheads than the fixed 16 bit checksum of MIL-STD-1760.

Another question that arises in calculating a checksum is whether to allocate the checksum to hardware, firmware or software. The advantage of calculating the checksum in software is that a complete end to end check of the logical interface, the terminals host processor, its memory and the embedded terminal processor and firmware is provided. This end to end check partly addresses the vulnerability of single bit data storage. However if checksum calculation are allocated to software Cyclic Redundancy Checks (CRC) are much less attractive as the implementation of modulo-2 division is less straightforward in software than hardware²¹. A related issue is how checksums are handled within a distributed system, i.e. should the checksum be generated at the point of data creation? This last issue can be a significant integration issue for legacy aircraft.

¹⁹ Rotating the blocks of data randomises the checksum inputs improving its performance against errors that are regular in nature, for example consistent corruption of a start/end word bit.

²⁰ Two's complement addition, and CRC checksums also perform better but with proportionally greater computational overhead.

²¹ Lookup tables and XOR calculations versus using linear feedback shift register.

6.10.3 Session layer

The MDT protocol elements are used by the aircraft to initialise the store with aircraft stored data upon start-up and to pass mass data between aircraft and store when the store is selected. The MDT protocol allows the implementation of bi-directional data transfers, the ability to initiate software programs and four levels of data integrity checking (File, Record, Block or none). Although not identified as safety critical in the same fashion as the store control and monitor messages MDT can potentially transfer data (ranging from targeting to software program) which bears directly upon the safe behaviour of the store.

Initially three options were investigated by the standardisation committee; separate data blocks to control and status the transfer, integrating the protocol into the actual data blocks or blind transfer via a dedicated sub address. While for efficiency of bus usage the first protocol was selected, it is interesting to note that the third approach was eliminated due to concern from the nuclear safety community that mass data might be erroneously transmitted to the incorrect destination with hazardous consequences (MIL-HDBK-1760A).

The down-load of such data therefore needs careful consideration as to the integrity of transfer required. Any assessment of criticality and justification for the required data integrity checks and is normally documented in the ICD. These decisions are driven by the requirements for access and modification of MDT data, and are also an area where interface incompatibilities can easily arise because of the optional nature of the checksum implementation. For example in one program the store implemented checksums only at the file level (as data was not intended to be modified after download) while the aircraft implemented them for records (as data was intended to be uploaded and modified) with neither design decision being explicitly documented in their respective ICDs. On aircraft file checksum processes need to be coordinated with those of the mission planning environment to ensure a valid transmission path from planner to store.

6.10.4 Transport layer

Fixed addresses are used to identify store RTs. If a variable addressing scheme were used it would degrade the safety of the system by introducing complex, non-deterministic and potentially hazardous behaviour. Because of this potential hazard and to further enhance nuclear safety, two S/A are set aside for exclusive use by nuclear munitions (an additional fire wall). Each critical control message also contains an Address Confirm field. The Address Confirm field provides a cross check of the MIL-STD-1553B address and reduces the likelihood of a correct command being processed by the incorrect store leading to a potentially hazardous state.

6.10.5 Network layer

MIL-STD-1760 outlaws the following mode commands:

1. dynamic bus control (aircraft is always BC), and

2. reserved mode codes.

The first constraint ensures that the bus architecture remains a master/slave type and enforces a simple bus design²², while the second ensures that developers do not subvert the standard by use of reserved mode commands. The potential hazard of a store failing to relinquish bus control (as is the case in token passing protocols) is also eliminated by this constraint.

Mission Store RTs must implement the following status word conditional message implementations:

1. use of the MIL-STD-1553B busy bit;
2. If a subsystem has a self-test capability, the a subsystem status flag is required;
3. Support Inhibit Terminal Flag if Terminal Flag Bit is implemented; and
4. If Service Request is used the Transmit Vector Data Word must be available when the bit is set.

Use of the MIL-STD-1553B RT status word's optional busy bit indicates to a BC that an RT/subsystem is unable to move data in compliance with the BC's command. Maximum busy time allowed is 500 ms for start-up and 50 μ s otherwise. The use of the busy bit reduces state ambiguity allowing the BC to discriminate between a failed RT and one that is still processing.

The BC must interpret the subsystem flag bit as total loss of the store again enforcing the fail safe behaviour. Because the flag may have been set by a transient condition the standard recommends that a reset terminal mode code be sent²³ and the flag rechecked. This redundant check provides a path to a safe state with a reduced probability of transitioning due to a false alarm.

The MIL-STD-1553B terminal flag bit is used to indicate a detected RT hardware failure and, in conjunction with mode code commands to deactivate and activate the terminal BIT, supports fault diagnosis by the BC.

A message error bit is required in RTs by MIL-STD-1553B, this bit is set to Logic 1 for several error conditions and the status word suppressed. These actions ensure that the BC has a capability to detect faults masked by the RT fault-tolerance mechanism. Message errors can be related to the application, datalink, network or physical layers, as Figure 6-7 illustrates. Should a message error occur, MIL-STD-1553B requires the entire message to be considered as invalid²⁴, eliminating the possible use of ambiguous data with hazardous consequences.

The optional checks for illegal command messages of MIL-STD-1553B also map to this layer. Illegal commands (such as reserved mode codes) are those that pass the validity checks but cannot be implemented by

²² Relative to more complex ones such as bus contention or token ring.

²³ This command resets the terminal, the subsystem cannot be reset.

²⁴ Message validation in MIL-STD-1553 strictly applies to the data component of a message, i.e. an invalid command word without a data word is ignored by the RT.

the RT by design. The standard requires that the BC developer ensure that no illegal commands are sent (as part of a fault avoidance strategy) while illegal command detection is optional for the RT. Where implemented illegal commands are handled by an RT in the same way as invalid commands i.e. no response and set the message error bit in the status word.

Safety critical message types are also required by MIL-STD-1760 to have a header word as the first data word. This allows the RT to check if the message was in fact intended for it and not a hazardous command generated by a double bit error in the transmission of the command word.

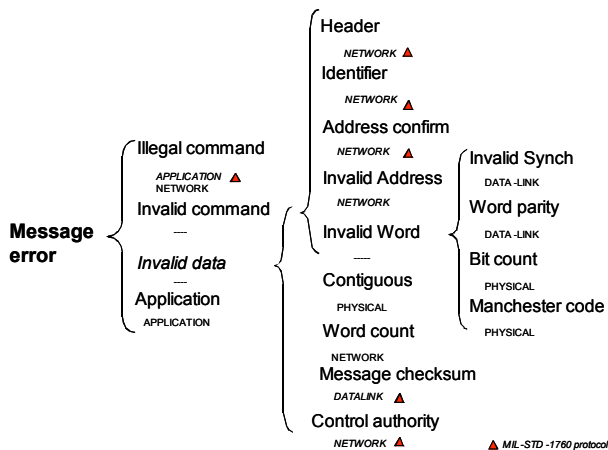


Figure 6-7 MIL-STD-1760 message error taxonomy

6.10.6 Data link layer

In addition to the constraints introduced by Notices 1 and 2 of MIL-STD-1553B²⁵, MIL-STD-1760 further limits the use of broadcast commands. The standard permits, but discourages, use of broadcast address 31 and introduces the following constraints on its use:

1. Safety critical data cannot be broadcast;
2. A store accepting broadcast messages must also accept the same data in non-broadcast mode;
3. The BC cannot issue a broadcast command to MIL-STD-1760 specified sub-addresses;
4. The store is required to work with an aircraft or carriage store that does not support broadcast;
5. Error detection schemes for significant broadcast data are required; and
6. Stores must implement the broadcast command received bit in the status word, allowing for post broadcast round robin polling.

These restrictions are intended to reduce the likelihood of the following potentially hazardous states occurring:

1. undetected message loss or error;

2. earlier model RTs²⁶ do not recognise broadcasts and improperly interpret them; or
3. a store using broadcast messages ends up on an aircraft where it's broadcast S/A is used for something else.

The fundamental problem with broadcast mode is that it does not provide positive closed-loop control, which denies the BC any ability to check for failures or errors (Leveson 1995). The use of broadcast commands also complicates fault tolerant behaviour, for example an RT which has failed silent upon detection of an illegal address assignment may still legally process a broadcast command. The problems introduced by broadcast are illustrative of real time distributed system coordination problems. Hazards 2 and 3 are an instance of the backwards and forwards compatibility coordination problem discussed in 3.2.2.

S/A 08 (decimal) is reserved to avoid misinterpretation of a status word (with service request set) as a command word for sub-address 08. The standard does allow it to be used for test purposes at the user's risk.

The first three bits of each 20 bit MIL-STD-1553B word are a synchronisation field (an invalid Manchester waveform) allowing a decode clock to re-sync at the beginning of each new word. The advantages of a dedicated and unique synchronisation field are that it reduces the likelihood of a receiver missing the start of a new message and eliminates possible confusion of a synch field with part of a message (e.g. alialising). The field is also used to distinguish between MIL-STD-1553B command and data words transmitted. This is an example of both low level data redundancy and the decoupling of synchronisation from bit transmission functions afforded by MIL-STD-1553B as a character oriented protocol.

An alternative synchronisation scheme (taken from CAN, a bit oriented protocol) is to bit stuff a message with complementary value bits after a series of N same valued bits in order to maintain loop synchronisation. However, using this synchronisation scheme low level multi-bit errors can cause cascading errors in which bit errors cause misinterpretation of stuffing bits as data and vice versa leading to data pattern shifting, large bit error rates and resultant CRC failure (Tran 1999).

6.10.7 Physical layer

MIL-STD-1760 has specific physical interconnection media requirements for more stringent waveform specification at the transmitting-end and tolerance of more distortion and loss at the receiving-end than MIL-STD-1553B. These additional robustness requirements address the bus waveform distortion introduced when a mission store separates from the parent aircraft leaving behind a 'stub' of bus. Current limits are also introduced to prevent multiple high currents on the MIL-STD-1553B bus. The added MIL-STD-1760 margin above that of MIL-STD-1553B also offsets the greater uncertainty of

²⁵ The USAF believed that switching from a current BC to a backup BC via dynamic bus control mode command was too hazardous and prohibited this command in Notice 1 to MIL-STD-1553B.

²⁶ Broadcast mode was introduced to MIL-STD-1553B at Notice 2 of that standard.

assumptions made about actual bus performance in such a dynamic environment.

MIL-STD-1553B requires a maximum allowable word error rate of no greater than one in 10^7 words in the presence of an impulsive RF noise environment. The standard recognises that impulse noise (such as relay switching) are more typical of noise sources having adverse effects but notes that because it is extremely difficult to analyse the effects of impulse noise a worst case white Gaussian noise model was used to define its noise rejection requirements²⁷. The specification of such criteria form part of the complex signal design pattern identified in section 6.4.9.

Inherent RF noise rejection is provided by the bus 1 MHz fundamental frequency being lower than the frequency of most onboard noise sources. The use of a bipolar waveform increases the signal to noise amplitude relative mono-polar signals and provides improved EMI immunity. EMI rejection is further enhanced by a shielded twisted pair design for the bus and an outer shield for the aircraft to store umbilical. However, umbilical shielding is not necessarily implemented inside the aircraft and shielding from the internal EME can be a significant issue for legacy system integrations. The event triggered master/slave protocol also provides a more robust scheduling guarantee in the presence of EMI because the BC can flexibly retransmit a corrupted message to make maximum use of the bandwidth. In comparison time triggered protocols would require multiple duplicate transmissions to achieve the same robustness.

Where noise does occur and zero crossings generate a single bit error the error will notionally be detected by the MIL-STD-1553B word parity bit. However, a single noise event may disrupt more than one bit. Thus as the parity bit has two possible values (0 and 1), the bit is limited to a 50% chance of a >1 bit error detection. To enhance error detection rotated modulo 2 checksums are introduced by MIL-STD-1760 for messages and BCH polynomial error code authority words for safety critical commands.

Manchester coding is utilised by the MIL-STD-1553B protocol and provides (amongst other advantages) high noise immunity, RTs are then required to check that the word bits are a valid Manchester II code before acting upon them. While an RT could detect a 'skewed' bit and recover the value, MIL-STD-1760 requires that for critical control bits the value be discarded. Again the discard data policy eliminates the use of ambiguous data for safety critical operations with potentially hazardous consequences.

The MIL-STD-1553B protocol requires at least 4 μ s between messages, with the RT required to respond to a command within a period of 4 to 12 μ s. The inter-message gap addresses the coincidental response hazard of a RT receiving and processing a message as valid which has actually arrived by coincidence, while the

minimum response time addresses the RTs inherent latency in processing a BC command. If there is a delay in response of greater than 14 μ s, it is assumed by the BC that no response occurred, which then must respond to the un-responsive (and potentially hazardous) communications channel (Jaffe 1989).

MIL-STD-1553B also requires that terminals (RT or BC) contain a hardware fail safe timer to prevent any transmission on the data bus exceeding 800 μ s. As no valid transmission is longer than 660 μ s only a failure in the terminal could result in such a transmission. The fail-safe timer prevents a 'babbling idiot' failure propagating to total bus failure.

7 Using MIL-STD-1760 as part of a safety case

7.1 The need for an integration safety case

Safety cases or arguments have become an accepted part of the development of complex safety critical systems, reflecting a trend in away from the prescriptive application of regulations and towards requiring a developer to formally justify the safety of the system. By definition, MIL-STD-1760 expresses a form of safety argument in which the implicit definition of safety is 'compliance with the standard'. Logically a safety case for a stores integration program should incorporate this argument as part of the overall safety argument. However, there are a number of challenges that need to be addressed when doing so.

As part of such a safety case the integrator must also establish that the interface definitions, usually based on different revisions of the standard, have not introduced coordination style hazards. As illustrated by the Goal Structure Notation (GSN) diagram of Figure 7-1 this definition provides the essential context for the safety argument in the same way that a hazard analysis provides the context for arguments about hazard control. Similarly the integrator must justify the assumption of completeness, including where the integration program stopping rules. For example, it might be assumed that the design is currently safe based on (in part) the existence of an existing safety program or service history.

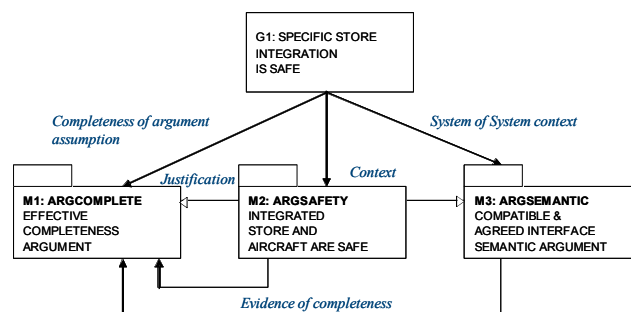


Figure 7-1 Integration safety case

7.2 Safety argument patterns

High level safety arguments can vary markedly; one program may adopt a hazard avoidance argument pattern whilst another may adopt a functional integrity argument pattern. Figure 7-2 illustrates how a MIL-STD-1760

²⁷ For a band limited system random impulse noise may be approximated by Gaussian noise.

compliance argument can be integrated into a hazard directed or functional integrity style arguments constructed using interface and contract extensions for Goal Structure Notation (GSN) where element of the MIL-STD-1760 argument (goals, evidence or solutions) are referenced to the hazard or functional integrity argument.

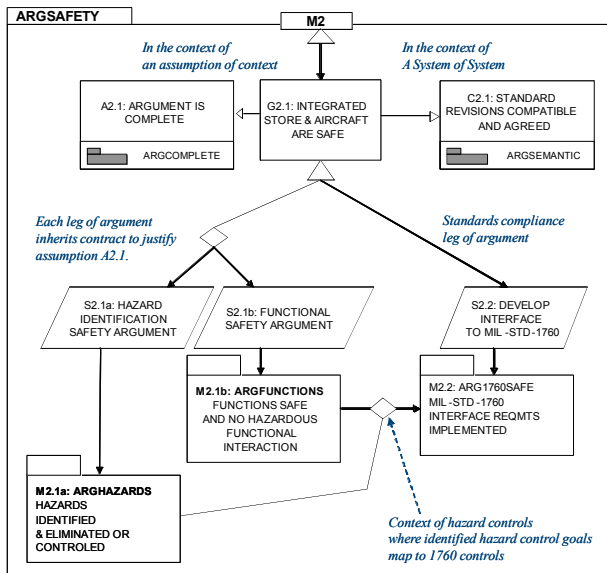


Figure 7-2 Integrated MIL-STD-1760 safety argument

7.2.1 Hazard avoidance argument

In a hazard avoidance style argument, the implicit definition of safe is ‘hazard avoidance’. The usual style of this argument is to argue that all identified hazards have been either eliminated or their risk controlled. The challenge for this argument is to provide a plausible (i.e. comprehensive) context of identified hazards. MIL-STD-1760 can be used in a hazard avoidance argument to identify hazards, both directly and by inference from the safety constraints of the standard.

7.2.2 Functional integrity argument

In an integrity style argument, the implicit definition of safe is ‘integrity level’. The usual style of this argument is to argue that all safety critical functions have been identified; all identified functions meet specified integrity levels and do not interact in a hazardous fashion. The challenge for this argument is to demonstrate either the independence of functions or their non hazardous interaction.

Simplistically, having applied MIL-STD-1760, it could be argued that the required integrity level is achieved. However because of the distance between the premises and conclusion of such an argument it may be more convincing to breakout the interface into lower service functions (as Figure 7-3 illustrates) and use MIL-STD-1760 implementation as evidence for each layer. Arguments as to the potential for hazardous interactions across the logical and direct interfaces of the protocol are also required. This approach provides more direct ‘evidence’ based assurance than the traditional process based assurance typically associated with functional

safety arguments (Weaver 2003). It is also unlikely that application at a high level of abstraction can be argued because MIL-STD-1760 would rarely be implemented uniformly on both sides of the interface.

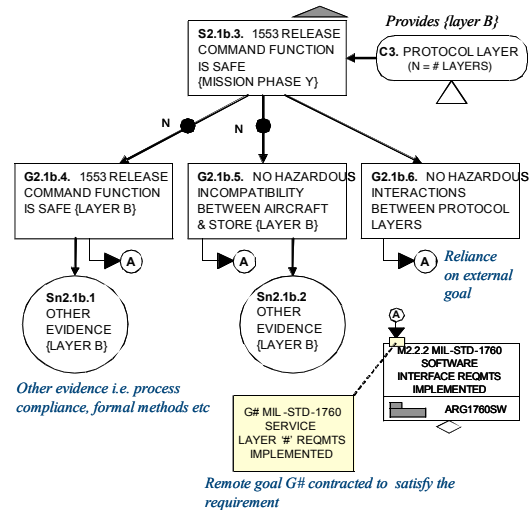


Figure 7-3 Functional integrity argument

7.2.3 The issue of completeness

For integration programs the completeness of a safety argument can be compromised by:

1. originally incomplete specifications,
2. invalidated un-stated interface assumptions,
3. unidentified or unrecorded pre-existing hazards,
4. unidentified new integration related hazards, or
5. in-complete implementation of the standard.

Because it is impossible to argue that any safety argument is complete, it is necessary to make an assumption of completeness, and ideally this assumption should also be both explicit and rigorously justified²⁸. MIL-STD-1760 can assist in justifying this assumption by providing:

1. a comprehensive (not necessarily complete) specification of AEIS inputs and outputs;
2. a reduction in assumptions about the interface;
3. broad safety criteria to control un-identified hazards; and
4. an interface design of known provenance reducing the likelihood of new hazards.

Implementing the latest requirements of MIL-STD-1760 to comply with a store’s ICD may also prove to be impractical and, if implemented, may not contribute materially to safety. Where such circumstances arise, an argument as to why an ‘effectively’ complete implementation is acceptable must support any general argument for completeness. As the argument fragment of Figure 7-4 illustrates, safety is not an absolute and is

²⁸ Usually this is achieved through a mix of product and process evidence (Chinneck 2004).

often evaluated through design provenance and experience as much as risk analysis (AC 21-101-1).

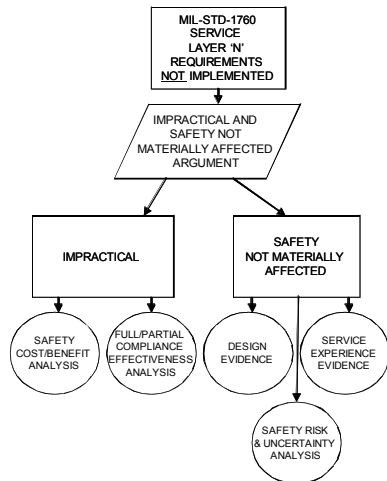


Figure 7-4 'Safety not materially affected' argument

7.3 Complexity

One of the challenges in developing a safety case for a complex system is managing the complexity of the argument itself. Safety arguments also generally draw on multiple sources of information which can further complicate argument structure. Figure 7-5 illustrates the use of a hierarchical structure to manage the complexity of a MIL-STD-1760 compliance argument. Module 2.2 of Figure 7-5 can be further broken into a software hierarchy based on the OSI model allowing the argument to separate out the specific safety concerns at each layer of the model. Similarly Module 2.3 can be broken into a part/whole hierarchy matching the various interfaces of AEIS. The dashed figures of Figure 7-5 indicate where module 2.1, 2.3 and 2.3 support the cohort safety arguments identified in Figure 7-2.

The architecture of MIL-STD-1760 can also be used to structure the functional and hazard directed arguments. For example hazards identified in the hazard avoidance argument can be grouped by functional interface or OSI service layer for clarity (Chinneck 2004).

Finally MIL-STD-1760 provides a 'one stop shop' of reference material, further simplifying the arguments structure.

7.4 Incremental safety case development

One of the often stated goals for a safety case is that the safety argument should influence the design. To achieve this requires an incrementally developed safety case starting with a high level architectural safety pattern argument followed by the development of supporting safety arguments for hardware and software implementation. This dictates a means/ends hierarchy where safety patterns selected during preliminary design must necessarily assume that the detail design will not subvert these patterns and, once developed, hardware and software safety arguments must validate this assumption. This structure, reflected in Figure 7-5, consists of three modularised sub-arguments with the architecture module providing goals for the software and hardware modules,

while they in turn provide validation of the architecture through compliance evidence.

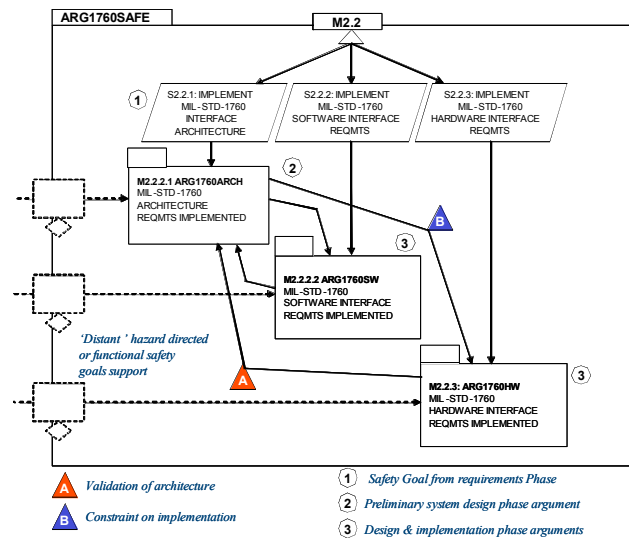


Figure 7-5 Incremental safety argument development

8 Conclusions

Architectural decisions can have significant impact upon the safety of a system and the effort required to verify its safety. For a given stores integration program consideration must also be given to those system of system coordination issues forming the programs context.

As an interface standard MIL-STD-1760 specifies a coherent set of functional interfaces that emphasise fault tolerance to achieve the safety and reliability goals of a specific program. As with any standard the downside is that developers must also sacrifice design freedom to achieve these benefits. However this reduction is offset by enhanced interoperability and effectiveness at the system of systems level.

Integrating MIL-STD-1760 compliance into a safety case can strengthen the safety argument by providing an ability to argue the reuse of successful architectural patterns. Use of the standard can also provide evidence to support assumption of completeness as well as an organising principle for the arguments structure. Because MIL-STD-1760 contains both architecture and implementation elements it can be used in the incremental development of safety arguments with initial architectural strategies being levied as safety requirements for subsequent detailed design efforts. A MIL-STD-1760 compliance argument can support both hazard and safety integrity style arguments.

9 References

- AC 21-101-1 (2003): Establishing the Certification Basis for Changed Aeronautical Products, *Advisory Circular*, Federal Aviation Administration, 28 April 2004.
- ARP 4754 (1996): Certification Considerations for Highly Integrated or Complex Avionics Systems, Society of Automotive Engineers.
- Chinneck, P., Pumfrey, D., McDermid, J. (2004): The HEAT/ACT Preliminary Safety Case: A case study in

- the use of Goal Structuring Notation, *9th Australian Workshop on Safety Related Programmable Systems (SCS'04)*, Vol 47.
- Douglass, B.P (1999): *Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*, Addison Wesley, 1999.
- DSP 304 (2001): CanOpen Safety Protocol, CAN in Automation (CiA) GmbH, January 2001.
- FlexRay (2005): Communications System Protocol Specification Version 2.1, FlexRay Consortium, May 2005.
- Gray, D. (1985): Why Do Computers Stop and What Can Be Done About It?, Tandem Computers Technical Report 85.7 PN87614, June 1985
- Hyman, M. (2003): Space Station Takes Unique Twist on MIL-STD-1553B, *COTS Journal, Volume 4 No. 7*, p52-71.
- IEC 61508 (1998-2000): Functional Safety of Electrical/Electronic or Programmable Electronic Safety-Related Systems, Volumes 1 to 7, International Electro-technical Commission (IEC).
- ISO/IEC 7498-1 Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model.
- ISO 11898 Road Vehicles - Interchange of Digital Information- Controller Area Network (CAN) for high-speed communication.
- Jaffe, M., and Leveson, N. (1989): Completeness, Robustness, and Safety In *Real-Time Software Requirements Specification*, Proc. ACM.
- Jaffe, M. (1999): Architectural Approaches To Limiting The Criticality Of Commercial-Off-The-Shelf (Or Other Re-Used Software), DASC.
- Kelly, T. (1998): Arguing Safety – A Systematic Approach to Managing Safety Cases, Doctoral thesis, University of York.
- Leveson, N.G. (1995): *Safeware, System Safety and Computers*, Addison Wesley.
- Lewis, D.K. (1969): *Convention: A Philosophical Study*, Harvard University Press.
- Lutz, R.R. (1993): Targeting Safety Related Errors During Software Requirements Analysis. In *Proceedings SIGSOFT 93: Foundations of Software Engineering*.
- Mackall, D.A. (1988): Development and Flight Test Experiences with a Flight-Critical Digital Control System. NASA Technical Paper 2857, NASA, Dryden Flight Research Facility, California, USA.
- Maier, M. (1996): Architecting Principles for Systems of Systems. In *Proc. of the Sixth Annual International Symposium, International Council on Systems Engineering*, Boston, MA, p567- 574.
- Maxino, T.C. (2006): The Effectiveness of Checksums for Embedded Networks, Thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- Meyer, M., The Art of System Architecting, CRC, 1998.
- McDermid, J. A, Pumfrey, D.J Software Safety: Why is there no Consensus?, in *Proceedings of the International System Safety Conference (ISSC) 2001*, Huntsville, System Safety Society, 2001.
- MIL-HDK-244A (1990), Guide to Aircraft/Stores Compatibility, US Dept. of Defense, 6 April 1990.
- MIL-HDBK-1553A (1998), Multiplex Applications Handbook, US Dept. of Defense, 1 November 1988.
- MIL-STD-1553B Notice 4 (1996), Digital Time Division Command/Response Multiplex Data Bus, 15 January 1996, US Dept. of Defense.
- MIL-STD-1760D (2003): Interface Standard for Aircraft-store Electrical Interconnection System (AEIS) US Dept. of Defense.
- NATO STANAG 3908: Standardised Avionics Terms and Abbreviations, 2nd Ed., NATO.
- Rushby, J. A., (2001): Comparison of Bus Architectures for Safety-Critical Embedded Systems, CSL Technical Report, Computer Science Laboratory, SRI International.
- RTCA DO-178B/ED-12B (1992): Software Considerations in Airborne Systems and Equipment Certification. RTCA.
- SAE-AS-1B3 (2002): General Aircraft Stores Interface Framework (GASIF), draft F.
- Sivencrona, H., Chalmers, Hedberg, J., Röcklinger H. (2001): Comparative Analysis of Dependability Properties of Communication Protocols in Distributed Control Systems, PALBUS Task 10.2. www.sp.se/pne/software&safety/palbus.
- Spray, S.D. (1995): Principle Based Passive Safety In Nuclear Weapon Systems, *High Consequence Operations Safety Symposium*, Sandia National Laboratories, Albuquerque, 13 July 1994, as quoted in Leveson 1995.
- Tran, E. (1999): Multi-Bit Error Vulnerabilities in the Controller Area Network Protocol, Thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- TTTech (1999): Specification of the TTP/C Protocol, Time-Triggered Technology (TTTech) Computertechnik AG, Vienna, Austria.
- Ward, J.R. (1993): Beyond Integrated Weapon System Management - Acquisition in Transition, Executive Research Project RS8, The Industrial College of the Armed Forces National Defense University, Fort McNair, Washington, D.C. 20319-6000.
- Weaver, R., Kelly, T., Fenn, J. (2003): A Pragmatic Approach to Reasoning about the Assurance of Safety Arguments, In *8th Australian Workshop on Safety Related Programmable Systems (SCS'04)*, Vol 33.

Implementation of a Triple Modular Redundant FPGA based Safety Critical System for reliable software execution

Venkatesh Vasudevan Email:venkat@itee.uq.edu.au

Peter Waldeck Email:waldeck@itee.uq.edu.au

Hardik Mehta Email:mehta@itee.uq.edu.au

Neil Bergmann Email:bergmann@itee.uq.edu.au

School of Information Technology and Electrical Engineering
University of Queensland

Abstract

This paper describes the implementation of a TMR (Triple Modular Redundant) microprocessor system on a FPGA. The system exhibits true redundancy in that three instances of the same processor system (both software and hardware) are executed in parallel. The described system uses software to control external peripherals and a voter is used to output correct results. An error indication is asserted whenever two of the three outputs match or all three outputs disagree. The software has been implemented to conform to a particular safety critical coding guideline/standard which is popular in industry. The system was verified by injecting various faults into it.

1 Introduction

1.1 Background

Field Programmable Gate Arrays (FPGA's) are semiconductor integrated circuits (IC's)/chips that facilitate custom user logic to be programmed using a bitstream. The devices can be reprogrammed in the field whenever the logic changes thus removing the need to remove the device or design a new system. Hence the product can be upgraded in the field with new features without any replacement of parts. The whole system (processor and its peripherals) can be housed in a single FPGA thus reducing board size considerably. Time to market or development time is considerably reduced due to rapid prototyping. FPGA's allow a software-hardware co-design methodology which is a must for safety critical applications and hence it is convenient to use FPGA's for the development of the same.

1.2 Objective

Triple Modular Redundancy (TMR)(B.W.Johnson 1989) is a popular concept being used by many designers of high reliability systems. The common methodology involves development of software conforming to safe coding guidelines (viz one may use a language like Esterel (G.Berry et al 2000) to define the specifications of the system and then implement it or if coding in traditional programming languages like C, follow guidelines set out by subsets of the languages like MISRA-C(MISRA 2004)) and implementation of

the same on three separate microprocessors (each on its own motherboard). Our scheme followed the same strategy. However the three microprocessor systems were implemented on one FPGA. The goal was to achieve reliability in software execution by employing hardware redundancy so that if one of the software fails due to a hardware fault (bitstream / configuration errors, stuck at faults, bit flips due to radiation etc) then the other softwares running in parallel can keep the system in operation. This is due to the fact that the hardware error occurs only in a part of the FPGA and hence only a small part is affected, not the whole FPGA. The correctness of the software was tested by debugging it using XMD (Xilinx Microprocessor Debugger) which allows the user to single step through the code. The software was included in the system after thorough testing. The software was written in C and checked for MISRA-C(MISRA 2004) rules by Abraxas Software's CodeCheck v1300 B1 tool (results in System Verification section). The code was for a seconds counter and instances of the same code were allowed to run on each of the three microprocessors. The design was targeted for a Xilinx FPGA and hence Xilinx design flows.

1.3 Literature Review

Other teams have worked on TMR systems on FPGA's. Bitstream faults were investigated by (L.Carro et al 2005) and results were based on number and placement of voters. (Rami Melhem et al 2002) have performed analysis of energy efficiency of Duplex and TMR systems. (Hyunki Kim et al 2002) have developed a TMR system based on MC68000 (microprocessor from Motorola). A number of other works were reviewed and it was found that little work has been done on implementing TMR software systems on FPGA's with a focus on improving software reliability on FPGA's. Our team was and is focussed on reliable software execution on FPGA's for implementing FPGA based Safety Critical Systems. Hence we have implemented a system which can execute its software reliably during various hardware faults.

2 System Design

2.1 The System

As shown in Figure 1 the system is composed of three Microblaze processor systems and the deciding voter. Each processor system (Figure 2) is a whole system in itself in that it contains the following essential components:

- Microblaze or PowerPC microprocessor (in this case Microblaze)

Copyright©2006, Australian Computer Society, Inc. This paper appeared at the *11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, Melbourne. Conferences in Research and Practice in Information Technology, Vol. 69. Tony Cant, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included

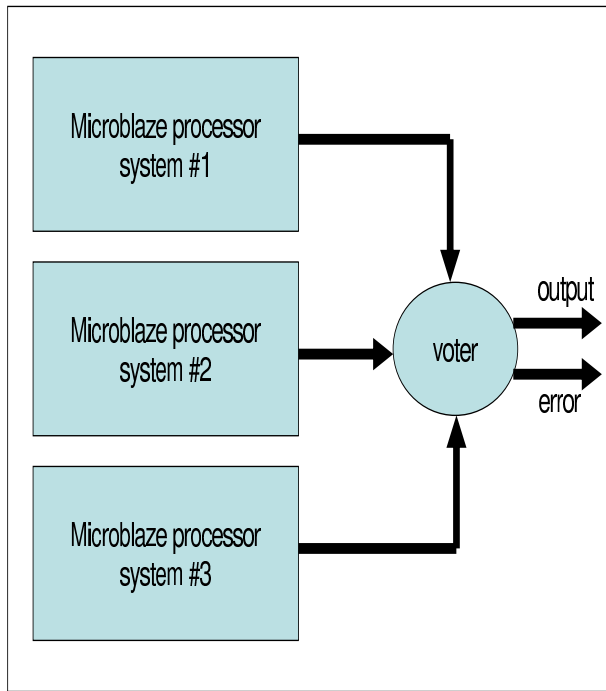


Figure 1: The System

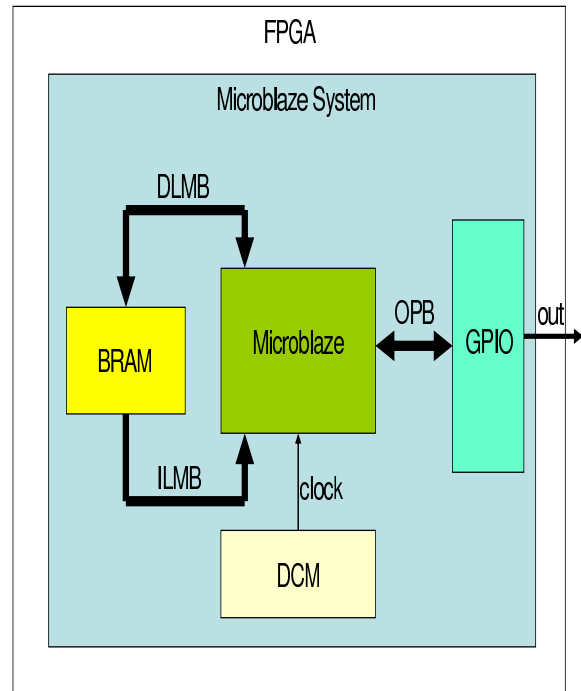


Figure 2: Microblaze System

- Program Memory (in this case Block RAM)
- Data Memory (in this case Block RAM)
- Local Memory Bus (LMB) for both instruction side and data side and associated controllers
- OnChip Peripheral Bus (OPB)
- Digital Clock Manager (DCM) and
- Peripheral (in this case GPIO (general purpose I/O))

For the above shown system, the voter peripheral and GPIO peripherals (Figure 2) were created using 'Create - Import Peripheral' a tool available in Xilinx EDK (Embedded Development Kit) for creating custom peripherals (that are not otherwise available in EDK) and attaching to the processor via either OPB (OnChip Peripheral Bus), PLB (Peripheral Local Bus) or FSL (Fast Simplex Link). In this case the peripherals were designed to interface to the OPB of Microblaze. This whole system was housed in a Virtex 4 (XC4VLX25) FPGA on the ML401 Xilinx evaluation board.

The whole system was developed using the EDK design flow by manually creating the MHS (Microprocessor hardware specification) and MSS (Microprocessor software specification) files. This means that these files were not automatically generated by EDK which is what generally happens during development. In our case we had to write out these files using the syntax for MHS and MSS files since our system could not be directly generated by EDK. These files are the deciding factors in Xilinx EDK that determine the hardware synthesized and the software libraries generated.

All the code and data resided in internal BRAM (block RAM) blocks (3 in number)

The GPIO peripherals interfaced to the microprocessor via the OPB

The voter peripheral however even though designed for interface via OPB was not interfaced to the CPU rather it was interfaced only to the GPIO outputs (inputs to the voter) and external outputs (the output or result itself and error output).

The external outputs were led's driven by FPGA I/O's.

2.2 Software

Three instances of the same code were created as shown below (incorporating fault injection) conforming to MISRA-C (MISRA 2004) rules. Due to the presence of three code instances and identical hardware architectures, true concurrency of execution of operation was obtained.

The code in this case is for a simple counter whose two least significant bits are output to the voter. The voter in turn outputs the two bit value to LED's on the board. The header file 'xparameters.h' contains definition of the base address of the GPIO (general purpose input output) whose output is connected to the voter input. The header file 'vgpio.h' contains function prototypes of GPIO read/write functions. It also includes the 'xbasic_types.h' header file which contains typedef'd data type definitions e.g Xuint8. The counter runs at a speed of 1 Hz due to the fact that the external clock frequency is 100 MHz and the number of clock ticks that are counted are also the same number (ledX_delay where X=0,1,2).

The code is shown to demonstrate how safe code was written and how we plan to write it in future (e.g no use of pointers, use of macros, adequate function prototypes etc). The three codes although the same were different when testing for faults namely stuck at faults. For example (see fault injection in code) the GPIO's were stuck at either 0x00 or 0xff for one of the codes with the remaining codes being intact.

Microblaze System #1

```
//counter0.c
```

```
#include "xparameters.h"
```

```
#include "vgpio.h"
```

```
#define led0_base XPAR_VGPIO_0_BASEADDR
```

```
#define led0_delay 100000000
```

```
#define led0_offset 0
```

```
void delay(void)
```

```
int main()
```

```
{
```

```
Xuint8 led0_data;
```



```

led0_data = 0x00;
while(1)//GPIO write
{
    //Stuck at fault injection
    //VGPIOMWriteReg(led0_base,led0_offset,0xff);
    //VGPIOMWriteReg(led0_base,led0_offset,0x00);
    VGPIOMWriteReg(led0_base,led0_offset,led0_data);
    delay();
    led0_data++;
}

```

```

return 0;
}

```

```

void delay(void)
{
    Xuint32 led0_delay_value;
    led0_delay_value = 0x00;
    while(led0_delay_value < led0_delay)
    {
        led0_delay_value++;
    }
}

```

```

Microblaze System #2
//counter1.c
#include "xparameters.h"
#include "vgpio.h"

```

```

#define led1_base XPAR_VGPIO_1_BASEADDR
#define led1_delay 100000000
#define led1_offset 0
void delay(void)
int main()
{
    Xuint8 led1_data;
    led1_data = 0x00;
    while(1)//GPIO write
    {
        //Stuck at fault injection
        //VGPIOMWriteReg(led1_base,led1_offset,0xff);
        //VGPIOMWriteReg(led1_base,led1_offset,0x00);
        VGPIOMWriteReg(led1_base,led1_offset,led1_data);
        delay();
        led1_data++;
    }
}

```

```

return 0;
}

```

```

void delay(void)
{
    Xuint32 led1_delay_value;
    led1_delay_value = 0x00;
    while(led1_delay_value < led1_delay)
    {
        led1_delay_value++;
    }
}

```

```

Microblaze System #3
//counter2.c
#include "xparameters.h"
#include "vgpio.h"

```

```

#define led2_base XPAR_VGPIO_2_BASEADDR
#define led2_delay 100000000
#define led2_offset 0
void delay(void)
int main()
{
    Xuint8 led2_data;
    led2_data = 0x00;
    while(1)//GPIO write
    {

```

```

//Stuck at fault injection
//VGPIOMWriteReg(led2_base,led2_offset,0xff);
//VGPIOMWriteReg(led2_base,led2_offset,0x00);
VGPIOMWriteReg(led2_base,led2_offset,led2_data);
delay();
led2_data++;
}

```

```

return 0;
}

```

```

void delay(void)
{
    Xuint32 led2_delay_value;
    led2_delay_value = 0x00;
    while(led2_delay_value < led2_delay)
    {
        led2_delay_value++;
    }
}

```

For the above code instances, the libraries were generated by EDK after parsing MHS and MSS files. New drivers were created by the tool for the user created peripherals viz GPIO and voter. However only driver for the GPIO peripheral is being used.

2.3 Peripherals

This section explains design of peripherals and the design methodology employed as it impacts the ease of development. As has already been pointed out the GPIO and voter peripherals were created by the team using 'Create - Import Peripheral' a utility that is shipped alongwith Xilinx EDK. Of course one can use Xilinx provided GPIO cores as well. Development of our own GPIO peripherals happened as a result of issues related to the device driver. The voter peripheral was created due to our adopted design methodology. The design phase can actually have two different design flows. One is the Xilinx ISE (Integrated System Environment) flow (Figure 3) and the other is the EDK flow (Figure 4). Looking at Figure 1, one might say that the ISE flow looks like the logical one since the voter can be entirely developed in ISE (VHDL RTL) and the microprocessor file developed in EDK can be included in the ISE project and both the entities can be instantiated within a top level entity thus completing the entire system. This design flow works alright if only the microprocessor file is included and synthesized in ISE. It can be made to work with the VHDL and uP files instantiated together but this involves tinkering which goes very deep into the Xilinx files.

Hence we decided to take the EDK design flow methodology. Obviously since this is not ISE a direct implementation of the voter was not possible. Hence we had to create our own voter peripheral the discussion of which is carried forward in the subsections. The case of our own GPIO peripherals is also discussed in the following subsection.

2.3.1 VGPIOM

The GPIO core is not the standard Xilinx GPIO core. Its called VGPIOM which was developed by us for purposes described in the following.

The Xilinx GPIO is a complex core with certain functionalities that we did not require and the device driver too has complex usage in that it has to be initialized and configured the correct way so that it becomes suitable for our system. Our GPIO, the VGPIOM has a simple device driver and one can

immediately write to or read from it without any special initialization and configuration.

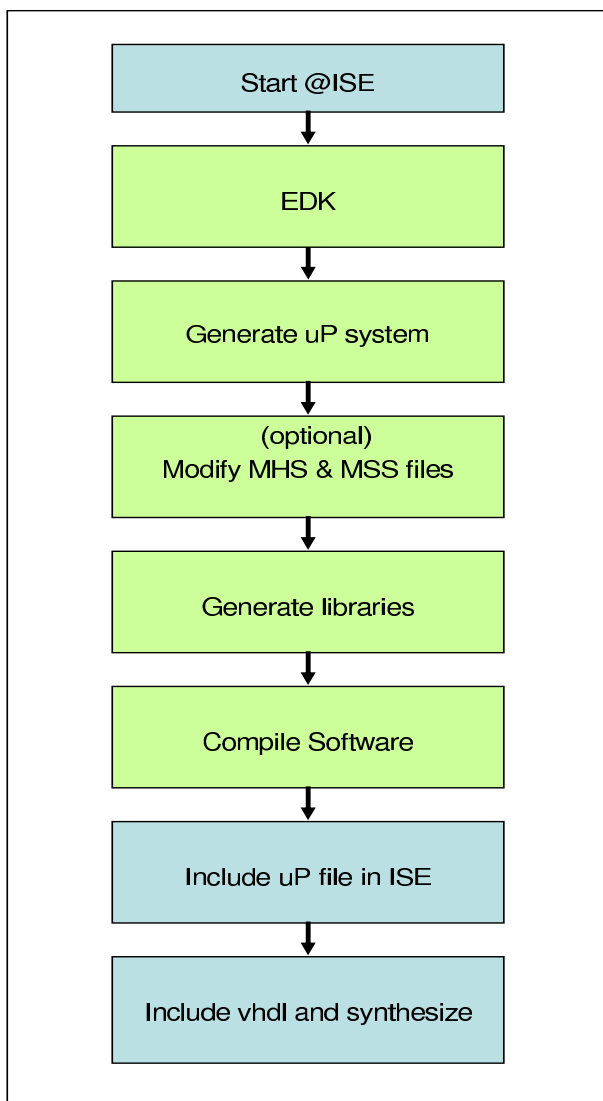


Figure 3: ISE Design Flow

Figure 5 shows the architecture of our GPIO. As shown it's a simple 32 bit register (for simplicity since Microblaze is 32 bit) featuring big endian (as Microblaze follows big endian) format.

- VGPIO device driver
EDK builds libraries containing drivers for each peripheral in the design after parsing the MHS and MSS files. In case of the VGPIO peripheral we used the basic write function which was provided by default for writing into the VGPIO register.

Before implementing the final system (Figure 1) we implemented the system as shown in Figure 6. This scheme had timing issues in that the outputs of VGPIO's (connected to inputs of voter) would arrive at the same time however the voter would have its third input (coming from Microblaze #2) much earlier thus only two inputs to the voter would match resulting in error LED turned on.

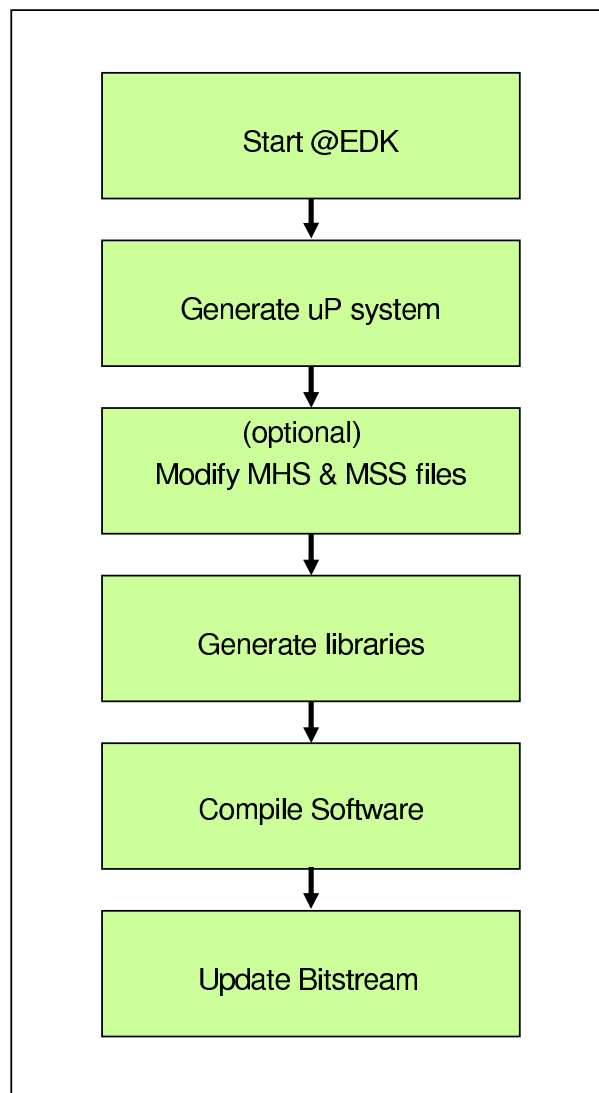


Figure 4: EDK Design Flow

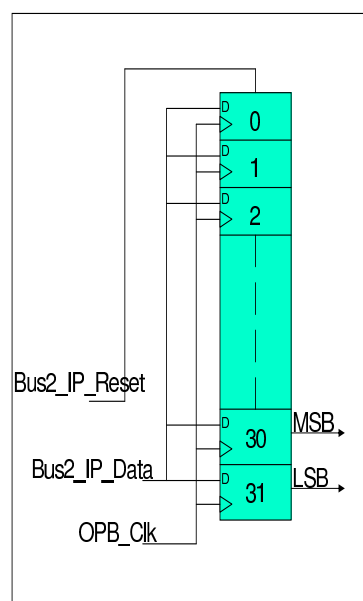


Figure 5: VGPIO Architecture

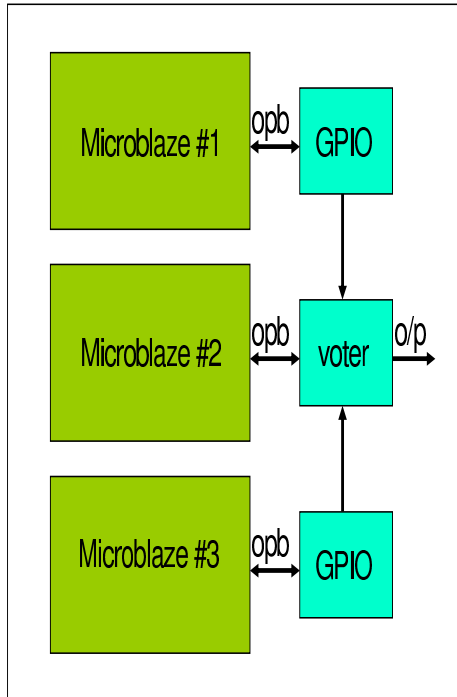


Figure 6: Initial System

2.3.2 Voter

The voter architecture is basically that of a comparator which looks at three inputs and checks for various combinations of inequalities. The voter too was built using 'Create - Import Peripheral' though it wasn't interfaced to any Microblaze. It contained both the OPB interface hardware as well as the comparator hardware both of them working in parallel without any connection whatsoever. We had to opt for this approach as the solution was the EDK design flow and hence we needed the voter to be available in EDK so that we could include it in our MHS description. The following VHDL snippet describes the voter architecture. Note that the OPB part has been omitted as it is not required to be shown. The VHDL code for the voter (a 2 bit module) basically compares the three inputs with each other and passes that value which appears on at least two inputs to the output. An error is generated if either only two inputs match or all inputs mismatch. In the event where all inputs mismatch the output is made "00". The inputs can assume values "00", "01", "10" or "11" hence the output can assume these same values. Even if only two inputs match, the voter will still output the value keeping the system in operation but at the same time it indicates to the personnel that an internal error has occurred.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

```
library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
```

```
entity voter is
port(voter_in0:std_logic_vector(0 to 1);
voter_in1:std_logic_vector(0 to 1);
voter_in2:std_logic_vector(0 to 1);
voter_out:std_logic_vector(0 to 1);
sys_error:std_logic);
end entity voter;
```

```
architecture voter_arch of voter is
signal voter_in0_store:std_logic_vector(0 to 1);
signal voter_in1_store:std_logic_vector(0 to 1);
signal voter_in2_store:std_logic_vector(0 to 1);
begin
voter_action : process( voter_in0_store,
                        voter_in1_store,
                        voter_in2_store ) is
```

```
begin
if(voter_in0_store = voter_in1_store) and
(voter_in0_store /= voter_in2_store) then
voter_out_store <= voter_in0_store;
sys_error_store <= '1';
elsif(voter_in0_store = voter_in2_store )
and (voter_in0_store /= voter_in1_store) then
voter_out_store <= voter_in0_store;
sys_error_store <= '1';
elsif(voter_in1_store = voter_in2_store ) and
(voter_in1_store /= voter_in0_store) then
voter_out_store <= voter_in1_store;
sys_error_store <= '1';
elsif(voter_in0_store = voter_in1_store) and
(voter_in1_store = voter_in2_store ) then
voter_out_store <= voter_in0_store;
sys_error_store <= '0';
else
voter_out_store <= "00";
sys_error_store <= '1';
end if;
end process voter_action;
voter_out <= voter_out_store;
sys_error <= sys_error_store;
end architecture voter_arch;
```

As shown in the above code the voter looks for different match possibilities and asserts an error if only two inputs match or all three inputs don't match. Figure 7 shows the voter architecture.

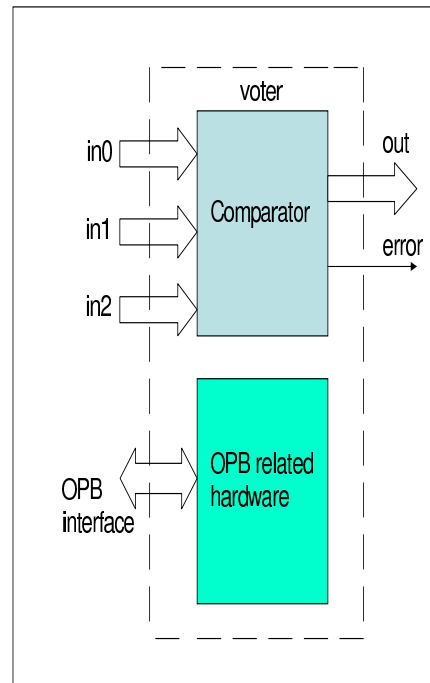


Figure 7: Voter Architecture

As shown the voter has two distinct parts, the comparator which performs the voting action and the dummy OPB part used only to get the voter peripheral included in the MHS file via the 'Add Edit cores' option in EDK. The comparator is purely combinational in nature for now and may be made sequential in future. Following is the MHS code snippet to illustrate how the voter was interfaced.

```

BEGIN voter
PARAMETER INSTANCE = voter_0
PARAMETER HW_VER = 1.00.a
PORT voter_in0 = voter_sig0
PORT voter_in1 = voter_sig1
PORT voter_in2 = voter_sig2
PORT voter_out = sys_out_io
PORT sys_error = sys_err_io
END

```

As shown the voter is interfaced only to the inputs and outputs. Voter_sig0, voter_sig1 and voter_sig2 are internal connections connecting the VGPIO outputs to the voter inputs. Similarly sys_out_io and sys_err_io are connections to the outputs.

3 System Verification

3.1 MISRA-C compliance

MISRA-C(MISRA 2004) sets out guidelines for the use of C language in safety critical systems. Our software was checked for MISRA-C(MISRA 2004) rules by Abraxas Software's CodeCheck MISRA-C(MISRA 2004) rules checker. The tool basically parses the user's C code and lists warnings corresponding to each MISRA-C rule that has not been satisfied in the code. Following is the results obtained by the rules checker.

```

Abraxas Software (R) CodeCheck
Windows Version 1300 B1 DEMO
Copyright (c) 1988–2006, by
Abraxas Software Inc.
All rights reserved
Checking extended ANSI C file
counter0.c with rules from misra04.cc

```

```

counter0.c(9): Warning W0076: counter0.c(9):
Rule 76 (REQUIRED) 16.5 Functions with no
parameters shall be declared with parameter
type void.
counter0.c(9): Warning W0071: counter0.c(9) :
Rule 71 (REQUIRED) 8.1 Functions shall always
have prototype declarations. {DEFN}
counter0.c(9): Warning W0074: counter0.c(9) :
Rule 74 (REQUIRED) 16.4 identifiers given for
any of the parameters decl and/or defn must
be same
counter0.c(14): Warning W0071: counter0.c(14) :
Rule 71 (REQUIRED) 8.1 Functions shall always
have prototype declarations. {CALL}
File counter0.c check complete.

```

As shown above the applicable rules have been pointed out by the checker wherever required and we have verified our code for potential hazards at these points. For example line 9 warning W0076→This means that MISRA-C rule number 76 has not been satisfied in the code and so on. Similar results were obtained for the files counter1.c and counter2.c. The warnings with the level 'REQUIRED' have been taken care of but they appear because some part of the code is in header files which were not included (they were initially included which gave rise to warnings due to the presence of other header files which was not an indication of non safety critical code) during CodeCheck run. For example rule 71 at line 14 says that functions shall always have prototype declarations which has been implemented in the header files. Hence the warnings are not serious and the code can be considered to comply

with MISRA-C guidelines.

3.2 Microblaze compiler output

The Microblaze gcc compiler output for each of the software files (counter0.c, counter1.c, counter2.c) has been shown. The fact that the compilation took place without errors is justified due to the presence of the size of code (hex 6ac,6ac,6ec) and executable.elf file for each of the three files. The result also shows the code and data memory map used viz for the first code instance (counter0.c) the program memory and data memory start at 0x0000, for the second code instance (counter1.c) the program memory and data memory start at 0x4000 and for the third code instance (counter2.c) the program memory and data memory start at 0x8000.

Following is the Microblaze compiler (mb-gcc) output:

```

At Local date and time: Mon Apr 24 15:55:37 2006
Command xbash -q -c "cd /cygdrive/d/aSCSA1/;
/usr/bin/make -f system.make program; exit;"
Started...

```

```

mb-gcc -O2 microblaze_0/code/counter0.c -o
counter0/executable.elf \
-Wl,-defsym -Wl,_TEXT_START_ADDR=0x0000
-mno-xl-soft-mul -g -I./microblaze_0/include/
-L./microblaze_0/lib/ \
-xl-mode-executable \
mb-size counter0/executable.elf
text data bss dec hex filename
664 12 1032 1708 6ac counter0/executable.elf
mb-gcc -O2 microblaze_1/code/counter1.c -o
counter1/executable.elf \
-Wl,-defsym -Wl,_TEXT_START_ADDR=0x4000
-mno-xl-soft-mul -g -I./microblaze_1/include/
-L./microblaze_1/lib/ \
-xl-mode-executable \
microblaze_1/code/counter1.c:30:2: warning: no
newline at end of file
mb-size counter1/executable.elf
text data bss dec hex filename
664 12 1032 1708 6ac counter1/executable.elf
mb-gcc -O2 microblaze_2/code/counter2.c -o
counter2/executable.elf \
-Wl,-defsym -Wl,_TEXT_START_ADDR=0x8000
-mno-xl-soft-mul -g -I./microblaze_2/include/
-L./microblaze_2/lib/ \
-xl-mode-executable \
microblaze_2/code/counter2.c:30:2: warning: no
newline at end of file
mb-size counter2/executable.elf
text data bss dec hex filename
728 12 1032 1772 6ec counter2/executable.elf
Done.

```

3.3 Fault Injection

Various hardware faults were emulated both in software and hardware. Stuck at faults were emulated by inserting fault injection code in software (see software section). Stuck at fault means that the nodes are stuck either at logic high or low due to a configuration / bitstream error or fabrication defects or hardware design fault. Bit flip faults were emulated by inserting 'bit flip fault' code in VGPIO VHDL file (code hasn't been shown as file is too large to be included).

3.4 Board level Verification

The system was run on ML401 Xilinx Evaluation board by tying the outputs to LED's. Following is a listing of the user configuration file (UCF) created in

EDK.

```
Net sys_clk_pin LOC=AE14;
Net sys_clk_pin IOSTANDARD = LVCMOS33;
Net sys_rst_pin LOC=D6;
Net sys_rst_pin PULLUP;
## System level constraints
Net sys_clk_pin TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin
10000 ps;
Net sys_rst_pin TIG;
```

```
## FPGA pin constraints
Net sys_out_pin<0> LOC=E2;
Net sys_out_pin<0> IOSTANDARD = LVCMOS25;
Net sys_out_pin<1> LOC=E10;
Net sys_out_pin<1> IOSTANDARD = LVCMOS25;
Net sys_err_pin LOC=A5;
Net sys_err_pin IOSTANDARD = LVCMOS25;
```

The above listing shows connections of various signals (clk,rst,sys_out_pin(0),sys_out_pin(1),sys_err_pin) to FPGA pins (AE14,D6,E2,E10,A5) respectively. The constraints applied also show the I/O standards in effect, for example the sys_clk_pin has an I/O standard LVCMOS33 which means Low Voltage CMOS 3.3V. Similarly the sys_out_pins and sys_err_pin have I/O standard LVCMOS25 (Low Voltage CMOS 2.5V). The sys_rst_pin has been pulled up to Vcc (power viz logic high) since on the board the reset (sys_rst_pin) is active low (meaning that the system is reset when sys_rst_pin is logic low). This basically means that when reset button is not pressed the sys_rst_pin will be logic high and when reset button is pressed it will be logic low. The system level constraints have three commands namely TNM_NET, TIMESPEC and TIG whose explanation is as follows : TNM_NET means that sys_clk_pin is to be used in a timing specification. TIMESPEC defines the clock period viz 100MHz. TIG means that sys_rst_pin is to be ignored for a timing specification.

4 Conclusion

A triple modular redundant technique for reliable software execution in the event of hardware faults adhering to MISRA-C(MISRA 2004) rules was implemented and verified on a Virtex 4 FPGA (XC4VLX25) on the ML401 Xilinx Evaluation Board. The technique exhibited true concurrency in behaviour and operated correctly for long periods of time. This TMR work is one step in our investigation of reliable FPGA based programmable controller for safety critical applications.

5 Future Work

Presently the three copies of code are identical to each other. To guard against design faults in software it is possible that each of the three codes is different in that the functionality of the codes remain same however the implementation of this functionality differs from code to code. For example the simple counter implemented in this case can be implemented in three different ways viz

1. For loop
2. While loop
3. Do While loop

The issue with such methodology is the timing difference that may occur at outputs of each Microblaze System due to which outputs arrive at different instants of time thus giving an unstable operation in that the voter would signal error every now and then

which could cause system malfunction.

We propose to investigate in this direction of code diversity in TMR systems and challenges involved in design of voter for such circumstances.

References

- MISRA(2004), MISRA-C : 2004, Guidelines for the use of the C language in critical systems.
- IEC(1998), International Standard IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety related systems.
- B.W.Johnson(1989), Design and Analysis of fault tolerant digital systems.
- G.Berry and the Esterel team(2000), The Esterel v5.91 System Manual.
- F.Lima Kastensmidt, L.Sterpone, L.Carro, M.Sonza Reorda(2005), On the Optimal Design of Triple Modular Redundancy Logic for SRAM based FPGA's, *in* 'IEEE Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)', Vol. 2, pp. 1290-1295.
- Elmootazbellah Elnozahy, Rami Melhem, Daniel Mosse(2002), Energy Efficient Duplex and TMR Real Time Systems, *in* 'Proceedings of the 23rd IEEE Real Time Systems Symposium (RTSS'02)', pp. 256-266.
- Hyunki Kim, Hyung Joon Jeon, Keyseo Lee, Hyun-tae Lee(2002), The Design and Evaluation of All Voting Triple Modular Redundancy System, *in* '2002 IEEE Proceedings Annual of Reliability and Maintainability Symposium', pp. 439-444.

Author Index

Bergmann, Neil, 113

Cant, Tony, iii

Gärtner, J.U., 19

Garlan, David, 3

Griffiths, Alena, 23

Hansen, Klaus Marius, 35

Hunter, Bruce, 45

Kelly, Tim, 53

Mehta, Hardik, 113

Qureshi, Zahid H., 67

Reinhardt, Derek, 79

Schmerl, Bradley, 3

Squair, Matthew John, 93

Vasudevan, Venkatesh, 113

Waldeck, Peter, 113

Wells, Lisa, 35

Recent Volumes in the CRPIT Series

ISSN 1445-1336

Listed below are some of the latest volumes published in the ACS Series *Conferences in Research and Practice in Information Technology*. The full text of most papers (in either PDF or Postscript format) is available at the series website <http://crpit.com>.

Volume 53 - Conceptual Modelling 2006

Edited by Markus Stumptner, *University of South Australia*, Sven Hartmann, *Massey University, New Zealand* and Yasushi Kiyoki, *Keio University, Japan*. January, 2006. 1-920-68235-X.

Contains the proceedings of the Third Asia-Pacific Conference on Conceptual Modelling (APCCM2006), Hobart, Tasmania, Australia, January 2006.

Volume 54 - ACSW Frontiers 2006

Edited by Rajkumar Buyya, *University of Melbourne*, Tianchi Ma, *University of Melbourne*, Rei Safavi-Naini, *University of Wollongong*, Chris Steketee, *University of South Australia* and Willy Susilo, *University of Wollongong*. January, 2006. 1-920-68236-8.

Contains the proceedings of the Fourth Australasian Symposium on Grid Computing and e-Research (AusGrid 2006) and the Fourth Australasian Information Security Workshop (Network Security) (AISW 2006), Hobart, Tasmania, Australia, January 2006.

Volume 55 - Safety Critical Systems and Software 2005

Edited by Tony Cant, *University of Queensland*. April, 2006. 1-920-68237-6.

Contains the proceedings of the 10th Australian Workshop on Safety Related Programmable Systems, August 2005, Sydney, Australia.

Volume 56 - Vision in Human-Computer Interaction

Edited by Roland Goecke, Antonio Robles-Kelly, and Terry Caelli, *NICTA*. November, 2006. 1-920-68238-4.

Contains the proceedings of the HCSNet Workshop on the Use of Vision in Human-Computer Interaction (VisHCI 2006).

Volume 57 - Multimodal User Interaction 2005

Edited by Fang Chen and Julien Epps *National ICT Australia*. April, 2006. 1-920-68239-2.

Contains the proceedings of the NICTA-HCSNet Multimodal User Interaction Workshop 2005, Sydney, Australia, 13-14 September 2005.

Volume 58 - Advances in Ontologies 2005

Edited by Thomas Meyer, *National ICT Australia, Sydney* and Mehmet Orgun *Macquarie University*. December, 2005. 1-920-68240-6.

Contains the proceedings of the Australasian Ontology Workshop (AOW 2005), Sydney, Australia, 6 December 2005.

Volume 60 - Information Visualisation 2006

Edited by Kazuo Misue, Kozo Sugiyama and Jiro Tanaka. February, 2006. 1-920-68241-4.

Contains the proceedings of the Asia-Pacific Symposium on Information Visualization (APVIS 2006), Tokyo, Japan, February 2006.

Volume 61 - Data Mining and Analytics 2006

Edited by Peter Christen, *Australian National University*, Paul J. Kennedy, *University of Technology, Sydney*, Jiuyong Li, *University of Southern Queensland*, Simeon Simoff, *University of Technology, Sydney* and Graham Williams, *Australian Taxation Office*. December, 2006. 1-920-68242-2.

Contains the proceedings of the Australasian Data Mining Conference (AusDM 2006), Sydney, Australia. December 2006.

Volume 62 - Computer Science 2007

Edited by Gillian Dobbie, *University of Auckland, New Zealand*. January, 2007. 1-920-68243-0.

Contains the proceedings of the Thirtieth Australasian Computer Science Conference (ACSC2007), Ballarat, Victoria, Australia, January 2007.

Volume 63 - Database Technologies 2007

Edited by James Bailey, *University of Melbourne* and Alan Fekete, *University of Sydney*. January, 2007. 1-920-68244-9.

Contains the proceedings of the Eighteenth Australasian Database Conference (ADC2007), Ballarat, Victoria, Australia, January 2007.

Volume 64 - User Interfaces 2007

Edited by Wayne Piekarski, *University of South Australia*. January, 2007. 1-920-68245-7.

Contains the proceedings of the Eighth Australasian User Interface Conference (AUIC2007), Ballarat, Victoria, Australia, January 2007.

Volume 65 - Theory of Computing 2007

Edited by Joachim Gudmundsson, *NICTA, Australia* and Barry Jay *UTS, Australia*. January, 2007. 1-920-68246-5.

Contains the proceedings of the Thirteenth Computing: The Australasian Theory Symposium (CATS2007), Ballarat, Victoria, Australia, January 2007.

Volume 66 - Computing Education 2007

Edited by Samuel Mann, *Otago Polytechnic* and Simon Newcastle *University*. January, 2007. 1-920-68247-3.

Contains the proceedings of the Ninth Australasian Computing Education Conference (ACE2007), Ballarat, Victoria, Australia, January 2007.

Volume 67 - Conceptual Modelling 2007

Edited by John F. Roddick, *Flinders University* and Annika Hinze, *University of Waikato, New Zealand*. January, 2007. 1-920-68248-1.

Contains the proceedings of the Fourth Asia-Pacific Conference on Conceptual Modelling (APCCM2007), Ballarat, Victoria, Australia, January 2007.

Volume 68 - ACSW Frontiers 2007

Edited by Ljiljana Brankovic, *University of Newcastle*, Paul Coddington, *University of Adelaide*, John F. Roddick, *Flinders University*, Chris Steketee, *University of South Australia*, Jim Warren, *the University of Auckland*, and Andrew Wendelborn, *University of Adelaide*. January, 2006. 1-920-68249-X.

Contains the proceedings of the ACSW Workshops - The Australasian Information Security Workshop: Privacy Enhancing Systems (AISW), the Australasian Symposium on Grid Computing and Research (AUSGRID), and the Australasian Workshop on Health Knowledge Management and Discovery (HKMD), Ballarat, Victoria, Australia, January 2007.

Volume 72 - Advances in Ontologies 2006

Edited by Mehmet Orgun *Macquarie University* and Thomas Meyer, *National ICT Australia, Sydney*. December, 2006. 1-920-68253-8.

Contains the proceedings of the Australasian Ontology Workshop (AOW 2006), Hobart, Australia, December 2006.

Volume 73 - Intelligent Systems for Bioinformatics 2006

Edited by Mikael Boden and Timothy Bailey *University of Queensland*. December, 2006. 1-920-68254-6.

Contains the proceedings of the AI 2006 Workshop on Intelligent Systems for Bioinformatics (WISB-2006), Hobart, Australia, December 2006.