Safety case architectures to complement a contract-based approach to designing safe systems

S.A. Bates; The University of York; York, UK

I.J. Bate; The University of York; York, UK

R.D. Hawkins; The University of York; York, UK

T.P. Kelly; The University of York; York, UK

J.A. McDermid; The University of York; York, UK

Abstract

The benefits of using contracts when developing software for safety-critical systems are wide-ranging. Using contracts the cost of maintaining, reusing and changing/upgrading software components is lessened as developers may rework software components with knowledge of the constraints placed upon them. Our previous work has looked at how design and safety contracts may be generated for components. In this paper we extend this work to examine how design and safety contracts can be supported by a complementary safety case architecture and an appropriate means of gathering evidence.

Current approaches for producing safety cases are monolithic. Constructing safety cases in this way means that the benefits of having a modular architecture design with contracts is lost. In order to reflect the move towards contracts, a new way of constructing safety case arguments has been proposed. The approach is modular and features the use of safety case contracts. In this paper we show how this approach to developing safety cases can be integrated with the use of design and safety contracts to maximise the benefits of a modular approach. The paper illustrates how this can be achieved through a small example.

Introduction

Modularity has been proposed as the key element of the 'way forward' in developing systems. The primary reasons behind this include ease of change, obsolescence management and maintenance. This has however produced challenges in the way that we perform safety analysis and construct safety cases. This is because, classically, safety analysis and safety cases were based on global analysis of the system, whereas with modular systems the requirement is for local analysis as well as global analysis. This 'local' analysis is required so that to an existing system can have predictable and traceable consequences. To adopt this requires modularity in safety cases. The Goal Structuring Notation or GSN is used to argue that a system is safe to operate. Recently, the GSN has been extended to support modularity and this paper is intended to show how the principles of modular systems can be used to develop modular safety cases.

This paper is composed of four further sections. In the first section the modular principles will be derived and the requirements relating to the design of modular systems analysed. In the second section these modular system requirements will be applied to modular safety cases and safety case specific requirements derived. In the third section the modular requirements will be applied to a small example. Finally, in the fourth section this work will be concluded and future work identified.

Modular Principles and Design

The principles of modular systems lead to ideas such as reuse, modifications and upgrades which, when applied to cars, planes etc. are tangible. Applying these ideas to software is more difficult and in instances such as Ariane 5 has disastrous consequences. After Ariane 5 exploded on 4 June 1996, a fault was found in the software of the Inertial Reference System or SRI. This software had been reused from Ariane 4. The fault occurred due to functionality required by Ariane 4, but not by Ariane 5 being reused [13]. This is an example of an incident where the principles of modular systems were inadequately applied and as such we need to understand what modular systems are before we can discuss how to argue that they will be safe to operate.

A module is defined as being *'any more or less self-contained unit, which goes to make up a complete set, a finished article, etc.'* [7]. This definition gives us a sense that a module is independent from other modules and that it forms part of a collective.

Modular is defined as *'involving or consisting of modules or discrete units as the basis of design, construction, or operation; (also) intended to form part of such a system.'* [7]. From this definition we get a sense that a modular software system is one that is designed, constructed and operated using modules as the basis for the system. This however, implies that modular systems need more than just modules to work, but gives us an insight into the types of things that we are concerned with when discussing modular systems. These are:

- Modular systems are constructed from modules.
- Modules are as independent as possible.
- Modules simplify the design by breaking it into manageable chunks.
- Modular systems are operated through the combination of modules.

Even from this understanding we can see that modular software systems are developed by dividing system requirements between independent modules. Extending this further we arrive at the need for modules to be highly cohesive i.e. a module contains only things relevant to itself. Modular systems work by integrating it's modules to 'make up a complete set'. In software this is done via interfaces; the mechanism through which modules access each other's functions. In order to maintain module independence these are required to exhibit low coupling i.e. modular dependencies are kept to a minimum [9].

By ensuring that modules are highly cohesive and have low coupling we have started to realise how modular systems have the potential to tolerate change. To fulfil this potential we need to consider things such as:

- What modules are clients of other modules? [9]
- What assumptions may clients make of other modules? [9]
- What modules are likely to change?
- What is the impact of change?

The answers to questions like these allow the system developer to locate how changes to a module are likely to affect the system. Changes likely to occur in a system are usually due to maintenance, modifications or extensions to a module's functions. These are likely to occur throughout the life of a system, as such it is beneficial for the changes to have predictable and traceable consequences. For the purposes of this document a predictable change is thought of as covering peer level changes i.e. how a change in one module results in a change in another module in the same decomposition level and traceable changes relate to hierarchical changes i.e. how a change in one module can propagate to changes within higher level modules.

By reviewing these 'changes' it can be seen that in order to predict changes effectively and efficiently there is an implicit requirement to use a top-down decomposition approach to systems design, but more than this there is a need to use interfaces and track dependencies so that system changes have predictable and traceable consequences. If these interfaces and dependencies are used effectively and efficiently, then it should be possible to contain the change within a module and exhibit only a local or contained change to the modular system. This is an example of where it may be useful to use contracts. These are a mechanism through which the interfaces and dependencies between modules can be captured and documented. Ways of doing this are presented in [10][11][12] and [14], and are discussed later in this section.

Top-down decomposition approaches to modular software systems design are used to iteratively distribute requirements between modules. This step-wise approach to system design requires the designer to identify high-level system requirements and functions. These are then iteratively broken down until function specific modules can be designed or there is no cost-benefit in further decomposition. This approach and its use with software architectures will be discussed in the next section.

Modularity, Software Architectures and Contracts: Work conducted by Bass et al[2][4], Hofmeister et al [1], and Bate [6] etc. has contributed to the way we use software architectures. For clarity the following extracts have been included to define what software architectures are and give an idea of what they contain:

'Two main aspects of software architecture are that it provides a design plan – a blueprint – of a system and that it is an abstraction to help manage the complexity of a system.' [1]

'The structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.' [2]

These definitions convey two important things about software architectures:
- software architectures are designed through successive layers of abstraction and that
- software architectures are a representation of the software design.

The design of software through successive layers of abstraction is ideal for our needs when designing modular systems for reasons stated earlier. However, the notion that software architectures capture the design rationale is very important when considering change management [6]. By extending this further we can see that software architectures 'comprise software components' which, using the ideas we have already discussed are synonymous with software modules. These components have 'externally visible' properties or module interfaces and the components have 'relationships between them' or dependencies.

These interfaces and dependencies (as mentioned earlier) need to be well defined and captured so that change and potentially reuse of software modules can occur. The defining and capturing of interfaces and dependencies can be done through contracts [10]. A contract is *'a mutual agreement between two or more parties that something shall be done or forborne by both'* [7]. Applying this to computer based systems there are two generic types of contract. These are:

- Hierarchical
- Peer

The Hierarchical contracts are formed as the successive layers of abstraction are built up. They can be thought of as being generated from a top-down decomposition approach to system design i.e. the contracts are put in place to enforce a parental elements requirements

on the child elements properties. With this in mind a hierarchical contract could be thought of as:

A mutual agreement between a parent element and any child element specifying what will be done or forborne by both.

The Peer contracts cover such things as an application interacting with another application, or an object communicating with other objects. This type of contract is also applicable to a layered type approach to system design such as an IMA Three Layer stack [12]. Essentially they are defined to control the interactions that can occur between the peers. This type of contract can be defined as:

A mutual agreement between two or more peer level components specifying what will be done or forborne by both.

In [14] it was shown how both of these contract types could be derived for a safety-critical software architecture. Importantly in [14] the idea of a safety contract was introduced. These contracts constrain the interactions, which occur between objects and hence can ensure system behaviour is safe. These safety contracts are made up of pre and post conditions. Pre-conditions must be true before the operation call is made and post-conditions must be true by the execution of the call. The properties of an interaction that we are interested in from a safety perspective are function, timing and value. Analysis of each of these aspects results in requirements that are included in the safety contract.

Summary: Both software architectures and contracts are independently useful to correctly designing functioning systems. Software architectures amongst other things allow the developer to trace requirements throughout the design. Contracts are useful on a peer level as they explore the external properties of a module and what they require from other modules i.e. pre-conditions and what they guarantee in return i.e. post conditions. In a hierarchical sense they are used to ensure that regardless of implementation the combined child modules will meet at least the minimum requirements of its parent/containment module. By combining software architectures and contracts we have a method through which safe systems can be designed and built.

This section has introduced various requirements to consider when designing modular systems and software architectures. These are:
- Modules are required to be as independent as possible.
- Modular systems and software architectures need to be constructed top-down to improve the traceability and predictability of change.
- Modules need to be highly cohesive and exhibit low coupling.
- Modules need well-defined interfaces and all module dependencies need to be captured. It has been shown that this could be done with contracts[14].

In the next section these requirements will be used as a guide to derive modular safety cases and safety case architectures that are tolerant and robust to change.

Safety Case Construction

Safety cases are a means through which an argument is presented to a certifying body that a developed system is safe to operate. The Goal Structuring Notation or GSN [8] has been devised to allow these arguments to be constructed through a graphical argumentation notation. This 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 ("solved by") sub-goals until a point is reached where claims can be supported by direct reference to available evidence. 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 [8].
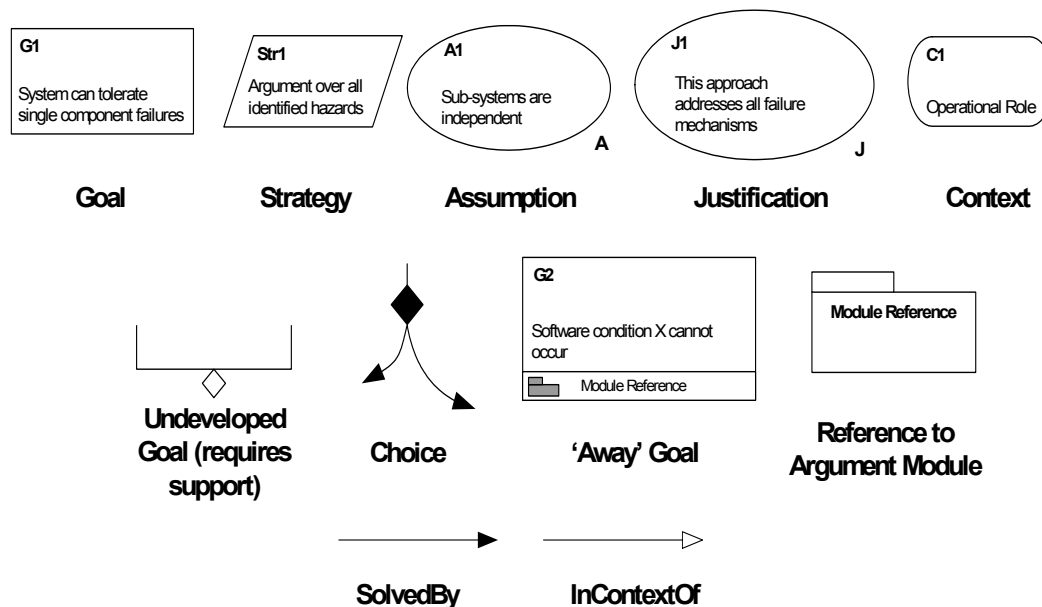

Figure 1 – Principal Elements of the Goal Structuring Notation

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 in one place as a single artefact rather than as a series of modularised interconnected arguments. Recently, there has been a push to move towards a modular approach to arguing safety. For this reason GSN has been extended with the introductions of Away Goals and Argument Modules and these are the focus of the next section.

Modular Principles and the Safety Case: The analysis of modular systems and software architectures previously, could potentially provide the stimulus for the creation of modular safety cases and safety case architectures. The following definition has been adapted from the software architecture definition found in [2]:

'*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.*' [3]

From this definition it can be seen that when thinking about modular safety cases and safety case architectures we should approach them with the same mindset as modular systems and software architectures. To proceed, further definitions are required. The symbols shown in figure 1 are the core GSN symbols. The symbol 'reference to argument module' allows 'the high level organisation of the safety case into component of argument and evidence'. By allowing modules to exist in a safety case we need to ensure two things, these modules must

independently stand up to scrutiny and their dependence on other modules captured. This is done by first using a modular interface shown in figure 2 to capture 'the externally visible properties of that module' i.e. the goals requiring or offering support for other modules. Once defined a claim matching process can occur. When 'the interdependencies that exist between' modules are identified these are captured using a safety case contract as shown in figure 3

Argument Module:			
	Identifier	Summary	Module
Goals Addressed			
Solution Presented			
Context Defined			
Goals Requiring Support			
Inter-module dependencies — Goals			
Solutions			
Context			

Safety Case Module Contract			
Participant Modules			
Participating Modules:			
Goals Matched Between Participant Modules			
Goal	Required by	Addressed by	Goal
Collective Context and Evidence (Solutions) of Participant Modules held to be consistent			
Context		Evidence	
Resolved Away Goal, Context and Solution References between Participant Modules			
Cross Referenced Item	Source Module		Sink Module

Figure 2 – Safety Case Module Interface   Figure 3 – Safety Case Module Contract

For a modular safety case to be constructed it must follow the same requirements as for modular system construction. By applying these requirements we arrive at the following understanding:

- **Modules must be as independent as possible** – This is important as it means that a safety case module can de developed without the needing to consult other modules and therefore, can potentially support work divisions and contractual boundaries. As long as these modules are wisely chosen it should be possible to identify and document what is required for a module to fulfil its work and contractual boundaries. This places a demand on the safety case architecture to support, where possible, these sorts of boundaries in the safety case.

- **Modules must exhibit high cohesion and low coupling** – The requirements for high cohesion and low coupling places an implicit demand on the modular safety case that a module's objectives should be naturally consistent and that cross referencing across modular boundaries should be minimised as much as possible.

- **Modular safety cases and safety case architectures must be constructed top-down** – Due to the GSN implicitly being hierarchical this requirement is easily met. However, it should be noted that just because the GSN is implicitly hierarchical does not infer that any goal structure is modular.

- **Modules must have well defined interfaces** – This is necessary for two reasons. Firstly, it gives a safety case module the ability to specify what goals it requires support for and what goals it is offering support to. This allows the safety case developer to proceed with the modular argument and revisit those goals when the modules are combined to form the safety case. Secondly, it allows the safety case architecture to support future expansion. By having explicit modular boundaries another module added, for example, as part of a life cycle upgrade, knows what it must guarantee to an existing module if it going to offer/provide support for that module and hence form part of the overall safety case.

- **All modular dependencies must be captured** – This is important when managing change. If a change is made to an existing safety case module then the capturing of dependencies can be use to trace and predict what effects it will have with regards to the rest of the modules in the safety case.

From these requirements it is clear that modular safety cases have to be carefully considered before composition. It is therefore necessary to consider what you require from your safety case before diving head first into the construction of a modular safety case. In the next section these requirements will be used with a small example.

Summary: This section has shown how the requirements for a modularly designed system can be applied to a modular safety case. These will be explored further in the next section through a small example.

Application to a Stores Management System Example

In this section we will show how the requirements: *'modules must be as independent as possible'*, *'modules must exhibit high cohesion and low coupling'* etc. can be used to drive the safety case architecture for a Stores Management System (SMS). The SMS, illustrated in figure 4, is a system that is used on an aircraft to control the release, jettison and selection of stores such as missiles or fuel. This particular example was developed in [14]. In [14], a trade-off analysis was used to obtain the software architecture shown in figure 4. A trade-off analysis process is used when differing system objectives need to be supported by the design. In [14], it was shown how safety objectives could be used as part of this trade-off analysis. The outcome of this trade-off analysis was a number of hierarchical and peer design and safety contracts. The exact derivation details are out of the scope of this document, for further details refer to [14].
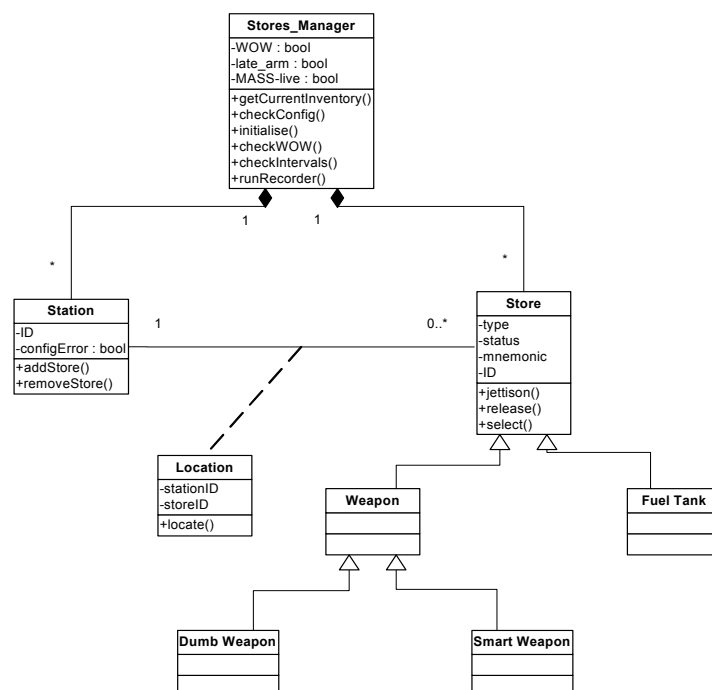

Figure 4 – A peer level decomposition for the Stores Management System

There are a number of possible ways to argue that the SMS shown in figure 4 is safe to operate. In this paper the safety case architecture are based on the software architecture. By doing this we still have choices to make about how we are going to construct the modular safety case. The safety case architectures shown in figures 5 and 6 are examples of these

choices. Both architectures show how the SMS is composed of the modules Stores Manager, Station and Store. Figure 5 shows an example that corresponds directly the software architecture. Whereas figure 6 shows an example where the arguments about the interactions between the software architecture modules are collected together to form the InteractionsArg safety case module. Both the safety case architectures could potentially be used to argue the safety of the system. Therefore, we need to decide which one best supports the way we want to argue the safety of the SMS.
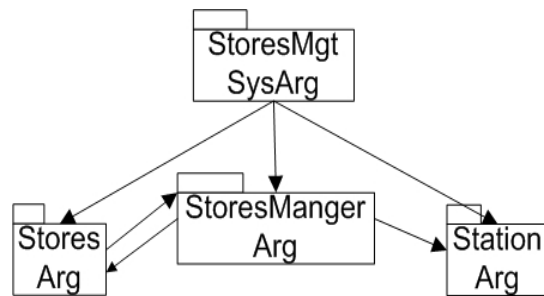

Figure 5 – A possible Safety Case Architecture relating to the Stores Management System
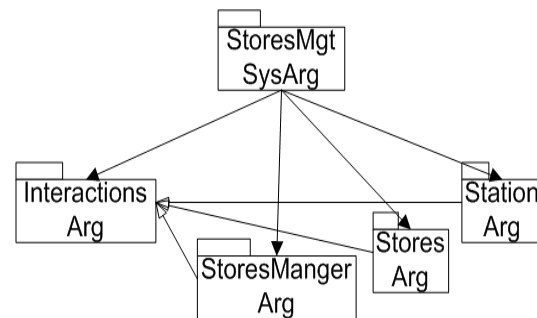
Figure 6 - Another possible Safety Case Architecture relating to the Stores Management System

In this paper modular system requirements such as *'modules must be as independent as possible'* have been derived and how these can be applied to safety case architectures examined. We will now show how the requirement *'modules must exhibit high cohesion and low coupling'* can be used to select an appropriate safety case architecture to argue the safety of the SMS shown in fig 4. Both the safety case architectures in figures 5 and 6 can be seen to be highly cohesive. The safety case architecture shown in figure 5 is highly cohesive as the argument relating to all aspects including the inter-modular interactions of the individual system modules are contained within individual safety case modules. However, the safety case architecture shown in figure 6 introduces an *'InteractionsArg'* module. By introducing this module the arguments about the inter-modular interactions that would be distributed between the safety case modules of figure 5 are contained in one module. Therefore, when the safety case architectures are reviewed for their support of the 'low coupling' requirement the main distinctions between the safety case architectures are identified. The modules in figure 5 will exhibit more dependencies between each other than the modules in figure 6. This is because each of the safety case modules shown in figure 5 would have to independently argue about all the interactions that occur between the modules. Each safety case module would also have to make claims about how the individual interactions are safe and make reference to similar claims in the other safety case modules, which could lead to numerous dependencies and safety case contracts (figure 3) being formed. However, the arguments underlying the safety case modules in figure 6 can be developed independently of interactions by making claims such as *'All Hazardous interactions between modules are safe'* once, and assuming that support for this claim will be provided by the *'InteractionsArg'* module. As such the dependencies that will exist in the safety case architecture in figure 5 are minimised by the safety case architecture in figure 6 and is the best choice to argue the safety of the SMS.

As stated previously when designing modular systems there are two types of contracts, hierarchical and peer. The hierarchical contracts are formed to ensure that a child element will meet the minimum requirement of it's parent element. The GSN's hierarchical nature allows the way the child element meets the requirements to be easily captured. This is done in two stages. Firstly, the safety case module interface (figure 2) between the parent and child elements will be made from the requirements of the hierarchical modular system contract. In the parent safety case module these requirements will become goals/claims that require support. In the child safety case module these will become top-level goals that it will be required to satisfy. Secondly, a safety case contract (figure 3) is made when the child safety

case module can provide the required support for the parent safety case module's requirements or goals. The peer contracts are captured in a modular system to control and constrain it's modules interactions. In [14] it was shown how safety contracts could be formed to mitigate against hazardous interactions between system modules. The InteractionsArg safety case module shown in figure 6 could be used to support this style of contract. An in depth discussion of how this argument is formed is out of the scope of this paper, so a simplistic overview will be presented here. In the InteractionsArg safety case module the top-level claim will be *'There are no hazardous interactions between modules'*. Presenting an argument about the correctness, completeness and validity of the safety contract would then be used to support this.

Summary: In this section we have reasoned about how we could argue the safety of a modular system using a modular safety case. We have also discussed how the contracts made throughout the modular system design can be supported by a modular safety case.

Conclusions

This paper has analysed work relevant to the modular construction of systems and has used this to develop ways of reviewing modular safety cases. It has been shown that managing change is perhaps the most important requirement in the construction of a modular safety case. This can be done providing the safety case's modular composition is sensibly chosen. The work has been performed in conjunction with complementary work on the design architecture. The future work on modular safety cases will look at developing guidelines to aid their development and decomposition. These will focus on how the safety case can be constructed to have predictable and traceable consequences.

References

[1] Hofmeister, C., Nord, R. and Soni, D., Applied Software Architectures. Addison Wesley: October 1999

[2] Bass, L., Clements, P. and Kazman, R., Software Architectures in Practice. Addison Wesley: 1998

[3] Kelly, T.P., Managing Complex Safety Cases. Proceedings of the Eleventh Safety-Critical Systems Symposium, Bristol, UK. February 2003

[4] Clements, B., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J., Documenting Software Architectures – Views and Beyond. Addison Wesley: 2003

[5] Kelly, T., Concepts and Principles of Compositional Safety Case Construction. COMSA/2001/1/1, 2001.

[6] Bate, I., and Kelly, T., Architectural Considerations in the Certification of Modular Systems. In proceeding of SAFECOMP 2003.

[7] OXFORD ENGLISH DICTIONARY. 2002.

[8] Kelly, T., Arguing Safety – A Systematic Approach to Managing Safety Cases. York: Department of Computer Science, The University of York, 1998.

[9] Pooley, R., and Stevens, P., Using UML. Addison Wesley: 1999.

[10] Meyer, B., Eiffel The Language. Prentice Hall: 1992.

[11] Hawkins, R. and McDermid, J., Performing Hazard and Safety Analysis of Object Oriented Systems. In Proceedings of 20th ISSC, 2002: p.802-811.

[12] Conmy, P., McDermid, J., High Level Failure Analysis for Integrated Modular Avionics. 6th Australian Workshop on Industrial Experience with Safety Critical Systems and Software, Brisbane, Australia, June 2001.

[13] Lions, J., Ariane 5 Flight 501 Failure Report by the Inquiry Board. Paris: ESA, 1996.

[14] Bate, I., Bates, S., and Hawkins, R., A Contract Based Approach to Designing Safe Systems. Submitted to SAFECOMP 2003.

Biographies

S.A. Bates, MEng, Research Associate, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 433385, Fax: 0044 (0) 1904 432708, Email: simon.bates@cs.york.ac.uk

Simon Bates has been a Research Associate in the BAE SYSTEMS funded Dependable Computing Systems Centre (DCSC) at the University of York since October 2002. This is his first role since leaving the University of Manchester in 2002 where he attained a MEng (Hons) in Electronic Systems Engineering.

Dr. I.J. Bate, Ph.D, Senior Research Associate, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 432786, Fax: 0044 (0) 1904 432708, Email: iain.bate@cs.york.ac.uk

Iain Bate has been working as a Research Associate since 1994 and is now a Senior Research Fellow. His research interests include scheduling and timing analysis, design for safety including architecture trade-off techniques, and the use of optimisation to derive appropriate design solutions.

R.D. Hawkins, MSc, Research Associate, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 433385, Fax: 0044 (0) 1904 432708, Email: richard.hawkins@cs.york.ac.uk

Richard Hawkins has been a Research Associate in the BAE SYSTEMS funded Dependable Computing Systems Centre at the University of York since November 2001. He is researching the use of object oriented techniques for safety critical systems. Before taking up his current role, he attained an MSc in Information Systems from the University of Liverpool and worked as a safety adviser for British Nuclear Fuels since 1997.

Dr Tim Kelly, Lecturer of Software and Safety Engineering, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 43 2764, Fax: 0044 (0) 1904 432708, Email: tim.Kelly@cs.york.ac..uk

Dr Tim Kelly (MA DPhil) is a lecturer in software and safety engineering within the Department of Computer Science at the University of York. He is also Deputy Director of the Rolls-Royce Systems and Software Engineering University Technology Centre funded at York. His expertise lies predominantly in the areas of safety case development and management. Tim has provided extensive consultative and facilitative support in the production of acceptable safety cases for companies from the medical, aerospace, railways and power generation sectors. He has published a number of papers on safety case development in international journals and conferences and has been an invited panel speaker on software safety issues.

Prof J.A. McDermid, Professor of Software Engineering, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, Tel: 0044 (0) 1904 432786, Fax: 0044 (0) 1904 432708, Email: john.mcdermid@cs.york.ac.uk

John McDermid has been Professor of Software Engineering at the University of York since 1987 where he runs the high integrity systems engineering (HISE) research group. HISE studies a broad range of issues in systems, software and safety engineering, and works closely with the UK aerospace industry. Professor McDermid is the Director of the Rolls-Royce funded University Technology Centre (UTC) in Systems and Software Engineering and the BAE SYSTEMS-funded Dependable Computing System Centre (DCSC). He is author or editor of 6 books, and has published about 250 papers.