ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT

Reliability Engineering and System Safety 81 (2003) 303–324

RELIABILITY ENGINEERING & SYSTEM SAFETY
www.elsevier.com/locate/ress

Architectural considerations in the certification of modular systems

Iain Bate*, Tim Kelly

Department of Computer Science, University of York, York YO10 5DD, UK

Abstract

Modular system architectures, such as integrated modular avionics (IMA) in the aerospace sector, offer potential benefits of improved flexibility in function allocation, reduced development costs and improved maintainability. However, they require a new certification approach. The traditional approach to certification is to prepare monolithic safety cases as bespoke developments for a specific system in a fixed configuration. However, this nullifies the benefits of flexibility and reduced rework claimed of IMA-based systems and will necessitate the development of new safety cases for all possible (current and future) configurations of the architecture. This paper discusses a modular approach to safety case construction, whereby the safety case is partitioned into separable arguments of safety corresponding with the components of the system architecture. Such an approach relies upon properties of the IMA system architecture (such as segregation and location independence) having been established. The paper describes how such properties can be assessed to show that they are met and trade-offs performed during architecture definition reusing information and techniques from the safety argument process.
© 2003 Elsevier Ltd. All rights reserved.

Keywords: Integrated modular avionics; Goal structuring notation; Modular systems; Architecture; Design assessment

1. Introduction

Extensibility, adaptability and resilience to change are increasingly desirable characteristics for many safety critical systems. Modular system architectures, such as integrated modular avionics (IMA) in the aerospace sector, are being seen by many as offering the potential benefits of improved flexibility in function allocation, reduced development costs and a means of managing the ever present issues of technology obsolescence and update.

The characteristics desirable in these systems (such as change resilience and timeliness) cannot easily be retrofitted into a system design. Ability to exhibit these characteristics depends to a large extent on the architecture of the system in question (e.g. concerning the partitioning, communication mechanisms and scheduling policies). Therefore, consideration must be given during architecture definition as to how these objectives will be satisfied. In this paper we highlight how dependability and maintainability criteria can be elaborated and considered during the architecture definition process. In particular, we describe how the exploration of alternative satisfaction arguments for these criteria can enable assessment of architectural tradeoffs.

One of the most significant problems posed by the adoption of modular systems in safety critical applications lies in their 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 IMA is that there is through-life (and potentially run-time) flexibility in the system configuration. An IMA system can support many possible mappings of the functionality required to the underlying computing platform.

In constructing a safety case for IMA 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 within the architecture. However, this nullifies the benefit of flexibility in using an IMA 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 arguments that has a correspondence with the structure of the underlying system architecture. However, to create such arguments requires a system architecture that has been designed with explicit consideration of enabling properties

* Corresponding author. Tel.: +44-1904-432786; fax: +44-1904-432708.

E-mail addresses: iain.bate@cs.york.ac.uk (I. Bate), tim.kelly@cs.york.ac.uk (T. Kelly).

0951-8320/03/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S0951-8320(03)00094-2

such as independence (e.g. including both non-interference and location 'transparency'), increased flexibility in functional integration, and low coupling between components. In this paper we describe how a modular safety case can be established and managed.

For a modular safety critical system architecture, a modular, change resilient, safety case architecture is also desirable. However, it is extremely difficult for system architects to reason about the implications of their choices in defining the system architecture on the possible configurations of the certification case without a means of defining and reasoning about safety case architecture. In the paper we highlight the dependency between the possible organisation of the safety case and the choices made during system architecture definition and how recognition of these dependencies can be factored into the architecture trade-off analysis process.

We begin by presenting a process for the evaluation of required architectural quality attributes in Section 2. This process is illustrated by means of an example, evaluating a number of specific quality attributes, within Section 3. Section 4 defines the concept of modular safety cases and discusses how evaluation of *system* architecture can be extended to consider the degree to which a modularised safety case can be supported. Evaluation of the modifiability of any proposed *safety case* architecture is discussed within Section 5. Section 6 illustrates the modular approach to safety case construction by means of an example derived from consideration of IMA architectures. Finally, conclusions are presented in Section 7.

2. Evaluating required qualities during system architecture definition

The system architecture is of paramount importance when producing a system because it dictates how much effort is involved in designing, integrating, verifying, maintaining and reusing the system and its component parts. There is no architectural design that is best in all circumstances. Instead for a particular system being developed, there are architectural designs that will meet some of these objectives better than others. An inappropriate architectural design can make achieving the desired results difficult and costly. As systems are becoming complex, the need to perform architectural analysis is becoming apparent but to date few techniques exist and those that do exist are still evolving to industrial strength solutions [1]. Part of our work is looking at how architectures should be designed and analysed.

As part of designing an appropriate system architecture it is important to consider the following activities

1. *Derivation of choices*: identifies where different design solutions are available for satisfying a goal.
2. *Manage sensitivities*: identifies dependencies (i.e. contracts) between components such that consideration of whether and how to relax them can be made. A benefit of relaxing dependencies could be a reduced impact to change.
3. *Evaluation of options*: allows assessment criteria to be derived whose answers can be used for identifying solutions that do/do not meet the system properties, judging how well the primary objectives, that is property requirements are met, and indicating where refinements of the design might add benefit. A similar process can be followed for secondary objectives. The distinction between primary and secondary objectives is that primary objectives are properties that have to be met (e.g. correct functional behaviour) whereas secondary objectives are characteristics that should be met as well as possible but a failure to meet them does not affect the system's operation (e.g. supporting managed change).
4. *Influence on the design*: identifies constraints on how components should be designed to support the meeting of the system's overall objectives. This leads to definitions of functionality, abstractions and interfaces.
5. *Influence on the design*: identifies constraints on how components should be designed to support the meeting of the system's overall objectives.

2.1. Analysing different design solutions and performing trade-offs

Fig. 1 provides a diagrammatic overview of the proposed method. Stage (1) of the trade-off analysis method is

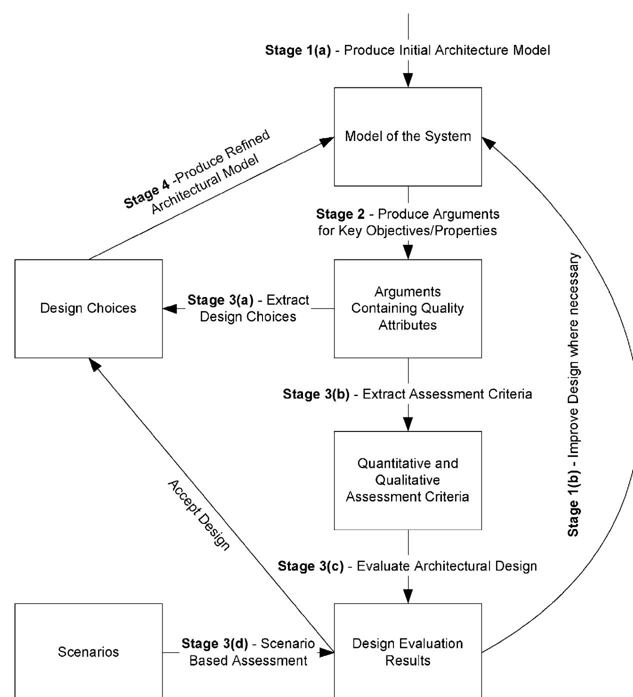

Fig. 1. Overview of the method.

producing a model of the system to be assessed. This model should be decomposed to a uniform level of abstraction. Currently our work uses the class diagrams from UML [2] for this purpose, however, it could be applied to any modelling approach that clearly identifies components and their interactions (interaction is considered to be the link and interfaces between two components).

In stage (2), arguments are then produced for each interaction to a corresponding (but lower so that the impact of later choices can be made) abstraction level than the system model (an overview of goal structuring notation (GSN) symbols is given in Section 2.3, further details of the notation can be found in Ref. [3]). The arguments are derived from the top-level properties and objectives of the particular system being developed. The properties often of interest are lifecycle cost, dependability, and maintainability. Clearly these properties can be broken down further, e.g. dependability may be decomposed to reliability, safety, timing (as described in Ref. [4]). Safety may further involve providing guarantees of independence between functionality. In practice, the arguments should be generic or based on patterns where possible. The objectives often of interest are managed change, ease of integration and ease of verification. Stage (3) then uses the information in the argument to derive options and evaluate particular solutions via assessment criteria.

When evaluating particular design solutions, the results from more than one assessment criteria have to be traded-off because a design modification that suits one-assessment criterion may not suit another. For example, introducing an extra processor may reduce the load across the processors in the system making task schedulability easier. However, it may increase the load on the communications bus making message schedulability more difficult and more power would be used. Initially when the design is in its early stage the evaluation may have to be qualitative in nature but as the design is refined then quantitative assessment may be used where appropriate. Representative scenarios have been used as part of this activity to evaluate solutions.

Based on the findings of stage (3), the design is modified to fix any problems that are identified—this may require stages (1)–(3) to be repeated to show the revised design is appropriate. When this is complete and all necessary design choices have been made, the process returns to stage (1) where the system is then decomposed to the next level of abstraction using guidance from the arguments. Components reused from another context could be incorporated as part of the decomposition. Only proceeding when design choices and problem fixing are complete is preferred to allowing trade-offs across components at different stages of decomposition because the abstractions and assumptions are consistent.

Currently the refinement of the design (stage (4) of the process) and the refinement of the model (stage 1(b)) are currently performed manually to decide how best to decompose the current architecture to the next level. Future work will look at using a combination of the current approach and multi-criteria optimisation to address the problem.

The method will be illustrated by means of an example presented in Section 3.

2.2. Previous work on architectural analysis

A technique (the architecture trade-off analysis method (ATAM)) [5] for evaluating architectures for their support of architectural qualities, and trade-offs in achieving those qualities, has been developed by the Software Engineering Institute. Our proposed approach is intended for use within the nine-step process of ATAM. The approach is largely based on deriving quality attributes from overall system objectives, and then turning the quality attributes into questions that can be asked of the architecture and its designers.

There are numerous other techniques that follow similar processes—for example goal question metrics (GQM) [6], and quality function deployment (QFD) [7]. GQM could be used within our proposed approach to help derive assessment criteria from the objectives contained within the goal structured attribute arguments. Again, we can use some of the findings when converting our arguments into assessment criteria—i.e. transitioning from stage 3(b)–(c). The differences between our strategy and other existing approaches, e.g. ATAM, include the following.

1. The techniques used in our approach are already accepted and widely used (e.g. nuclear propulsion system and missile system safety arguments) [2], and as such processes exist for ensuring the correctness and consistency of the results obtained.
2. The technique offers: (a) strong traceability and a rigorous method for deriving the attributes and assessment criteria (this is considered to be more rigorous than questions) with which designs are analysed; (b) the ability to capture design rationale and assumptions which is essential if component reuse is to be achieved.
3. Information generated from their original intended use can be reused, rather than repeating the effort.
4. The method is equally intended as a design technique to assist in the evaluation of the architectural design and implementation strategy as it is for evaluating a design at particular fixed stages of the process.

As stated above, a defining characteristic of our proposed method is the use of a well established argumentation notation (GSN) to explore alternative satisfaction arguments for desirable architectural criteria. Before providing an example walkthrough of the method a brief introduction to GSN is given in Section 2.3.

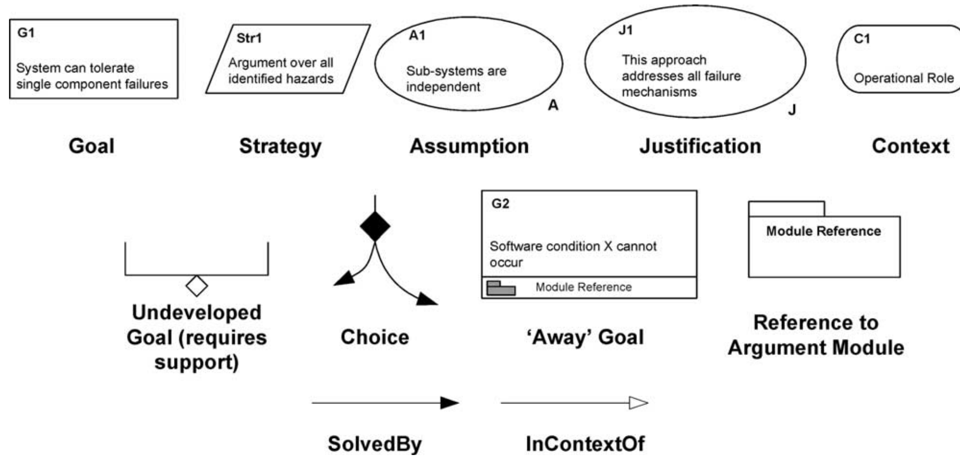

Fig. 2. Principal elements of the goal structuring notation.

2.3. Overview of GSN

The GSN [8]—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 Fig. 2 (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 Ref. [8].

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. Later in the paper we show how GSN has been extended to explicitly represent interrelated modules of safety case argument.

3. Example—simple control system

The example presented in this section is used throughout the course of this work to illustrate the techniques presented. The example is a continuous control loop that has health monitoring to check for whether the loop is complying with the defined correct behaviour (i.e. accuracy, responsiveness and stability) and then takes appropriate actions if it does not. The platform is scheduled using static scheduling. Static scheduling is where each task in the system is assigned its own slot(s) that are always executed at the same time. The sequence of slots is continually repeated.

At the highest level of abstraction the control loop (the architectural model of which is shown in Fig. 3) consists of three elements; a sensor, an actuator and a calculation stage (with feedback from actuator to sensor). It should be noted that at this level, the design is abstract of whether the implementation is achieved via hardware or software. The requirements (key safety properties to be maintained are signified by (S), functional properties by (F) and non-functional properties by (NF), and explanations, where needed, in italics) to be met are

1. the sensors have input limits (S) (F);
2. the actuators have input and output limits (S) (F);
3. the overall process must allow the system to meet the desired control properties, i.e. responsiveness (dependent on errors caused by latency (NF)), stability (dependent on errors due to jitter (NF) and gain at particular frequency responses (F)) [6] (S);
4. where possible the system should allow components that are beginning to fail to be detected at an early stage by comparison with data from other sources (e.g. additional sensors) (NF). Early recognition would

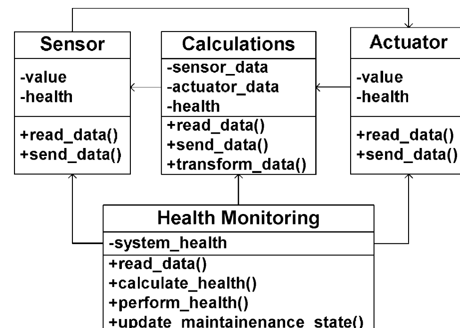

Fig. 3. Class diagram for the control loop.

allow appropriate actions to be taken including the planning of maintenance activities.

In practice as the system development progresses, the component design in Fig. 3 would be refined to show more detail. For reasons of space only the *calculation–health monitor* interaction is considered. In Section 3.1 we will describe how satisfaction of architectural criteria concerned with this interaction can be examined and tradeoffs identified using the method as outlined in Section 2.

3.1. Overall argument that the system meets its objectives

This section presents the argument that the system meets its objectives. At higher levels of architectural decomposition the arguments can be generic in nature. That is, the arguments only become specific when the design of the system is at a low level. For the purposes of this paper, higher-level issues are to be considered, therefore, the arguments are to be generic in nature.

The overall argument that the system meets its objectives is shown in Fig. 4. The argument shows how the overall objective is split into three parts; that the system is dependable as judged by the appropriate regulatory authority, the system is maintainable (i.e. upgrades can be performed with a minimum of effort) and the lifecycle costs are minimized. Whilst some definitions of dependability include maintainability [4] this is typically from a reliability and availability perspective. Our concern with maintainability is principally with *adaptation* of the system. The maintainability and dependability arguments are explored further in the following sections.

In the arguments, the leaf goals (generally at the bottom) have a diamond below them that indicates the development of that part of the argument is not yet developed. In the case of the argument in Fig. 5, the cost argument is left undeveloped.

3.2. Dependability argument

For a full decomposition of dependability we refer the reader to Ref. [4]. Fig. 5 presents a breakdown of the dependability property highlighting the particular concerns of timing, safety, reliability and function [4]. The timing

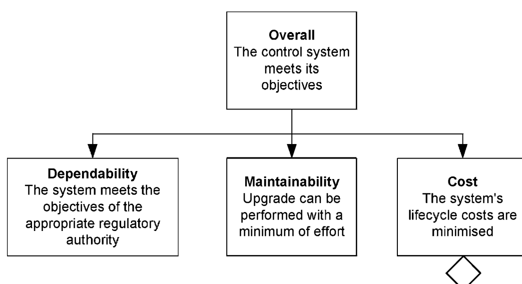

Fig. 4. Overall argument.

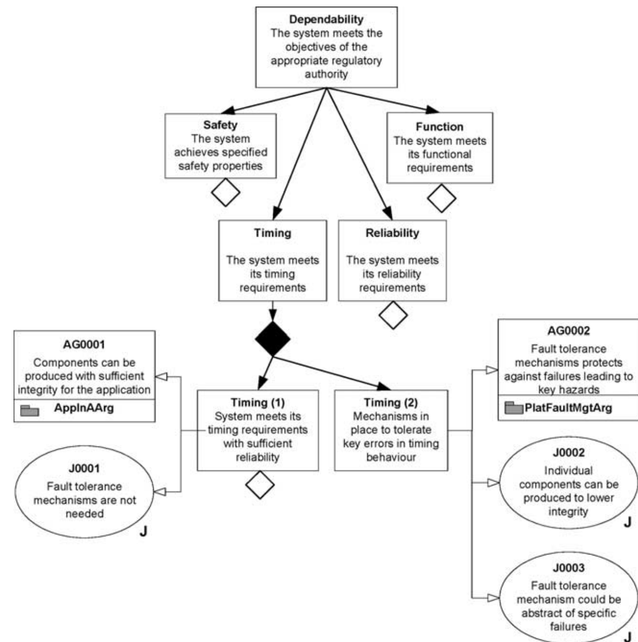

Fig. 5. Top level argument for dependability.

property is then broken down further. This shows a choice (a choice is depicted by a black diamond in the arguments) emerging from goal Timing that is often made between having highly-dependable components (goal Timing(1)) or less dependent components but with appropriate fault tolerance mechanisms in place (goal Timing(2)).

For each of the choices shown in Fig. 5 associated assumptions and justifications are stated. For example, related to goal Timing(1) an away goal, AG0001, has to be supported that it is possible to produce components of sufficient integrity and justify this is the case. The justification for doing this, J0001, is that the extra complication of fault tolerance is not needed.

The argument shown in Fig. 5 uses a new extension to GSN—the 'Away' Goal (elements AG0001 and AG0002). Away Goals are used within the arguments to denote claims that must be supported but whose supporting arguments are located in another part of the safety case. The name of the part of the safety case in which this supporting argument is expected to be found is shown at the bottom of the away goal symbol. For example, in Fig. 5 the use of the away goal AG0001 as context for Goal Timing(1) shows that the timing claim is made on the assumption that the component integrity claim will be satisfied in the argument labelled ApplnAArg. The ability to represent inter-argument dependencies in this way is crucial to being able to model modular safety cases. Section 4 discuss the modelling of these dependencies in more detail.

3.2.1. Goal timing(2)—tolerance of errors in timing

From an available argument pattern, the argument in Fig. 6 was produced that decomposes the goal Timing(2). Typically, this is achieved by the system having health

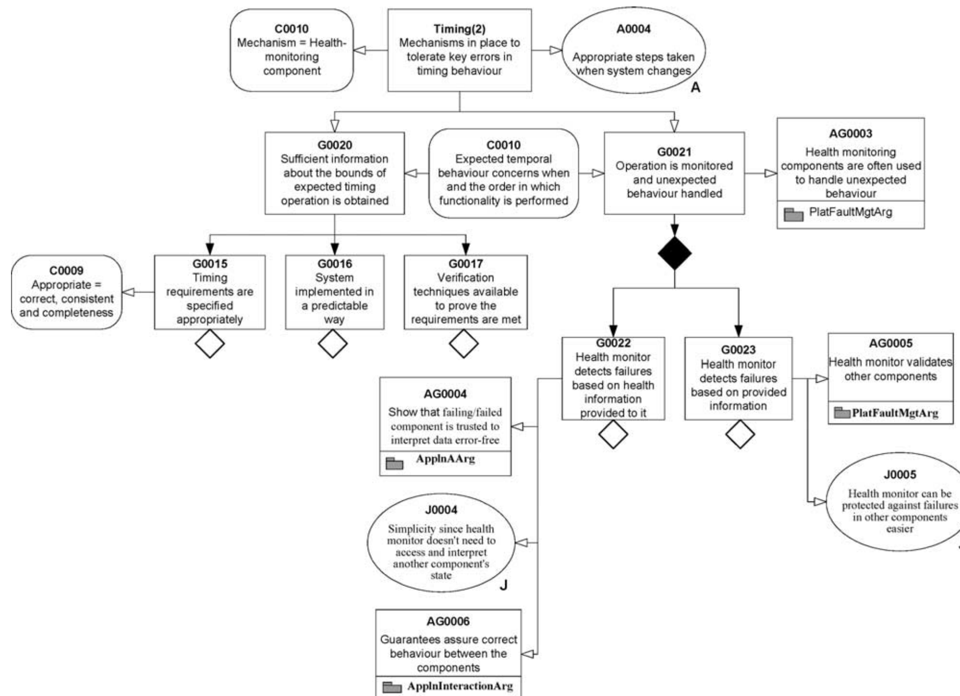

Fig. 6. Argument for tolerating errors in timing behaviour.

monitoring components—away goal AG0003. Fig. 6 shows how the argument is split into two parts. Firstly, evidence has to be obtained using appropriate verification techniques that the requirements are met in the implementation, e.g. when and in what order functionality should be performed. Secondly, the *health monitor* checks for unexpected behaviour. There are two ways in which unexpected behaviour can be detected—just one of the techniques could be used or a combination of the two ways. The first way is for the health monitor component to be functionally integrated with the calculation component. That is the health monitor component relies entirely on the results of the internal health monitoring of the *calculation* component to indicate the current state of the calculations. If the health monitor component is to rely on the calculation component then, a guarantee needs to be produced between the components that would act as a contract which could be re-visited when either of the components changed-away goal AG0006. Once the nature of the contract has been determined, there is a great deal of work available for specifying and managing the interaction between the components [9]. The second way is for the health monitor component to monitor the operation of the calculation component by observing the inputs and outputs to the calculation component.

Table 1
Choices extracted from the arguments

Content	Choice	Pros	Cons
G0021—operation is monitored and unexpected behaviour handled	G0022—health monitor detects failures based on health information provided to it	J0004—simplicity since health monitor does not need to access and interpret another component's state	AG0004—can a failing/failed component be trusted to interpret error-free data? AG0006—a contract needs to be establish and managed between the components
	G0023—health monitor detects failures based on provided information	J0005—health monitor can be protected against failures in other components easier	AG0005—health monitor is more complex and prone to change due to dependence on the component
Timing—the system meets its timing requirements	Timing(1)—system meets its timing requirements with sufficient reliability	J0001—there is no need for fault tolerance mechanisms to be provided	AG0001—it is hard to produce components of sufficiently high integrity and hard to justify this has been achieved
	Timing(2)—mechanisms in place to tolerate key errors in timing behaviour	J0002—individual components can be produced to a lower integrity J0003—fault tolerance mechanism can be abstract of specific failures	AG0002—fault tolerance mechanism has to be demonstrated to protect the component sufficiently well against anticipated failures

3.2.2. Consideration of choices derived from the dependability and timing arguments

The next stage (stage 3(a)) in the approach is the elicitation and evaluation of choices. This stage extracts the choices, and considers their relative pros and cons. In general, where there is a choice there should be both assumptions or away goals and justifications for each of the options. In many cases these form the basis for the pros and cons for each of the options. The results are presented in Table 1.

From Table 1 it can be seen that some of the choices that need to be made about individual components are affected by choices made by other components within the system. For instance, goal G0022 is a design option of producing the health monitoring component relying on the other components to provide health data in-order that it is less complex. However, this results in the other components being more complex and hence it is harder to justify they satisfy the Dependability goal in Fig. 4.

Table 2
Evaluation based on timing argument

Item No	Arg ids	Question	Importance	Response	Design advice
1	G0020	Is sufficient information about the bounds on timing behaviour known?	Essential	More information is needed	
2	G0015	Are the timing requirements specified appropriately?	Essential	More information is needed	Use a sufficiently expressive and well-defined notation for representing timing requirements
3	G0016	Is the system implemented in a predictable manner?	Essential	Static scheduling provides a deter ministic scheduling model	
4	G0016	Does the implementation approach support graceful degradation?	Value added	There is no graceful degradation available since the scheduling model assumes tasks execute in their given slot.	A scheduling approach such as fixed priority scheduling is considered to degrade gracefully in the presence of timing overruns [10].
5	G0016	Does the implementation approach support scalability?	Value added	There is no scalability available since the scheduling model assumes tasks execute in their given slot. As such the only way to handle change is to re-allocate slots and adjust their size.	A scheduling approach such as fixed priority scheduling is considered flexible in a manner that supports scalability well.
6	G0017	Are verification techniques available for the scheduling model available?	Essential	Yes (for details see Ref. [11])	
7	G0017	Are execution time analysis approaches available for the underlying platform? (i.e. processor)	Essential	More information is needed	Use a processor for which a valid model exists such that execution time analysis can be derived. This can be difficult especially with modern processors [12].
8	G0021	Is unexpected behaviour captured?	Essential	More information is needed	Requirements should be placed on the health monitoring component in-order to detect errors in the timing behaviour. The implementation of these requirements should be tolerant to the failures being detected, e.g. be able to respond to the software becoming live locked.
9	G0022	Can the health monitor detect failures using health information provided to it?	Essential	More information is needed	
10	G0023	Does the health monitor ensure the integrity of the information provided to it?	Essential	More information is needed	The health monitor needs to obtain data from another source that allows it to validate the information received. This requires the establishing and managing of contracts. For timing, a watchdog mechanism in either hardware or software can be used.
9	G0022	Can the health monitor detect failures using health information provided to it?	Essential	More information is needed	
10	G0023	Does the health monitor ensure the integrity of the information provided to it?	Essential	More information is needed	The health monitor needs to obtain data from another source that allows it to validate the information received. For timing, a watchdog mechanism in either hardware or software can be used.

3.2.3. Assessment criteria derived from the timing argument

Stage 3(b) then extracts assessment criteria from the argument that can then be used to evaluate whether particular solutions (stage 3(c)) meets the claims from the arguments generated earlier in the process. The assessment criteria have been derived mainly from the leaf goals in the tree. The reason is an assessment criteria derived from a higher-level goal would be a more general version of assessment criteria derived from the lower-level goals that support it.

Table 2 presents some of the results of extracting assessment criteria from the arguments for claim Timing(2) from Fig. 5. The table includes an evaluation of a solution based on a proportional integration differentiation (PID) loop with static scheduling as discussed in Section 3. Table 2 shows how assessment criteria have different importance associated (e.g. *Essential* versus *Value Added*). These relate to properties that must be upheld or those whose handling in a different manner may add benefit (e.g. reduced susceptibility to change).

The responses in the table are only partially for the solution considered due to the lack of other design information. Where information is provided it is mostly based on a qualitative assessment of the design. As the design evolves the level of detail contained in the table would increase and the table could then be populated with evidence gathered quantitatively from verification activities, e.g. timing analysis, where appropriate. The tables give a number of recommendations on how the design should evolve—note only recommendations are given at this stage since the satisfaction of other objectives may make these recommendations prohibitive.

The principal recommendations are that

- timing requirements are specified using an appropriate notation
- part of the health monitoring functionality should be to detect unanticipated timing behaviour
- use a scheduling approach that has a valid model associated with it and has additional properties such as better support for scalability and graceful degradation in the presence of timing errors. An example of this form of scheduling approach is fixed priority scheduling [10]
- contracts are established between the satisfaction arguments associated with health monitoring functionality and those concerning expected application behaviour.

3.3. Change argument

Fig. 7 presents the top-level argument for *Upgrade can be performed with a minimum of effort*—goal G11. The key requirements for change to be managed appropriately within the design should be taken from knowledge from previous projects or experience of what the likely changes are (assumption A11) and which of those changes can be considered killer changes (assumption A12). Killer changes are considered to be those that are both anticipated

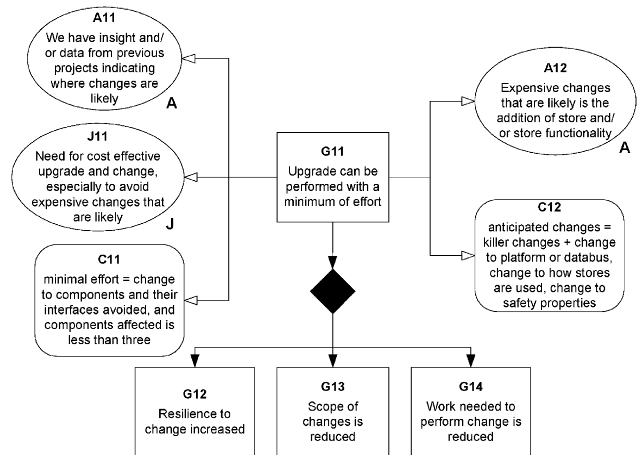

Fig. 7. Top level argument for change management.

and expensive to carry out. The goal G11 is satisfied by a choice of one or more of the following: the resilience to change is increased (goal G12), the scope of changes is reduced (goal G13) and the ease of change is increased (goal G14). The goal G11 is justified (justification J11) by the need for cost effective upgrade and changes to systems. The following subsections expand goals G12, G13 and G14.

3.3.1. Goal G12—increasing resilience to change

Fig. 8 contains the argument that expands on the previously stated goal, G12, that resilience to change is increased. This goal is satisfied by a strategy that splits it into functional and non-functionality properties. The functional aspects leads to a choice between building flexibility into the design such that foreseeable changes are handled and having generic functions instantiated at run-time. The non-functional aspects of the system is further split into timing, memory and availability that are all

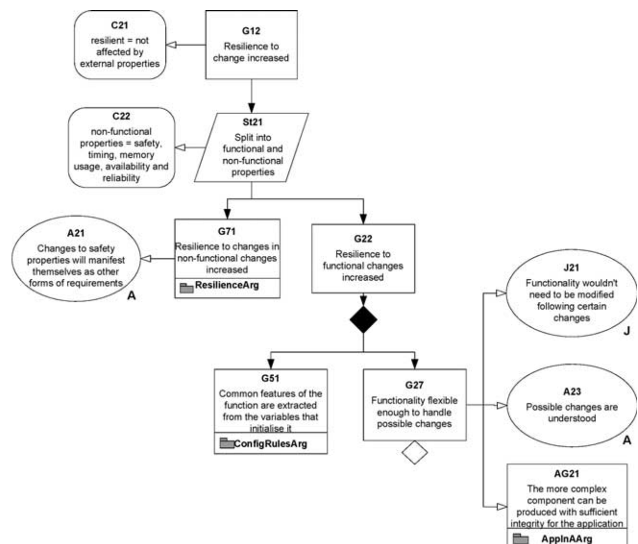

Fig. 8. Building resilience to change into the overall system's design.

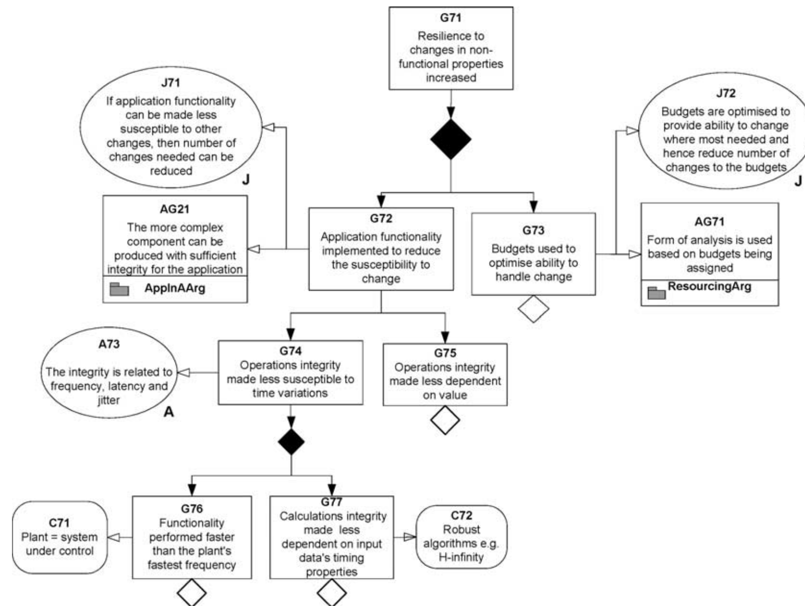

Fig. 9. Generic argument for increasing resilience to change.

satisfied by an away goal G71—an away goal is used so that the same argument strategy can be applied to each of the properties. The arguments for goals G71 and G51 are given in Figs. 9 and 10, respectively.

Goal G71 has two choices of how resilience can be increased; to make the functionality robust to changes, and to use a reservation-based form of analysis [13] and assign budgets to maximise scalability. Following of either of the approaches means the interaction between the components needs to be controlled through the use of contracts. The contracts would establish guarantees for the dependence of the functional properties of components on the non-functional properties of the systems (sometimes referred to as emergent properties). This would mean when any part of the system changed leading to a change in the non-functional properties, e.g. timing, then the impact on the functional properties can be checked.

3.3.2. Goal G13—reducing the scope of changes

Fig. 11 contains the argument that expands on the previously stated goal, G12, that scope of change is reduced. This goal is satisfied by a strategy that splits it into

- functional, which leads to a choice between
  - generic functions initialised at run-time using separately provided initialisation details (refer to the argument in Fig. 10 for further details), or
  - standard interfaces being established (refer to the argument in Fig. 12 for further details) or
  - separating out the parts of the system that are likely to change in a modular fashion such that the change can be reasoned about in a contained area (refer to the argument in Fig. 13 for further details)
- and non-functionality properties. The non-functional aspects are left undeveloped because the satisfaction of these goals places requirements on how the infrastructure of the system is implemented which is out of scope at this stage of architectural design.

3.3.3. Goal G14—easing the task of changing the system

Fig. 14 contains the argument that also expands on the previously stated goal, G14, that the work needed to perform change is reduced. This goal is satisfied by a strategy that splits it into functional and non-functionality properties. One of the following strategies again satisfies the functional aspects. It should be noted that the strategies are the same as the ones used in Section 3.3.2 that were

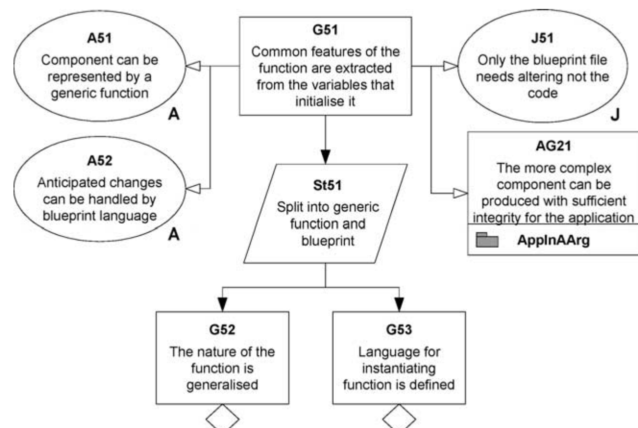

Fig. 10. Generic argument for the use of run-time initialisation.

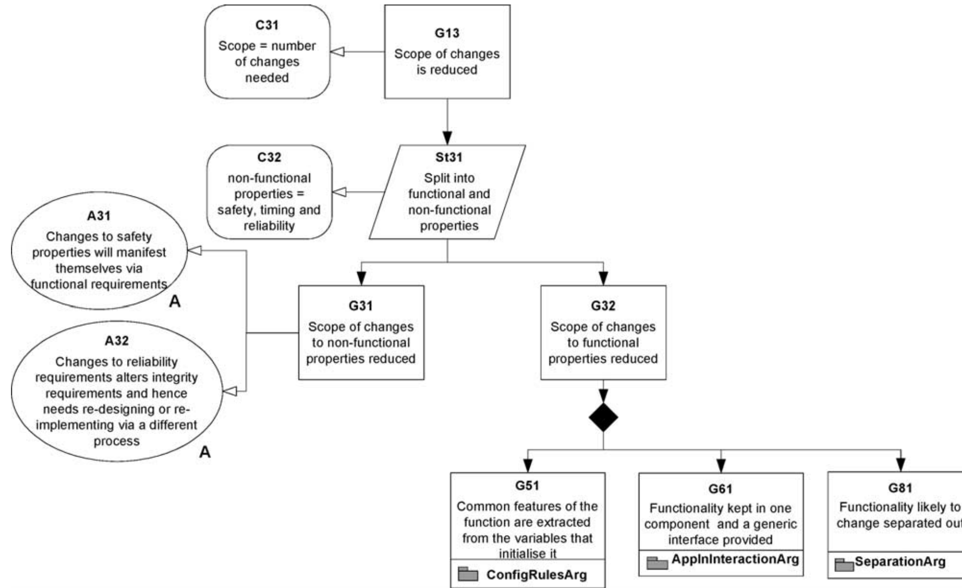

Fig. 11. Argument for reducing scope of changes.

developed for reducing the impact of change. To reuse the strategies is relevant because having a reduced scope often eases the task of performing change

- having generic functions that are initialised at run-time using separate initialisation details (refer to Fig. 10), or
- specific functions and generic interfaces (refer to Fig. 12), or
- separating out the functionality that is likely to change (refer to Fig. 13).

The non-functional aspects are left undeveloped because the satisfaction of these goals places requirements on how the infrastructure of the system is implemented which in the case of this work is out of scope.

As discussed earlier, where there is an interaction to be managed there will be a need to establish and manage contracts. The argument presented in Fig. 14 has a number of interactions. These include between

- the initialisation and generic parts of a component there will need to be a contract that ensures the initialisation part contains all the necessary information needed by the generic part
- the main part of a component and the parts that have been separated out there will need to be a contract that ensures the initialisation part contains all the necessary information needed by the generic part
- two components that communicate across a standard interface
- the non-functional and functional aspects of components—refer to Section 3.3.2 for further details.

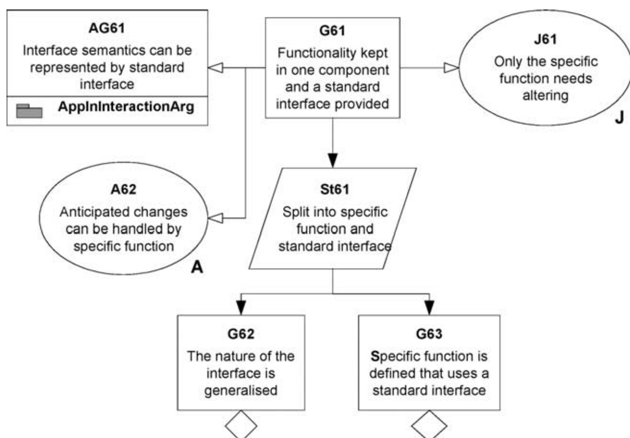

Fig. 12. Generic argument for the use of standard interfaces.

3.3.4. Consideration of choices derived from the change arguments

The choices presented in Table 3 emerge from considering the change arguments presented in Sections 3.3.1–3.3.3.

Table 3 indicates the relative pros and cons of the two choices that can be made. The actual best choice in

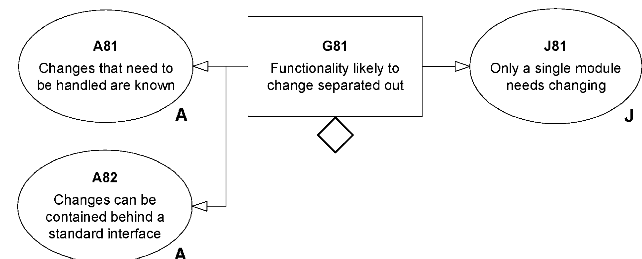

Fig. 13. Argument for the use of separation.

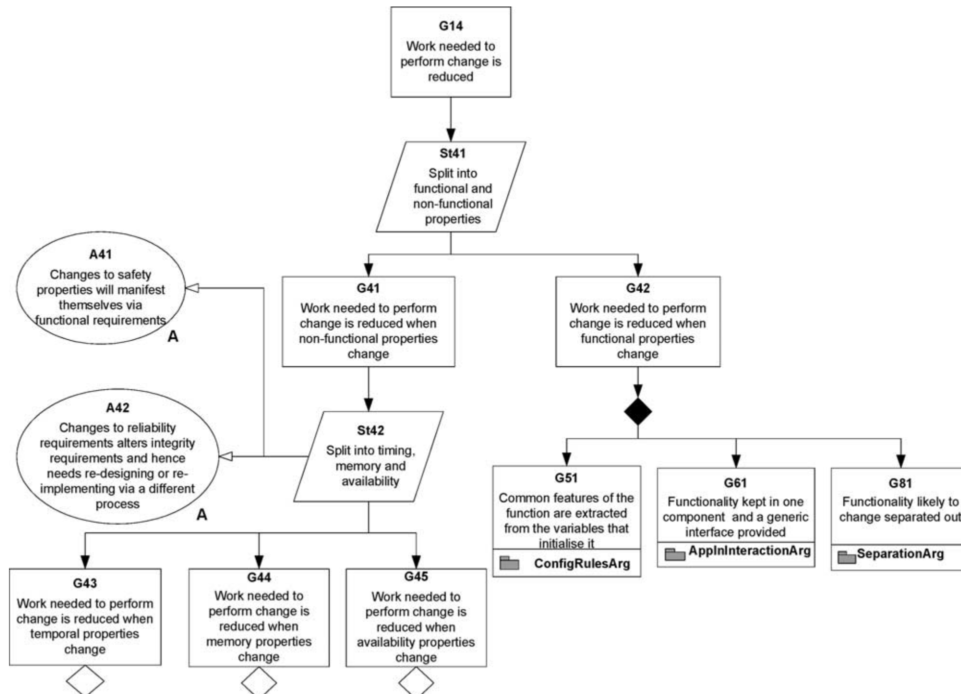

Fig. 14. Argument for ease of change.

a particular situation will depend on the nature of the component's functionality and the types of change that need to be carried out. For instance there are only certain classes of function that can be made generic or robust to change without considerable cost and effort; however, generic interfaces, reservation-based techniques and separating out functionality are more universally applicable. Also, functions that are generic, robust or separated out can only easily be applied to the component itself, whereas standard interfaces and reservation-based techniques can be applied to groups of components.

It should be noted that

- a combination of the options can be employed where needed
- some options will be affected by other objectives, e.g. supporting managed change at the expense of a more complicated design might affect the ability to certify the final product
- currently the options are chosen through the assessment criteria extracted from the arguments. In future it is envisage that multi-criteria optimisation may be used.

3.3.5. Assessment criteria derived from the change arguments

From the arguments presented in Sections 3.3.1–3.3.3, assessment criteria that can be applied to potential solutions have been derived as shown in Table 4. Table 4 indicates that the importance of all assessment criteria is value added rather than essential. The reason is managing change better is not essential. However, it does affect cost and when/if changes can be carried out during the system's operational life.

The recommendations in the table include:

- use a PID loop but separate out the gains used from the main functionality of the component and try to choose these gains such that the behaviour of the system is resilience to other changes and errors. A contract would be required between the functionality of the PID loop and the rest of the system that managed the interaction between the PID loop and the non-functional properties of the system
- schedule the calculation component such that other changes in the temporal behaviour have less effect on it. To do this in practice, a contract would have to be established that controls the interaction between the scheduler and the calculation component. This contract would specify properties such as latency and jitter.
- separate the functionality of the health monitoring component from that of the calculations wherever possible.

With the principles that we have established for organising the safety case structure 'in-the-large', and the complementary approach we have described for reasoning about the required properties of the system architecture, we believe it is possible to create a flexible, modular, certification argument for IMA. This is discussed in Section 4.

4. Establishing a modular safety case

A conventional safety case can be considered as consisting of the following four elements

Table 3
Consideration of choices

Goal	Choice	Pros	Cons
G22	G26—initialisation and generic function	J51—only the initialisation file needs altering not the code	1. A51—component can be represented by a generic function 2. A52—anticipated changes can be handled by the initialisation language. This means not only the types of changes but also their nature needs to be known.
	G27—use flexible functions capable of handling anticipated changes	J21—functionality would not need to be modified following certain changes	1. A23—possible changes have to be known at design time 2. AG21—the design will become more complicated leading to other difficulties (e.g. certification) which may negate any benefit.
G32 and G42	G53 and G46—initialisation and generic function	J51—only the initialisation file needs altering not the code	1. A51—component can be represented by a generic function 2. A52—anticipated changes can be handled by the initialisation language. This means not only the types of changes but also their nature needs to be known. 3. AG21—the design will become more complicated leading to other difficulties (e.g. certification) which may negate any benefit
	G61 and G47—use a standard interface	J61—only the specific function needs altering	1. AG61—interface semantics can be represented by standard interface 2. A62—anticipated changes can be handled by specific function. This means not only the types of changes but also their nature needs to be known.
	G35 and G48—separate out functionality	J81—only a single module needs to be updated	1. A81—the types of changes must be known 2. A82—changes can be contained behind a standard interface
G71	G72—make functionality robust to change	J71—if application functionality can be made less susceptible to other changes, then number of changes needed can be reduced	AG21—resilient functionality can be more complex to produce and reason about
	G73—use reservation-based techniques and optimise budgets	J72—budgets are optimised to provide ability to change where most needed and hence reduce number of changes to the budgets	AG71—a form of analysis is used that is based on budgets being assigned

- *Objectives*: the safety requirements that must be addressed to assure safety
- *Argument*: showing how the evidence indicates compliance with the requirements
- *Evidence*: information from study, analysis and test of the system in question
- *Context*: identifying the basis of the argument presented

Defining a safety case 'module' involves defining the objectives, evidence, argument and context associated with one *aspect* of the safety case. Assuming a top–down progression of objectives-argument-evidence, safety cases can be partitioned into modules both horizontally and vertically.

- *Vertical (hierarchical) partitioning*: the claims of one safety argument can be thought of as objectives for another. For example, the claims regarding software safety made within a system safety case can serve as the objectives of the software safety case.
- *Horizontal partitioning*: one argument can provide the assumed context of another. For example, the argument that 'all system hazards have been identified' can be the assumed context of an argument that 'all identified system hazards have been sufficiently mitigated'.

In defining a safety case module it is essential to identify the ways in which the safety case module depends upon the arguments, evidence or assumed context of other modules. A safety case module, should therefore be defined by the following interface

1. objectives publicly addressed by the module
2. evidence presented within the module
3. context defined within the module
4. arguments requiring support from other modules.

Table 4
Evaluation based on change argument

Item no	Arg ids	Question	Importance	Response	Design advice
1	G27	Can the functionality be made flexible enough to handle likely changes?	Value added	Whilst the nature of the calculations may change, the fundamental data input and output from the system should be relatively stable	Produce the health monitoring such that it is largely dependent on the input/output values (e.g. checking values are within a valid range) rather than the specifics of the calculation
2	G72	Can the resilience of the component's functionality to changes in timing behaviour be increased?	Value added	If the calculations are performed using a proportional integration differentiation (PID) loop, then time variations will cause errors and frequent changes to its gain parameters. Using a robust control technique (e.g. H-infinity) will significantly reduce the susceptibility to time variations	The advantage of PID loops is that it is a tried and tested app roach. Therefore, use a PID loop but separate out the gain factors such that change becomes more manageable
3	G72	Can the resilience of the component's functionality to changes in memory usage be increased?	Value added	The interaction in question is unaffected by memory usage at this level of design	
4	G72	Can the resilience of the component's functionality to changes in availability be increased?	Value added	The interaction in question is unaffected by availability at this level of design	
5	G73	Can budgets given to individual compo nents be assigned such that changes within the systems can be handled better?	Value added	The calculation component could be scheduled and designed with timing characteristics that assume a slower response than initially expected and features put in place to ensure faster than expected performance does not cause a problem	Assign the calculation compo nent a larger budget and assign it a place in the schedule that is not affected by release jitter. With fixed priority scheduling this can be achieved using off sets and priorities. With static scheduling this can be achieved using an appropriate slot position
6	G31	Can the scope of changes caused by temporal properties be reduced?	Value added	Refer to item 2	Refer to item 2
7	G31	Can the scope of changes caused by memory usage properties reduced?	Value added	Refer to item 3	Refer to item 3
8	G31	Can the scope of changes caused by availability properties reduced?	Value added	Refer to item 4	Refer to item 4
9	G51	Can the nature of the function be made abstract of the likely/costly changes and a standard language is defined for instantiating the function?	Value added	As stated in item 1, the health monitoring component's functionality can be based on just the inputs and outputs of the calculation component. This level of integration requires appropriate contracts to be established and managed	Refer to item 1
10	G61	Can an interface be produced that isolates the likely/costly changes on either side of an interaction?	Value added	Refer to item 9	Refer to item 9
11	G43	Can the work that needs to be per formed when temporal properties change is reduced?	Value added	Refer to item 2	Refer to item 2
12	G44	Can the work that needs to be performed when memory properties change be reduced?	Value added	Refer to item 3	Refer to item 3
13	G45	Can the work that needs to be performed when availability properties change is reduced?	Value added	Refer to item 4	Refer to item 4
14	A21, A31, A41	Can the component's functionality be made to robust to changes in the system's safety properties?	Value added	Refer to item 1	Refer to item 1
15	A22, A32, A42	Can the component's functionality be made to robust to changes in the system's reliability properties?	Value added	Again, robust algorithms could be used if their integrity can be justified. Alternatively, the PID loop gains could be optimised to withstand some errors	Optimise PID loop gains such that some errors

(*continued on next page*)

Table 4 (*continued*)

Item no	Arg ids	Question	Importance	Response	Design advice
16	AG21	Does any added complexity cause problems for other activities such as certification?	Value added	No added complexity has been proposed at this level of design refinement	
17	G22	Can the resilience of the compo nent's functionality to other func tional changes be increased?	Value added	Refer to item 1	Refer to item 1
18	G81	Can functionality where changes are likely to happen be separated out to reduce the cost of performing changes?	Value added	Refer to item 2	Refer to item 2

Inter-module dependencies

5. reliance on objectives addressed elsewhere
6. reliance on evidence presented elsewhere
7. reliance on context defined elsewhere.

Importantly, the interface abstracts from the internal detail of the safety argument presented within the module. Item 1 declares the public objectives supported by the module without revealing the supporting argument. Principally, the interface must specify the safety argument claims being addressed by a module. For example, the interface for a safety argument module for a specific piece of functionality must describe clearly the safety claims that are supported for the functionality. It is crucial that the interface also exposes items 2 and 3 of the interface shown above in order to ensure that when modules are composed together they form a consistent whole. Firstly, the module must define the pieces of supporting evidence that are used in support of the module argument. When modules are composed with others, it is necessary to check that this evidence is consistent with that used in the other modules (e.g. that the models and assumptions embedded in a piece of safety analysis are not contradicted by the models and assumptions of another piece of safety analysis). Context defines the bounds and limitations of the argument presented within the module. Context could be defined at any number of points during the safety argument within a module, e.g. to define the assumed operational context, interface with other systems, assumed duration and frequency of operation or acceptability/tolerability criteria. All such context must be exposed at the boundary interface of the module in order to ensure that in composition of modules the collective context is consistent. Item 3 defines the *environment* in which the argument presented within the module holds.

The argument within a safety case may not fully address all of the claims of the module. There may remain claims requiring support from other modules (as yet unknown). For example, part of a software safety argument may rest upon a claim regarding the reliability of input sensors. As this is a hardware systems issue it should fall to another module to support such a claim. Under item 4 of the interface proposed above, all argument claims remaining to be supported are required to be defined.

Items 5–7 of the interface definition recognise that there will be times within a specific safety case module when the argument may depend upon the claims, evidence or context defined within another (known) module. For example, within software safety argument the justification of developing the software to a specific level of integrity may depend upon the existence (claim) of an effective hardware interlock—addressed as part of the hardware argument presented in another module. Where such explicit cross-references exist they should be highlighted within the definition of the safety case module interface. This is strongly analogous to declaring the 'use' relationships that exist between software modules where one module is known to require the services of another module.

The principal need for having such well-defined interfaces for each safety case module arises from being able to ensure that modules are being used consistently and correctly in their target application context (i.e. when composed with other modules). This topic is addressed in the following section.

4.1. Safety case module composition

Safety case modules can be usefully composed if their claims and arguments complement each other—i.e. one or more of the claims 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 claims set by the system argument. At the same time, an important side-condition it that the 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.

The definition of safety case module interfaces and satisfaction of conditions across interfaces upon composition is analogous to the long established rely-guarantee

approach to specifying the behaviour of software modules. Jones in Ref. [14] talks of 'rely' conditions that express the assumptions that can be made about the interrelations (interference) between operations and 'guarantee' conditions that constrain the end-effect assuming that the 'rely' conditions are satisfied. For a safety case module, the rely conditions can be thought of items 4–7 of the interface introduced in Section 4 whilst item 1 (claims addressed) define the guarantee conditions. (Items 4–7 define the context in which the argument of the module is presented.) Items 2 (evidence presented) and 3 (context defined) must continue to hold (i.e. not be contradicted by inconsistent evidence or context) during composition of modules.

Table 5 defines the steps that must be undertaken when attempting to usefully compose two safety case modules A and B with interfaces defined in accordance with the format introduced in Section 4. The steps in Table 5 assume the argument in module A provides the target context for B (i.e. module A is expected to be at a higher level in the overall safety case argument and therefore sets claims to be satisfied by subsidiary modules such as B).

It should be recognised that *partial* claim matching is acceptable within Step 1—i.e. where a module is partially supported by another. A number of modules may need to be composed together in order to fully support outstanding claims. Where a number of modules are to be composed together pairs of modules can successively composed together. For example, to compose modules A–C together, a pair of modules is first composed together then the resulting composition is composed with the remaining module. Order of composition is not important as composition is commutative.

Table 5
Steps involved in safety case module composition

Step 1—claim matching	a. Assess whether any of the claims requiring support in Module A (i.e. those listed under item 4 of the declared interface for Module A) match the claims addressed by Module B (i.e. those listed under item 1 of the interface for Module B). b. Conversely, assess whether any of the claims requiring support in Module B (i.e. those listed under item 4 of the declared interface for Module B) match the claims addressed by Module A (i.e. those listed under item 1 of the interface for Module A).
Step 2—consistency checks	If matched claims are found as a result of Step 1, assess whether the context and evidence 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 evidence 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	a. Where cross-references are made by Module A to Module B (i.e. references listed under items 5–7 of the declared interface for Module A) check that the entities referenced do indeed exist within Module B. b. Conversely, where cross-references are made by Module B to A (i.e. references listed under items 5–7 of the declared interface for Module B) check that the entities referenced do indeed exist within Module A.

Consistency between modules is a symmetric relation (i.e. Module A is consistent with B implies Module B is consistent with A). Step 2 (Consistency Checks) is easily stated but in reality hard to satisfy given the varied nature of the evidence and context defined within any safety case argument. For example, within Step 2 lies the challenge of determining whether the safety analysis presented within one module (e.g. component level failure modes and effects analyses) is consistent with that presented in another (e.g. a fault tree analysis). Although these two different techniques have distinct roles as pieces of evidence they also potentially overlap in their model of the system behaviour, e.g. a component level failure mode within the FMEA may also appear as a basic event at the bottom of the fault tree. Where such overlap occurs lies the problem of potential inconsistency. Identifying and managing consistency between safety analysis evidence is sufficiently challenging that it has warranted discussion as a problem in its own right, see Wilson et al. in Ref. [15].

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 by UK Defence Standards 00-56 [16] and 00-55 [17]) 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.

4.2. The challenge of compositionality

It is widely recognised (e.g. by Perrow [18] and Leveson [19]) that relatively low risks are posed by independent component failures in safety-critical systems. However, it is not expected that in a safety case architecture where modules are defined to correspond with a modular system structure that a complete, comprehensive and defensible argument can be achieved by merely composing the arguments of safety for individual system modules. Safety

is a whole system, rather than a ‘sum of parts’, property. Combination of effects and emergent behaviour must be additionally addressed within the overall safety case architecture (i.e. within their own modules of the safety case). Modularity in reasoning should not be confused with modularity (and assumed independence) in system behaviour.

4.3. 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 Eiffel contracts [9] and contracts in object-oriented reuse [20].

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. A proposed format for contracts between composed safety case modules that covers each of these aspects is illustrated in Fig. 15.

4.4. Safety case architecture

We define safety case architecture 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 system 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 and 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

<table>
<tr><th colspan="12">Safety Case Module Contract</th></tr>
<tr><th colspan="12">Participant Modules</th></tr>
<tr><td colspan="12">(e.g. Module A, Module B and Module C)</td></tr>
<tr><th colspan="12">Claims Matched Between Participant Modules</th></tr>
<tr><td colspan="3">Claim</td><td colspan="3">Required by</td><td colspan="3">Addressed by</td><td colspan="3">Claiml</td></tr>
<tr><td colspan="3">(e.g. Claim G1)</td><td colspan="3">(e.g. Module A)</td><td colspan="3">(e.g. Module B)</td><td colspan="3">(e.g. Claim G2)</td></tr>
<tr><td colspan="3">…</td><td colspan="3">…</td><td colspan="3">…</td><td colspan="3">…</td></tr>
<tr><th colspan="12">Collective Context and Evidence of Participant Modules held to be consistent</th></tr>
<tr><td colspan="6">Context</td><td colspan="6">Evidence</td></tr>
<tr><td colspan="6">(e.g. Context C9, Assumption A2, Model M4)</td><td colspan="6">(e.g. Evidence Sn3, Sn8)</td></tr>
<tr><th colspan="12">Resolved Inter-Argument Module References</th></tr>
<tr><td colspan="4">Cross Referenced Item</td><td colspan="4">Source Module</td><td colspan="4">Sink Module</td></tr>
<tr><td colspan="4">(e.g. Away Goal AG3)</td><td colspan="4">(e.g. Module B)</td><td colspan="4">(e.g. Module C)</td></tr>
</table>

Fig. 15. Format of contract between safety case 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 IMA-based systems 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 are 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 Sections 4.2 and 4.3, 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.

To help consider the maintainability of the certification case for a given system architecture the corresponding safety case architecture must also be considered at an early stage in the system development lifecycle. In Section 4.5 we describe how change scenarios may be considered against a proposed safety case architecture in order to assess long term maintainability.

4.5. Managing modular safety case change

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).

The principal strength of modular safety cases will be observed when assessing the impact of possible changes through being able to state with confidence that the effects of change do not propagate outside of a safety case module boundary providing that the interface is preserved. The interfaces defined for modular safety cases will also help deciding upon a recovery action to fix a 'broken' argument by showing clearly the 'non-negotiable' safety properties that must be upheld through any change suggested to the safety argument in order to re-establish the argument.

In extremis for an IMA 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 4.1 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 not 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 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 V then it may be used in wider number of situations (e.g. for both 110–120 V main supply and 230–240 V mains supply). If the adaptor is qualified to a narrower operating range then obviously its scope of applicability is more restricted.

Claims to be supported within modules should state limits rather than objectives. Borrowing terminology from the ALARP (as low as reasonably practicable) framework [21], '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 claims within a module (i.e. claims 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 to 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 claims should ideally communicate both of limit and objective aspects of any requirement (by means of defining clearly the acceptance context of any undeveloped claim).

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.

5. Evaluation of safety case architecture

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—software architecture analysis method (SAAM) has been developed by Kazman et al. [22].

With little modification, the SAAM 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 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 partitioned and carefully cross referenced in accordance with the principles stated in this report 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 Fig. 16 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. 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.

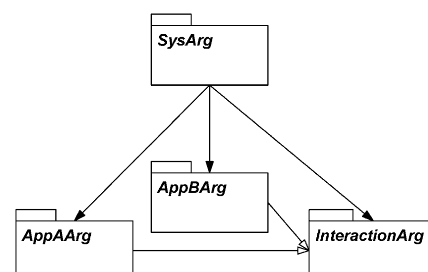

Fig. 16. A simple safety case architecture.

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. Fig. 17 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 rexamined 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. Fig. 18 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).

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. Fig. 19 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.

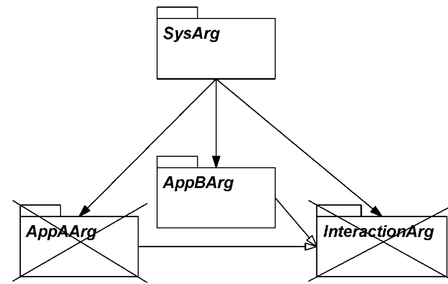

Fig. 18. Illustration of non-local change.

5.1. Reasoning about interactions 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 [23], 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 railways safety standard CENELEC ENV 50129 [15] makes an interesting distinction between those

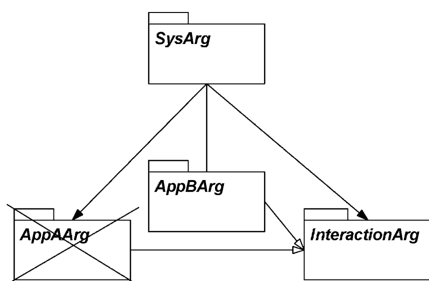

Fig. 17. Illustration of local change.

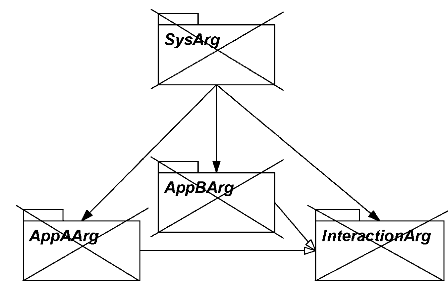

Fig. 19. Illustration of architectural change.

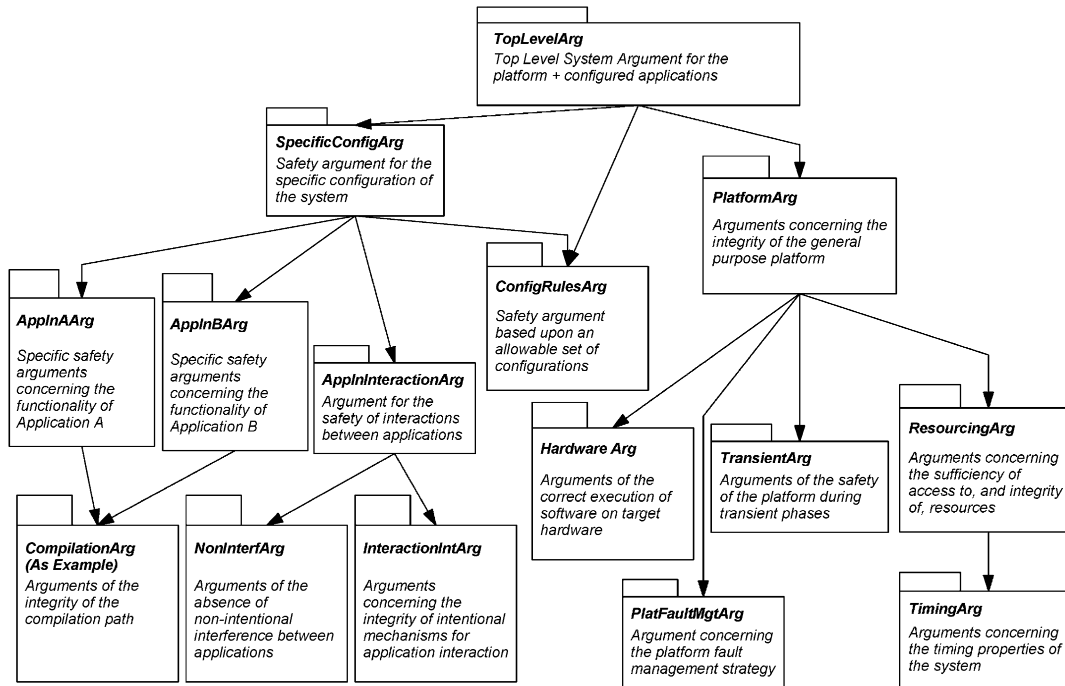

Fig. 20. Safety case architecture of modularised IMA safety argument.

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 ENV 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.

6. Example safety case architecture for a modular system

In Section 3 we illustrated how the elaboration of satisfaction arguments for key architectural qualities could be used in the process of architectural trade-off analysis for a simple control system. In particular, we described the possible structure of the timing argument and how this argument depended upon other aspects of the safety case. This timing argument (when developed) will form one of the 'modules' of the overall safety case required for this system. The overall architecture of this safety case is shown in Fig. 20 (the UML package notation is used to represent safety case modules). The module of the safety case concerned with timing is shown in the bottom right hand corner of Fig. 20.

The role of each of the modules of the safety case architecture shown in Fig. 20 is as follows

- *ApplnAArg*: 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 an abstract argument established once-for-all leading to support from specific modules for particular hardware choices
- *ResourcingArg*: overall argument concerning the sufficiency of access to, and integrity of, resources (including time, memory, and communications)
- *TimingArg*: overall argument concerning the system architectures ability to predict whether the timing requirements are met with sufficient reliability
- *ApplnInteractionArg*: argument addressing the interactions between applications, split into two legs: one concerning intentional interactions, the second concerning unintentional 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 unintentional interactions (e.g. corruption of shared memory) between

applications. Supporting module for ApplnteractionArg. 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).

It should be noted that Fig. 20 only depicts the satisfaction relationships that are expected between the various argument modules (e.g. showing how the Timing Argument—TimingArg—supports the Resourcing Argument—ResourcingArg) and does not (for clarity) show contextual dependencies.

The safety case architecture promotes the ideal (e.g. in the NonInterfArg module) 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.

6.1. Illustration of an example contract

In the argument concerned with the toleration of errors in timing behaviour (shown in Fig. 6 in Section 3.2.1), assumed to be contained within the TimingArg module as discussed Section 6, dependencies on the Platform Fault Management Argument (PlatFaultMgtArg), Application Argument (ApplnAArg) and Application Interaction

Safety Case Module Contract		
Participant Modules		
TimingArg, ApplnAArg, ApplnIntArg, PlatFaultMgtArg		
…		
Resolved Inter-Argument Module References		
Cross Referenced Item	*Source Module*	*Sink Module*
AG0003	TimingArg	PlatFaultMgtArg
AG0004	TimingArg	ApplnAArg
AG0005	TimingArg	PlatFaultMgtArg
AG0006	TimingArg	ApplnInteractionArg

Fig. 21. Extract from contract established between control system arguments.

Argument (ApplnInteractionArg) were highlighted. Where such dependencies exist between composed argument modules, contracts (as described in Section 4.3) should be established. Using the format described for documenting contracts in Section 4.3, Fig. 21 shows the relevant contract terms associated with the contextual dependencies of the timing argument on the other arguments.

As described in Section 4.5, the benefit of establishing such a contract between the participant modules is that it becomes possible to make changes to the internal detail of individual argument modules without affecting the others providing that the contract terms continue to be upheld.

7. Conclusions

In order to reap the potential benefits of modular construction of safety critical and safety related systems a modular approach to safety case construction and acceptance is also required.

This paper has presented a method to support architectural design and implementation strategy trade-off analysis, one of the key parts of component-based development. Specifically, the method presented provides guidance when decomposing systems so that the system's objectives are met and deciding what functionality the components should fulfil in-order to achieve the remaining objectives.

One of the criteria to be considered during architectural trade-off analysis is the maintainability of the associated safety case structure. This paper has indicated how safety case architecture can be represented and considered as part of the system design process.

References

[1] Dobrica L, Niemela E. A survey of software architecture analysis methods. IEEE Transact Software Engng 2002;28(7): 638–53.

[2] Douglass B. Real-time UML. Reading, MA: Addison-Wesley; 1998.

[3] Kelly TP. Arguing safety—a systematic approach to safety case management. UK: Department of Computer Science, University of York; 1998.

[4] Laprie J-C. Dependable computing and fault tolerance: concepts and terminology, 15th International Symposium on Fault Tolerant Computing (FTCS-15); 1985

[5] Kazman R, Klein M, Clements P. Evaluating software architectures—methods and case studies. Reading, MA: Addison-Wesley; 2001.

[6] Basili VR, Rombach HD. The TAME project: towards improvement-oriented software environments. IEEE Transact Software Engng 1988;14(6):758–73.

[7] Kogure M, Akao Y. Quality function deployment and CWQC in Japan. Qual Progr 1983;25–9.

[8] Kelly TP. A six-step method for the development of goal structures. Flixborough, UK: York Software Engineering; 1997.

[9] Meyer B. Applying design by contract. IEEE Comput 1992;25(10): 40–52.

[10] Locke CD. Software architecture for hard real-time applications: cyclic executive vs. fixed priority executives. J Real-Time Syst 1992; 4:37–53.

[11] Bate I. Scheduling and timing analysis for safety-critical systems. UK: Department of Computer Science, University of York; 1998.

[12] Bate I, et al. Use of modern processors in safety critical applications. Comput J 2001;44(6).

[13] Audsley N, Grigg A. Reservation-based timing analysis—a practical engineering approach for distributed real-time systems reservation-based timing analysis, Proceedings of IEEE Conference on Engineering Computer-Based Systems; 2001

[14] Jones C. Specification and design (parallel) programs. IFIP information processing, 83. Amsterdam: Elsevier; 1983.

[15] Wilson S, McDermid JA. Integrated analysis of complex safety critical systems. Comput J 1995;38(10):765–76.

[16] MoD, 00-56 Safety management requirements for defence systems. Ministry of Defence; 1996.

[17] MoD, 00-55 Requirements of safety related software in defence equipment. Ministry of Defence; 1997.

[18] Perrow C. Normal accidents: living with high-risk technologies. Basic books; 1984.

[19] Leveson NG. Safeware: system safety and computers. Reading, MA: Addison-Wesley; 1995.

[20] Helm R, Holland IM, Gangopadhyay D. Contracts: specifying behavioural compositions in object-oriented systems. in OOPSLA/ECOOP'90. ACM SIGPLAN Notices; 1990

[21] HSE, Reducing risk, protecting people. Health and safety executive; 1999.

[22] Kazman R, et al. Scenario-based analysis of software architecture. IEEE Software 1996;13(6):47–55.

[23] RTCA, Software considerations in airborne systems and equipment certification. Washington, DC: RTCA; 1992.