Architectural Considerations in the Certification of Modular Systems

Iain Bate and Tim Kelly

Department of Computer Science
University of York, York, YO10 5DD, UK
{iain.bate,tim.kelly}@cs.york.ac.uk

Abstract. The adoption of Integrated Modular Avionics (IMA) in the aerospace industry offers potential benefits of improved flexibility in function allocation, reduced development costs and improved maintainability. However, it requires 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-off performed during architecture definition reusing information and techniques from the safety argument process.

1 Introduction

Integrated Modular Avionics (IMA) offers potential benefits of improved flexibility in function allocation, reduced development costs and improved maintainability. However, it poses significant problems in certification. The traditional approach to certification relies heavily upon a system being statically defined as a complete entity and the corresponding (bespoke) system safety case being constructed. However, a principal motivation behind 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

S. Anderson et al. (Eds.): SAFECOMP 2002, LNCS 2434, pp. 321-333, 2002.
© Springer-Verlag Berlin Heidelberg 2002

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 such as independence (e.g. including both non-interference and location 'transparency'), increased flexibility in functional integration, and low coupling between components. An additional problem is that these properties are non-orthogonal and trade-offs must be made when defining the architecture.

2 Safety Case Modules

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 addressed by the module
2. Evidence presented within the module
3. Context defined within the module
4. Arguments requiring support from other modules

Inter-module dependencies:

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

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).

2.1 Safety Case Module Composition

Safety case modules can be usefully composed if their objectives and arguments complement each other – i.e. one or more of the objectives supported by a module match one or more of the arguments requiring support in the other. For example, the software safety argument is usefully composed with the system safety argument if the software argument supports one or more of objectives set by the system argument. At the same time, an important side-condition is that the collective evidence and assumed context of one module is consistent with that presented in the other. For example, an 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 [1] 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 as items 4 to 7 (at the start of section 0) of the interface whilst item 1 (objectives addressed) defines the guarantee conditions. 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.

The defined context of one module may also conflict with the evidence presented in another. There may also simply be a problem of consistency between the system models defined within multiple modules. For example, assuming a conventional system safety argument / software safety argument decomposition (as defined by U.K. Defence Standards 00-56 [2] and 00-55 [3]) consistency must be assured 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.

2.2 The Challenge of Compositionality

It is widely recognised (e.g. by Perrow [4] and Leveson [5]) 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.

2.3 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 [6].

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.

2.4 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 & 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.

Isolating Change – arguments that are expected to change (e.g. when making anticipated additions to system functionality) should ideally be located in modules separate from those modules where change to the argument is less likely (e.g. safety arguments concerning operating system integrity).

The principal aim in attempting to adopt a modular safety case architecture for 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.

2.5 Reasoning about Interactions and Independence

One of the main impediments to reasoning separately about individual applications running on an IMA based architecture is the degree to which applications interact or interfere with one another. The European railways safety standard CENELEC ENV

50129 [7] makes an interesting distinction between those interactions between system components that are *intentional* (e.g. component X is meant to communicate with component Y) and those that are *unintentional* (e.g. the impact of electromagnetic interference generated by one component on another). A further observation made in ENV 50129 is that there are a class of interactions that are unintentional 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.

Ideally 'once-for-all' arguments are established by appeal to the properties of the IMA infrastructure to address unintentional interactions. For example, an argument of "non-interference through shared scheduler" could be established by appeal to the priority-based scheduling scheme offered by the scheduler.

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.

The following section describes how properties of the system architecture, such as those discussed above, can be explicitly considered as part of the architecture definition activity.

3 Evaluating Required Qualities during System Architecture Definition

In defining 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 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 questions to be derived whose answers can be used for identifying solutions that do/do not meet the system properties, judging how well the properties are met and indicating where refinements of the design might add benefit.
4. *influence on the design* – identifies constraints on how components should be designed to support the meeting of the system's overall objectives.

A technique (the Architecture Trade-Off Analysis Method – ATAM [8]) 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 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 techniques offers: (a) strong traceability and a rigorous method for deriving the attributes and 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 a particular fixed stages of the process.

3.1 Analysing Different Design Solutions and Performing Trade-Offs

Figure 1 provides a diagrammatic overview of the proposed method. Stage (1) of the trade-off analysis method is producing a model of the system to be assessed. This model should be decomposed to a uniform level of abstraction. Currently our work uses UML [9] for this purpose, however it could be applied to any modelling approach that clearly identifies components and their couplings. Arguments are then produced (stage (2)) for each coupling to a corresponding (but lower so that impact of later choices can be made) abstraction level than the system model. (An overview of Goal Structuring Notation symbols is shown in Figure 2, further details of the notation can be found in [10]) The arguments are derived from the top-level properties 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 [11]). Safety may further involve providing guarantees of independence between functionality. In practice, the arguments should be generic or based on patterns where possible. Stage (3) then uses the information in the argument to derive options and evaluate particular solutions. Part of this activity uses representative scenarios to evaluate the solutions.
Based on the findings of stage (3), the design is modified to fix 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 goal structure. 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.

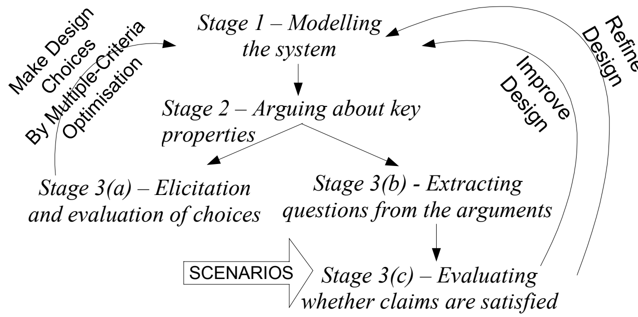

Fig. 1. Overview of the Method

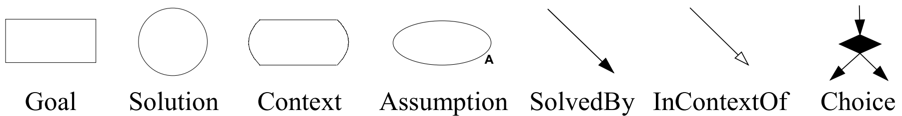

Fig. 2. Goal Structuring Notation (GSN) Symbols

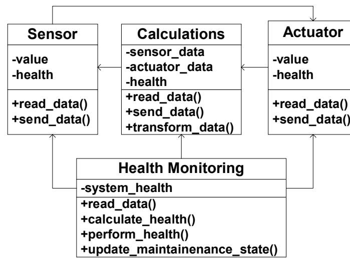

Fig. 3. Class Diagram for the Control Loop

3.2 Example – Simple Control System

The example being considered 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.

At the highest level of abstraction the control loop (the architectural model of which is shown in Figure 3) consists of three elements; a sensor, an actuator and a calculation stage. 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 allow appropriate actions to be taken including the planning of maintenance activities.

In practice as the system development progresses, the component design in Figure 3 would be refined to show more detail. For reasons of space only the *calculation-health monitor* coupling is considered.

Stage 2 is concerned with producing arguments to support the meeting of objectives. The first one considered here is an objective obtained from decomposing an argument for dependability (the argument is not shown here due to space reasons) that the system's components are able to tolerate timing errors (goal **Timing**). From an available argument pattern, the argument in Figure 4 was produced that reasons *"Mechanisms in place to tolerate key errors in timing behaviour"* where the context of the argument is *health monitor* component. Figure 4 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 (a choice is depicted by a black diamond in the arguments) – 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 rely entirely on the results of the internal health monitoring of the *calculation* component to indicate the current state of the calculations. 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.

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 complete. The evidence to be provided to support these goals should be quantitative in nature where possible, e.g. results of timing analysis to show timing requirements are met.

Next an objective obtained from decomposing an argument for maintainability (again not shown here due to space reasons) that the system's components are tolerant to changes is examined. The resultant argument in Figure 5 depicts how it is reasoned the *"Component is robust to changes"* in the context of the *health-monitor* component. There are two separate parts to this; making the integrity of the calculations less dependent on when they are performed, and making the integrity of the calculations less dependent on the values received (i.e. error-tolerant). For the first of these, we could either execute the software faster so that jitter is less of an issue, or we could use a robust algorithm that is less susceptible to the timing properties of the input data (i.e. more tolerant to jitter or the failure of values to arrive).

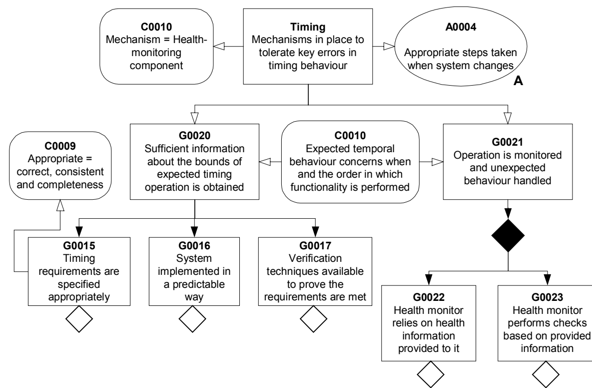

Fig. 4. Timing Argument

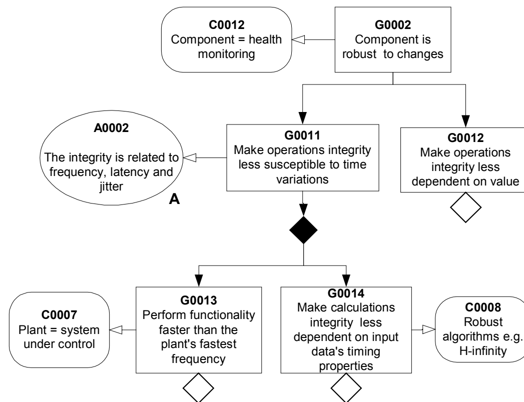

Fig. 5. Minimising Change Argument

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. 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 G0014** is a design option of having a more complicated algorithm that is more resilient changes to and variations in the system's timing properties. However **Goal G0014** is in opposition to **Goal G0023** since it would make the health-monitoring component more complex.

Stage 3(b) then extracts questions 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. Table 2 presents some of the results of extracting questions from the arguments for claim **G0011** and its assumption **A0002** from Figure 5. The table includes an evaluation of a solution based on a PID (Proportional Integration Differentiation) loop.

Table 1. Choices Extracted from the Arguments

Content	Choice	Pros	Cons
Goal G0021 - Operation is monitored and unexpected behaviour handled	**Goal G0022 -** Health monitor relies on health information provided to it	Simplicity since health monitor doesn't need to access and interpret another component's state.	Can a failing/failed component be trusted to interpret error-free data.
	Goal G0023- Health monitor performs checks based on provided information	Omission failures easily detected and integrity of calculations maintained assuming data provided is correct.	Health monitor is more complex and prone to change due to dependence on the component.
Goal G0011 - Make operations integrity less susceptible to time variations	**Goal G0013** – Perform functionality faster than the plant's fastest frequency.	Simple algorithms can be used. These algorithms take less execution time.	Period and deadline constraints are tighter. Effects of failures are more significant.
	Goal G0014 - Make calculations' integrity less dependent on input data's timing properties.	Period and deadline constraints relaxed. Effects of failures may be reduced.	More complicated algorithms have to be used. Algorithms may take more execution time.

Table 2 shows how questions for a particular coupling 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 are only partially for the solution considered due to the lack of other design information. As the design evolves the level of detail contained in the table would increase and the table would then be populated with evidence from verification activities, e.g. timing analysis.

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 the following section.

Table 2. Evaluation Based on Argument

Question	Importance	Response	Design Mod.
Goal G0011 - Can the integrity of the operations be justified?	Essential	More design information needed	Dependent on *response* to questions
Assumption A0002 - Can the dependency between the operation's integrity and the timing properties be relaxed?	Value Added	Only by changing control algorithm used	Results of other trade-off analysis needed

4 Example Safety Case Architecture for a Modular System

The principles of defining system and safety case architecture discussed in this paper are embodied in the safety case architecture shown in Figure 6. (The UML package notation is used to represent safety case modules.)

The role of each of the modules of the safety case architecture shown in Figure 6 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)
- **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 Applnteraction Arg. 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).

An important distinction is drawn above between those arguments that ideally can be established as 'once-for-all' arguments that hold regardless of the specific applications placed on the architecture (and should therefore be unaffected by application change) and those that are configuration dependent.

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

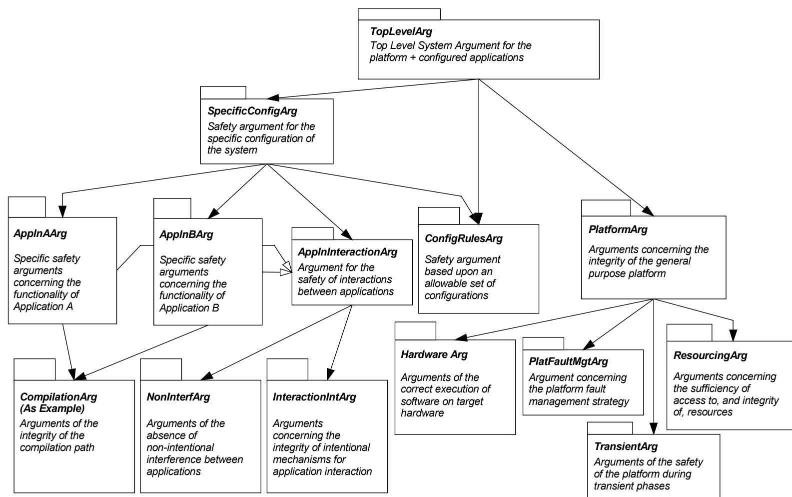

Fig. 6. Safety Case Architecture of Modularised IMA Safety Argument

5 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 addressed 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.

References

1. Jones, C. Specification and design (parallel) programs. in IFIP Information Processing 83. 1983: Elsevier.
2. MoD, 00-56 Safety Management Requirements for Defence Systems. 1996, Ministry of Defence.
3. MoD, 00-55 Requirements of Safety Related Software in Defence Equipment. 1997, Ministry of Defence.
4. Perrow, C., Normal Accidents: living with high-risk technologies. 1984: Basic Books.
5. Leveson, N.G., Safeware: System Safety and Computers. 1995: Addison-Wesley.
6. Meyer, B., Applying Design by Contract. IEEE Computer, 1992. 25(10): p. 40-52.
7. CENELEC, Safety-related electronic systems for signalling, European Committee for Electrotechnical Standardisation: Brussels.
8. Kazman, R., M. Klein, and P. Clements, Evaluating Software Architectures - Methods and Case Studies. 2001: Addison-Wesley.
9. Douglass, B., Real-Time UML. 1998: Addison Wesley.
10. Kelly, T.P., Arguing Safety - A Systematic Approach to Safety Case Management. 1998, Department of Computer Science, University of York.
11. Laprie, J.-C. Dependable Computing and Fault Tolerance: Concepts and Terminology. 1985. 15th International Symposium on Fault Tolerant Computing (FTCS-15).