# Architecture Trade-off Analysis and Codesign for Safety-Related Real-Time Embedded Systems

**N. C. Audsley and I. J. Bate**

***Real-Time Systems Research Group, Department of Computer Science, University of York, York, YO10 5DD, UK***

`{neil,ijb}@cs.york.ac.uk`

***Abstract***

*Large-scale safety-related embedded systems pose unique problems. Unlike most embedded systems, safety-related systems must be developed to meet exacting standards so that they can be verified as fit for use for the intended application. This has implications upon the whole development process used for the system. Conventionally, the process used is conservative and largely sequential, rather than the concurrent hardware and software development prescribed by a codesign process. Sequential development causes many problems, especially late in the lifecycle when it is too late or expensive to change the partitioning between hardware and software components. This paper examines some of the issues when codesign principles are incorporated within a conventional safety-related development process. A key element is the ability to perform architectural trade-off analysis throughout the lifecycle in a manner that results in evidence generated that the resultant design meets both functional and non-functional (i.e. safety) requirements.*

## 1 Introduction

Safety-related embedded systems, such as those found in aerospace applications, are characterised by their functional complexity, size (in terms of required software / hardware), relatively long lifecycles and requirement for validation and verification of their safety prior to deployment [1] Usually, unit cost of hardware is not an overriding concern – the relatively few units made mean one-off development costs are the prime cost consideration.

Conventional development processes for safety-related systems contain an early hard partitioning of system functionality between hardware and software. It is performed with minimal use of architecture trade-off techniques, rather high-level systems engineering principles [2]. Essentially, a "best guess" is made when functions are partitioned between hardware and software. Invariably an underestimate of the amount of software is made (hence the computing platform is under resourced).

One consequence of the safety-related system development process is that hardware is developed in isolation from software, usually prior to software development (due to long lead times for safety-related hardware). Effectively, the overall development process is sequential – hardware development followed by A key element of safety-related system development is the software development, finally integration of software and hardware. A critical problem occurs when / if additional functions are identified after system partitioning, these are usually pushed into software as the hardware is fixed. The hardware is considered fixed as it is expensive to redevelop the hardware to cope with either additional functions or to provide increased computing resource for the software components.

Codesign [12] recognises that systems implement required functions using a mixture of hardware and software components. Trade-offs can be explored regarding whether system functionality is implemented in hardware and software. Given a partitioning of functionality into hardware and software components, design / synthesis of hardware and software can proceed in parallel. Subsequently, the separate hardware and software are integrated to form the final system. A key element of the codesign process is that alternatives for the hardware / software partitioning are evaluated.

Safety-related embedded systems tend to be large in terms of the software size (in the order of millions of lines of software). For such systems, the codesign approach of automatic synthesis (particularly of software) becomes extremely difficult. However, the basic philosophy of the codesign process has important properties that can be incorporated into the conventional design process for safety-related systems. This is despite the fact that long lead times for hardware means some decisions have to be made earlier in the design lifecycle than for conventional systems.

This paper contends that important benefits arise by embedding a codesign process as a sub-process within the conventional safety-related system development process. This enables early partitioning of functions between hardware and software within the traditional safety-related system process, but enables functions identified later to be subject to a codesign process.

To support this approach, two main requirements must be met:

1. As part of the early partitioning, resource is reserved for future functions – e.g. FPGAs which can implement hardware or software functions.
2. The structured capture of design information.

The requirement for resource is to enable the codesign sub-process to partition functionality to hardware and software. Further discussion of this lies beyond this paper.

The capture of design information has two purposes. Firstly, it enables traceability through the development process, which is important in the verification of any safety-related system. Secondly, the design information can be used within the trade-off analysis at the codesign level. The trade-off analysis is to determine how well particular solutions meet the systems objectives in order that the "best guess" at partitioning can be made. The objectives could be properties that it is essential they are met (e.g. meeting of timing requirements) or value added properties (e.g. making the design flexible so that managed changes is supported).

In section 2 the overall approach is described further, explaining how the codesign process is embedded within the overall safety-related system design process. In section 3 a method for architectural trade-off analysis within codesign is given. Finally, conclusions are offered in section 4.

## 2 Approach

Systems engineering for a safety-related development is essentially a standard "V" [3] containing a decompositional phase (left hand side or downward side of "V"), and integration and qualification phase (right hand side or upward path of "V"). Essentially, requirements specification, definition of qualification / validation plan and design specification forms the bulk of the decompositional phase. This results in a number of design specifications that are passed over to individual engineering disciplines for detailed design, implementation and test (e.g. software engineering, electronics / electrical engineering etc.). After the individual disciplines have implemented parts of the system, the qualification phase ensures that the integrated system meets user requirements and sufficient evidence is available for the acceptance of the system by the regulatory authority (i.e. CAA / FAA for civil aircraft). This requires that all stages in the development produce traceable evidence and rationale that the system is sufficiently safe for intended use.

A codesign process can be incorporated within this structure to improve the traditional process and provide codesign of (computer) hardware[1]. However, a vanilla codesign process requires sufficient specification of functions to enable automatic synthesis of hardware and software components. For large safety-related systems, such specification is not usually available until late in the lifecycle. Usually, a minimal specification is available to enable the development of an initial system, with further functions specified later. For example, in the development of an aircraft, a minimal specification is produced to enable a test aircraft to be built. The test aircraft can be used for many experimental purposes including evolving the understanding of the aircraft's dynamic. Subsequent additional functionality is then specified to bring the aircraft up to customer requirements. Ideally, only software is changed between the test and final aircraft.

[1] At this point, a codesign process is not considered for the entirety of the system (including all engineering disciplines), although this does not rule out such an approach in future.

Therefore, the inserted codesign process must allow partial specification of the software, together with an expansion estimate. Together, these will allow hardware to be designed that allows for future expansion due to additional requirements / functions. For the purposes of codesign (specifically the ability to compare and evaluation alternative hardware / software designs), the expansion estimate can be given as resource requirements so that software can be synthesised with the same characteristics.

A key part of the integration of a codesign process into the systems engineering lifecycle is the continued ability of the overall process to collect evidence for system verification. The systems engineering process will produce and collect such information in a traceable and methodical manner – e.g. design rationale, implementation decisions, testing data are collected. The codesign process must function within this environment.

The contention of this paper is that this can be achieved by utilising a single architecture trade-off analysis method for the entire system [4]. As well as conventional trade-off analysis, this also provides a repository for design rationale. At the early stages of the system engineering process (i.e. decompositional phase) this can be used for gross partitioning of the system requirements and specification of sub-systems for the different engineering disciplines. Within the codesign phase, the same architectural trade-off analysis can be used to evaluate different alternative designs. The important outcomes are:

- a single repository of design decisions throughout the entire system, this is key to the production of safety evidence during system qualification), and
- a complementary design and certification approach and architecture [11].

### 2.1 Overview of the Architectural Trade-off

Architectural trade-off analysis for use by the systems engineering process, together with the inserted codesign process has the following properties:

- *derivation of choices* – identifies where different design solutions are available for satisfying a goal.
- *manage sensitivities* – identifies dependencies between components and design decisions.
- *evaluation of options* – allows evaluation of alternative solutions against required properties / specification.
- *influence on the design* – identifies constraints on how components should be designed to support the meeting of the system's overall objectives.
- *collection of design rationale* – forms a repository for design decisions to aid traceability throughout the design

The proposed approach could be used within the nine-step process of the Architecture Trade-Off Analysis Method (ATAM) [5]. The key difference between our strategy and other existing approaches, e.g. ATAM, is the way in which quality attributes are derived. (Quality attributes are the used to evaluate solutions, e.g. does the design support predictability?) Our proposed approach was chosen due to the following reasons.

- the techniques used in our approach are already accepted and widely used.
- the techniques offer strong traceability and the ability to capture design rationale.
- information generated from their original intended use can be reused, rather than repeating the effort.
- 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.

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 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 symbols is given in section 2.2, further details of the notation can be found in [10]) 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. 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. 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. Part of this activity uses representative scenarios to evaluate the 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) is 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.

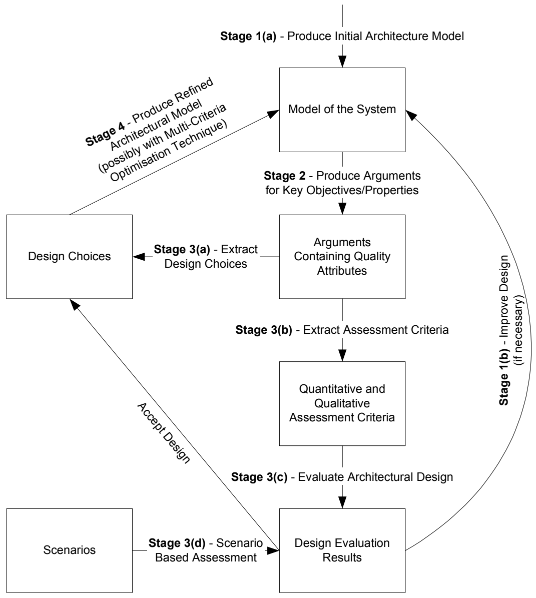
Stage 1(a) - Produce Initial Architecture Model
Model of the System
Stage 4 - Produce Refined Architectural Model (possibly with Multi-Criteria Optimisation Technique)
Stage 2 - Produce Arguments for Key Objectives/Properties
Design Choices
Stage 3(a) - Extract Design Choices
Arguments Containing Quality Attributes
Stage 3(b) - Extract Assessment Criteria
Stage 1(b) - Improve Design (if necessary)
Quantitative and Qualitative Assessment Criteria
Accept Design
Stage 3(c) - Evaluate Architectural Design
Scenarios
Stage 3(d) - Scenario Based Assessment
Design Evaluation Results

**Figure 1 - Overview of the Method**

### 2.2 Background on Goal Structuring Notation

The arguments are expressed in the GSN [10] that is widely used in the safety-critical domain for making safety arguments. In brief, any safety case can be considered as consisting of requirements, argument, evidence and definition of bounding context. GSN - a graphical notation - explicitly represents these elements and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific arguments, how argument claims are supported by evidence and the assumed context that is defined for the argument).

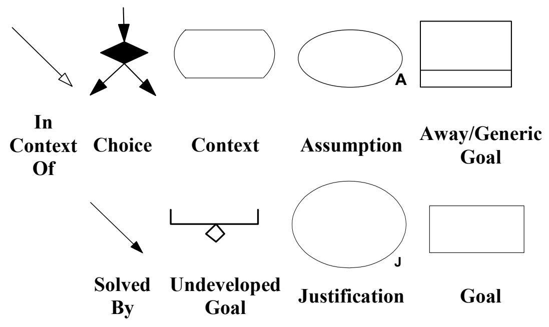

**Figure 2 - Principal Elements of GSN**

The principal symbols in the notation are shown in Figure 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 sub-goals until a point is reached where claims can be supported by direct reference to available evidence (**solutions**). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (**assumptions**, **justifications**) and the **context** in which goals are stated (e.g. the system scope or the assumed operational role). Further details are found in [10].

## 3 Application of Trade-Off Analysis within Hardware Software Co-Design

A key part of architecture trade-off analysis is deriving the top-level properties and objectives (i.e. goal) for the systems such that arguments can be produced that systematically break them down to lower-level goals. These goals are then used to form questions that can be used to judge whether a proposed solution is appropriate. During the production of these arguments, choices of how they can be supported (e.g. implement in hardware or software) will emerge and assumptions identified. (The assumptions are important when trying to reuse designs since they allow the basis for the existing components' design to be evaluated in the new context.) The following subsection proposes some properties for use in hardware software co-design that later in the section are developed into arguments that can be used.

### 3.1 Key Properties for Hardware Software Co-Design

Based on experience and conversations with industrial colleagues, the following objectives have been identified as being important. It should be noted that most objectives are derived from the overarching objective of maximising profit in some way.

- *Correctness* – using appropriate verification techniques sufficient evidence needs to be gathered that what is being produced meets its requirements. Sufficient is dependent on the nature of the application, for example it would be expected in critical systems development that more evidence is expected than for non-critical systems. In general, hardware development is considered to have verification techniques that can provide stronger evidence for correctness.
- *Managed change* – the system produced should be changeable or upgradeable in an efficient manner. For most applications, typical change patterns or potential upgrades are predictable with reasonable confidence. For some applications, there are known killer changes that are likely to occur and result in significant re-design effort being needed. In general, software is considered easier to change but as Ariane 501 demonstrated the assumptions that exist in a component being reused are not always handled appropriately.
- *Efficiency* – the system produced should make the best use of the available resources. The efficiency of a technology is strongly dependent on the nature of the technology. For instance, a FPGA is an effective means (in terms of the amount of silicon used is small) of implementing logic such as found in Statecharts but may not be as effective at implementing floating point operations.
- *Sufficiency* – the technology used in the implementation must be able to represent the design. There are many factors here. Considering just timing,
  - Von Neumann architectures are often considered to have the benefit of providing raw processing power, however for applications where hard real-time guarantees are needed the difficulty in modelling modern Von Neumann processors can lead to large amounts of pessimism in the analysis that reduces/eliminates the benefits [6].
  - FPGAs are better at handling concurrency [7].
  - FPGAs have little or no difference between their best, average and worst-case performance whereas Von Neumann do [7]. Variability in timing behaviour makes many applications, e.g. control systems, harder to produce [8].

The rest of this paper considers sufficiency and its relationship to architectural design in greater detail.

### 3.2 Arguments for Sufficiency

Figure 3 presents the top-level argument for the property *sufficiency*. This shows how the goal, **G0001**, that the implementation strategy is sufficient is broken down into three sub-goals that concern the functional (**G0002**), non-functional (**G0003**) and operational environment requirements (**G0004**). To satisfy any of the goals in Figure 3, the assumption **A0001** has to be satisfied that the requirements are specified in an appropriate manner, i.e. the requirements are complete and consistent – note correctness is not important for the implementation strategy only for the final system being fit for purpose. Goal **G0004** that deals with operational environments is left undeveloped.

Goal **G0002** that deals with functional requirements is satisfied using a generic goal **G2001** – *Implementation Sufficient for the Requirements to be Met*. The argument (**SufficiencyArg**) for the generic goal is presented in Figure 5. In the case of functional requirements it is assumed, **A0002**, either hardware or software can equally well be used for meeting the requirements. Goal **G0003** is decomposed into an argument that deals with whether the implementation strategy can satisfy the system's non-functional requirements. The goal **G0003** is split into five parts (one for each sub-property of non-functional -memory, safety, timing, reliability and communications) which are also satisfied by the generic goal **G2001**.

In Figure 5, the argument for goal **G2001**, originating in Figure 3, is split into three parts; the implementation allows the requirements to be met (goal **G2002**), requirements are specified appropriately (goal **G2003** which is left un-developed – indicated by a diamond below the goal), and sufficient evidence can be gathered the requirements are met (goal **G2004**).

The goal **G2004** results in two choices (a choice means that at least one of *N* strategies proposed is followed) which are evidence is either gathered by static or dynamic analysis. A feature of the technique that has been developed is that where a choice exists then justifications and assumptions should be captured and these normally relate to pros and cons of the choices that need to be explored. In this case, the assumptions and justifications indicate that static analysis has the advantage it can show absence of faults but getting appropriate models is difficult, hard to validate and often not practicable, whereas dynamic analysis is more generally applicable but can't guarantee the absence of failures. Often, the best compromise is a combination of the options.

The goal **G2002** is split into three parts; that the infrastructure is predictable (goal **G2005**), that the mapping of the application onto the infrastructure is predictable (goal **G2006**), and the application can be shown to meet its requirements (goal **G2007**). From the goals the key points that emerge are the choices in the hardware used (between FPGA, microprocessor and discrete circuitry) and the design notation used (between hardware-based and software-based languages). The latter of these choices shows that software-based languages have the advantage that they can represent both hardware and software but a disadvantage is raised that hardware-based languages often support devices and concurrency better.

Each of the hardware choices can be satisfied by the same argument, given in Figure 4. This argument shows that the choice should be based on whether the type of hardware has appropriate models available and whether these can be validated.

## 3.3 Using the Arguments

From the argument in section 3.2, a number of design choices and objectives (i.e. goals or quality attributes) have been identified. This section is to consider how these objectives should be used during the hardware software co-design process; issues include whether they should be used as part of a qualitative or quantitative assessment, whether the process is manual or automated and how they are used as part of making an overall decision. In general, qualitative assessments consist of asking questions which are based on experience and some consideration whereas quantitative assessment consists of a checklist of activities to be completed. A key difference is qualitative assessment can be performed anytime during a project whereas quantitative assessment often relies on certain design information (e.g. task execution times in the case of timing analysis) being available which may dictate when it can be performed.

Table 1 provides a partial trade-off analysis of the objectives given in the arguments presented in section 3.2. The analysis has lead to qualitative and quantitative assessments being derived as well as an indication of the relative importance of each. The rest of the section explains how this can be used as part of hardware software co-design. The qualitative assessment criteria can be used elsewhere as part of review checklists, and the quantitative assessment criteria are verification and validation requirements (e.g. the need to perform timing analysis).

Table 1 shows many of the quality attributes that have to be made when performing co-design. When performing co-design it would be necessary to produce a balanced design where the important constraints are met and others are achieved as well as possible.

Putting this into context, consider the design of a system whose principal function is to support a number of control loops. Key requirements for such a system are the ability to meet requirements and the ability to detect when the system is not performing as expected. Taking the first of these and considering timing, the main requirements are that precedence constraints are maintained, freshness of data when used in calculations and later output, and jitter requirements because of its impact on stability.

One co-design decision is whether a conventional microprocessor is used or a FPGA. For simplicity when making this decision, other variables could be considered as fixed, for example Ada as the programming language. Consider the assessment related to goal **G3001**, for a FPGA the models of the circuitry (i.e. individual cells and overall circuitry) are well-defined and comprehensive models available. However for anything other than a simple microprocessor models are rarely available, and even if they were validation would be difficult if not impossible [9]. This is particularly the case for timing where worst-case execution time analysis is hard because the information provided by the manufacturers is often difficult to interpret and containing errors and if a model can be derived then it is especially hard to validate and often pessimistic [9]. Additionally from a timing perspective related to **G2014**, it would be hard to meet the control systems timing requirements because modern microprocessors tend to have high variability in their executions, whereas FPGAs tend to have constant execution times, which makes meeting jitter requirements difficult [7]. Also, FPGAs support concurrency better than microprocessors which ease the problem of supporting multiple control loops.

Despite the problems of using microprocessors, obviously as common practice suggests they do provide a means to implement control systems especially where much of the processing demands does not need strict timing behaviour (e.g. health monitoring functionality). One key advantage of the trade-off analysis approach proposed is that the output helps guide the design of the system, for example the design of the scheduler should minimise the jitter of certain tasks related to the operation of the control loops.

Another key benefit of the arguments and trade-off analysis is the way they support the objective of the systems engineering process being partitioned into clear and distinct parts that can then be performed as independent entities. For instance in Figure 5, there are goals for whether the implementation meets its requirements. From this, sub-goals are derived that separate the meeting of the requirements for the application, infrastructure and mapping from one another. These activities can then be performed in isolation of one another. However the goals capture key assumptions that must be considered and later shown to hold such that when the activities merge back together that the overall requirements are met. As the design is further decomposed, more assumptions between these activities would be derived.

## 4 Summary and Future Work

This paper has shown how the codesign process can be incorporated as a sub-process within the conventional safety-related system process. A key integrating technology between the conventional process and codesign process is the ability to collect design evidence throughout the overall process, utilising it for both verification and trade-off analysis within the codesign sub-process.

This paper has proposed that this can be achieved by utilising a single architecture trade-off analysis method for the entire system. As well as conventional trade-off analysis, this also provides a repository for design rationale. At the early stages of the system engineering process (i.e. decompositional phase) this can be used for gross partitioning of the system requirements and specification of sub-systems for the different engineering disciplines. Within the codesign phase, the same architectural trade-off analysis can be used to evaluate different alternative designs. The important outcomes are:

- a single repository of design decisions throughout the entire system, this is key to the production of safety evidence during system qualification), and
- a complementary design and certification approach and architecture.
- a means to focus the co-design process so that different ways of satisfying the system's objectives can be traded off.

Current work is looking at how the process may be automated and proving the concepts in this paper with a case study. Automation could be achieved for instance by the use of heuristic search algorithms, such as genetic algorithms, to optimise the design with respect to its objectives.

## 5 References

[1] Y.C. Yeh, *Dependability of the 777 Primary Flight Control System*. Proceedings of the 5th IFIP Conference on Dependable Computing for Critical Applications, 1995.

[2] D. M. Buede, *The Engineering Design of Systems*, pub. Wiley, 2000.

[3] K. Forsberg, H. Mooz, *The Relationship of Systems Engineering to the Project Lifecycle,* Engineering Management Journal, 4(3), 36-43, 1992

[4] I. Bate, N. Audsley, *Architecture Trade-off Analysis and the Influence on Component Design*, Proceedings of Workshop On Component-Based Software Engineering: Composing Systems from Components, 2002.

[5] R. Kazman, M. Klein, and P. Clements, *Evaluating Software Architectures - Methods and Case Studies*. 2001: Addison-Wesley.

[6] I Bate, P Conmy, T Kelly, J McDermid, *Use of Modern Processors in Safety-critical Applications*, The Computer Journal, 44 (6), 531-543, 2001.

[7] M. Ward, N.C. Audsley, *Hardware Compilation of Sequential Ada,* Proceedings of CASES, 2001.

[8] I Bate, *Scheduling and Timing Analysis of Safety Critical Hard Real Time Systems*, Thesis, Department of Computer Science, University of York, YCST-99-04, 1999.

[9] J. Engblom: *Processor Pipelines and Static Worst-Case Execution Time Analysis*, Uppsala Dissertations from the Faculty of Science and Technology, ISBN 91-554-5228-0, 2002.

[10] T. Kelly, *Arguing Safety – A Systematic Approach to Safety Case Management*, DPhil Thesis, YCST-99-05, Department of Computer Science, Univ. of York, 1998.

[11] I. Bate, T. Kelly, *Architectural Considerations in the Certification of Modular Systems*, To Appear in Proceedings of 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2002), 2002.

[12] G. de Micheli, R. Ernst, W. Wolf, *Readings in Hardware / Software Codesign,* pub. Morgan Kaufmann, 2002.

| Goal | Qualitative | Quantitative | Importance |
|---|---|---|---|
| **G3001** – Hardware used is predictable | For proposed hardware, raise questions such as:<br>a. Does documentation exist?<br>b. Does it seem comprehensive? | Choice (originating from goal **G2004**) of either:<br>a. derivation of models for the different non-functional properties that are validated against the actual hardware.<br>b. development of an appropriate testing strategy. | Normally high, but dependent on the integrity the part of the system and particular property needs to attain. |
| **G2017** and **G2018** – Use a software or hardware based language | For a proposed language, raise questions such as:<br>a. Can sufficient engineers be found?<br>b. Have previous systems been successfully developed with it?<br>c. Does the language provide a set of features that are probably sufficient for the envisaged application?<br>d. Does the language allow static analysis to be performed?<br>e. Are support tools available? | Assessment activities could include:<br>a. Related to **G2013**, obtaining/defining semantics.<br>b. Related to **G2008**, obtaining/producing complementary static analysis tools.<br>c. Related to **A2006** and **A2007,** determine the language features needed and whether they are supported.<br>d. Related to **J2003** and **J2004**, is an appropriate mapping available from software to hardware? | High since changing the language used part way through development can lead to large amounts of (if not total) rework. |
| **G2014** – Infrastructure provided is sufficient | For a proposed infrastructure, raise questions such as:<br>a. Based on previous experience, does the resources available seem sufficient? E.g. Are more MIPS available than for other similar projects<br>b. Does the hardware provide a set of features that are probably sufficient for the envisaged application?<br>c. Are support tools available? | a. Related to **G2004,** early in the project, use data from previous similar systems and other metrics (e.g. number of requirements) to estimate whether infrastructure means the requirements can be met. Later in the project, use actual data obtained via static or dynamic analysis. At all stages, possibly perform sensitivity analysis to de-risk further development.<br>b. Related to **G2014**, determine the hardware features needed and whether they are supported. | Depends on how easily the infrastructure or application can be changed. Often from an early stage in projects changing the infrastructure or application design are not options, therefore importance would be high. |

**Table 1 – Assessment Using the Contents of the Arguments**

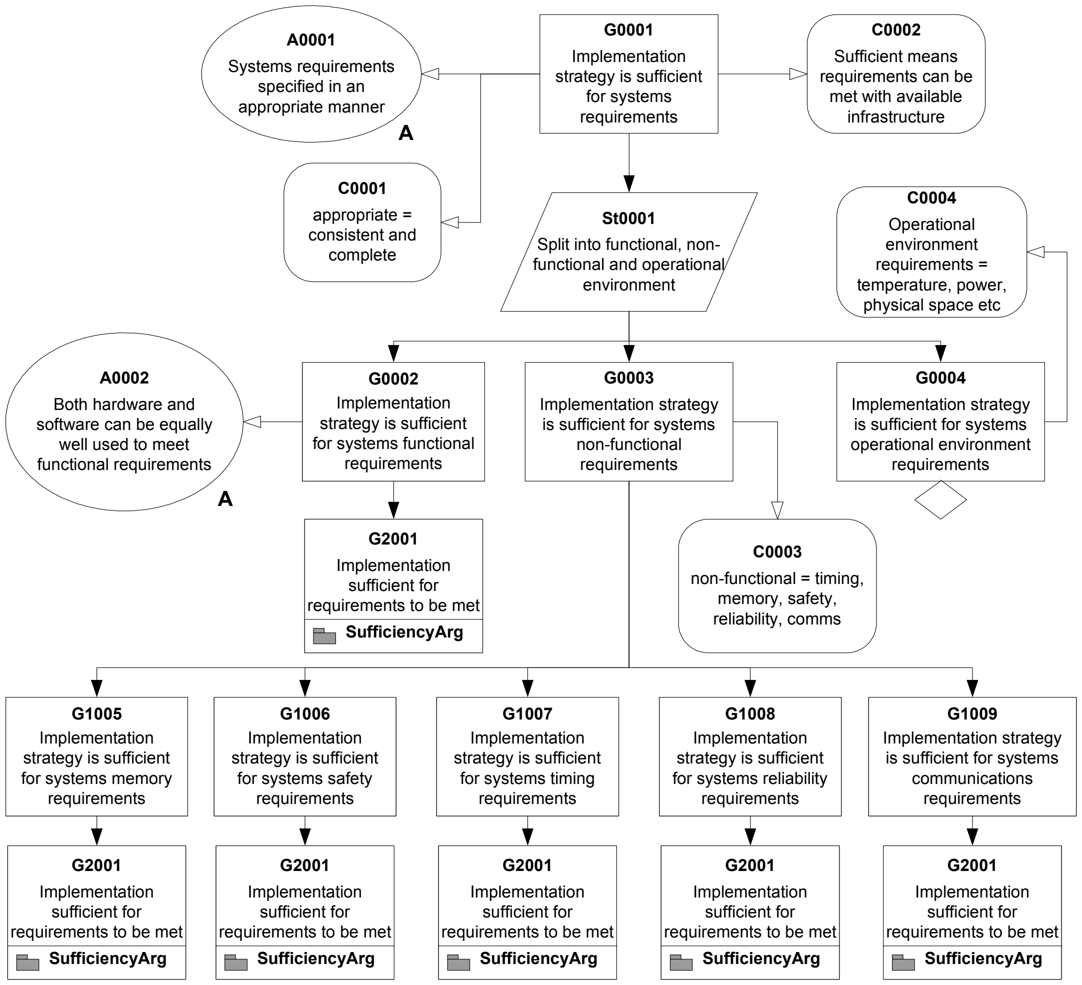
A0001
Systems requirements specified in an appropriate manner
A
G0001
Implementation strategy is sufficient for systems requirements
C0002
Sufficient means requirements can be met with available infrastructure
C0001
appropriate = consistent and complete
St0001
Split into functional, non-functional and operational environment
C0004
Operational environment requirements = temperature, power, physical space etc
A0002
Both hardware and software can be equally well used to meet functional requirements
A
G0002
Implementation strategy is sufficient for systems functional requirements
G0003
Implementation strategy is sufficient for systems non-functional requirements
G0004
Implementation strategy is sufficient for systems operational environment requirements
G2001
Implementation sufficient for requirements to be met
SufficiencyArg
C0003
non-functional = timing, memory, safety, reliability, comms
G1005
Implementation strategy is sufficient for systems memory requirements
G1006
Implementation strategy is sufficient for systems safety requirements
G1007
Implementation strategy is sufficient for systems timing requirements
G1008
Implementation strategy is sufficient for systems reliability requirements
G1009
Implementation strategy is sufficient for systems communications requirements
G2001
Implementation sufficient for requirements to be met
SufficiencyArg
G2001
Implementation sufficient for requirements to be met
SufficiencyArg
G2001
Implementation sufficient for requirements to be met
SufficiencyArg
G2001
Implementation sufficient for requirements to be met
SufficiencyArg
G2001
Implementation sufficient for requirements to be met
SufficiencyArg

**Figure 3 - Top Level Argument**

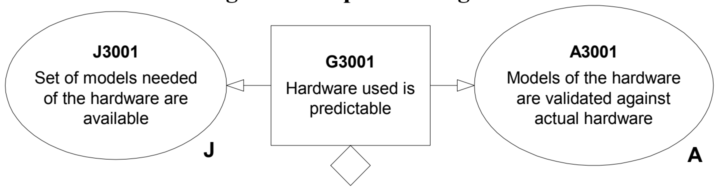
J3001
Set of models needed of the hardware are available
J
G3001
Hardware used is predictable
A3001
Models of the hardware are validated against actual hardware
A

**Figure 4 - Hardware is Predictable – *"HWPredArg"***

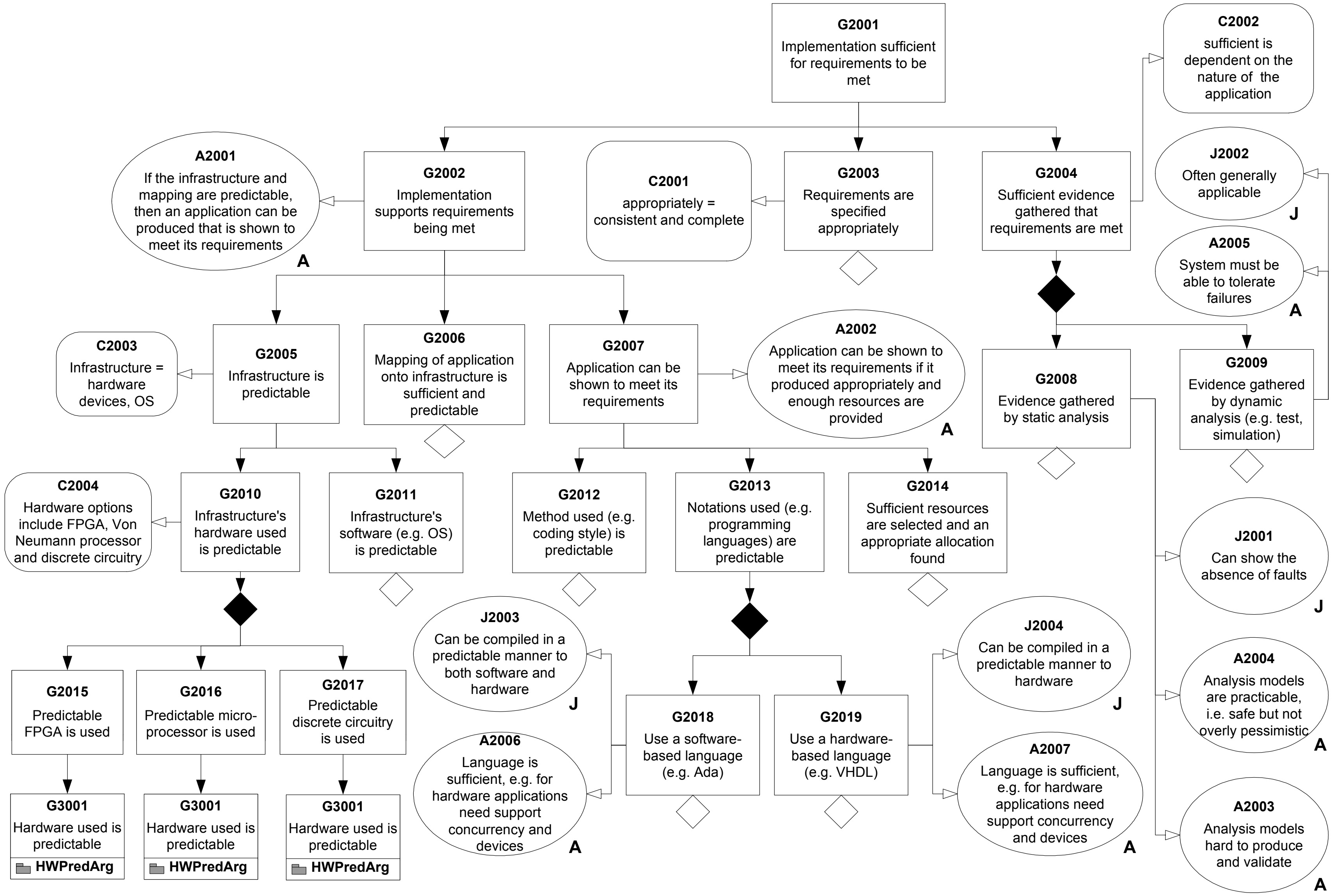
G2001
Implementation sufficient for requirements to be met
C2002
sufficient is dependent on the nature of the application
G2002
Implementation supports requirements being met
A2001
If the infrastructure and mapping are predictable, then an application can be produced that is shown to meet its requirements
A
C2001
appropriately = consistent and complete
G2003
Requirements are specified appropriately
G2004
Sufficient evidence gathered that requirements are met
J2002
Often generally applicable
J
A2005
System must be able to tolerate failures
A
C2003
Infrastructure = hardware devices, OS
G2005
Infrastructure is predictable
G2006
Mapping of application onto infrastructure is sufficient and predictable
G2007
Application can be shown to meet its requirements
A2002
Application can be shown to meet its requirements if it produced appropriately and enough resources are provided
A
G2008
Evidence gathered by static analysis
G2009
Evidence gathered by dynamic analysis (e.g. test, simulation)
C2004
Hardware options include FPGA, Von Neumann processor and discrete circuitry
G2010
Infrastructure's hardware used is predictable
G2011
Infrastructure's software (e.g. OS) is predictable
G2012
Method used (e.g. coding style) is predictable
G2013
Notations used (e.g. programming languages) are predictable
G2014
Sufficient resources are selected and an appropriate allocation found
J2001
Can show the absence of faults
J
J2003
Can be compiled in a predictable manner to both software and hardware
J
J2004
Can be compiled in a predictable manner to hardware
J
A2004
Analysis models are practicable, i.e. safe but not overly pessimistic
A
G2015
Predictable FPGA is used
G2016
Predictable micro-processor is used
G2017
Predictable discrete circuitry is used
G2018
Use a software-based language (e.g. Ada)
G2019
Use a hardware-based language (e.g. VHDL)
A2006
Language is sufficient, e.g. for hardware applications need support concurrency and devices
A
A2007
Language is sufficient, e.g. for hardware applications need support concurrency and devices
A
A2003
Analysis models hard to produce and validate
A
G3001
Hardware used is predictable
HWPredArg
G3001
Hardware used is predictable
HWPredArg
G3001
Hardware used is predictable
HWPredArg

**Figure 5 - Implementation Meets the Requirements – *“SufficiencyArg”***