

# Achieving Dependable Systems By Synergistic Development Of Architectures And Assurance Cases

Patrick J. Graydon<sup>1</sup> John C. Knight<sup>1</sup> Elisabeth A. Strunk<sup>2</sup>

<sup>1</sup> Department of Computer Science  
University of Virginia  
151 Engineer's Way, P.O. Box 400740  
Charlottesville, VA 22904-4740, USA  
(graydon|knight)@cs.virginia.edu

<sup>2</sup> Software Systems Engineering Dept.  
The Aerospace Corporation  
15049 Conference Center Drive, CH3/320  
Chantilly, VA 20151-3824  
elisabeth.a.strunk@aero.org

**Abstract.** *Assurance Based Development* (ABD) is an approach to the construction of critical computing systems in which the system and an argument that it meets its assurance goals are developed simultaneously. ABD touches all aspects of the system lifecycle, but in this paper we focus on how the evolving assurance argument can guide architectural choices to increase system dependability. The goals with this approach to architectural choice are twofold. The first is to develop the architecture so that it provides the required evidence. The second is to refine the assurance case as architectural choices are made so that the evidence that will be provided supports the assurance claims. Combining development and assurance in this way facilitates detection—and thereby avoidance—of potential assurance difficulties as they arise, rather than after development is complete.

## 1 Introduction

It is essential that there be a high degree of assurance that a critical computing system will operate dependably in its expected environment, and system architecture plays a major role in achieving that dependability. Unless the architecture of the system is well-matched to both its dependability needs and the associated assurance of that dependability, developers may waste effort on activities that bring unnecessary gains in one part of a system while failing to provide the needed assurance of dependability in others. This can happen in any phase of the lifecycle, but it is especially important for a system's architecture because inappropriate architectural decisions can have a major impact on subsequent development activities. Nevertheless, current approaches to assuring dependability—in the system's architecture, and in other aspects of its development—are frequently ad hoc.

*Assurance Based Development* (ABD) is a novel approach to the development of critical computing systems in which development of the system and of its assurance argument are integrated. The assurance argument, presented in an *assurance case*, documents how evidence from the system and the process used to construct it supports the system's dependability claims. Integrating system development and system assurance means that dependability-related development objectives are clearly laid out, and can be addressed specifically when making architectural choices. In this paper, we describe the ABD process and how it can be used to develop a software architecture.

In ABD, the assurance case and architecture are developed in parallel. Each architectural choice a developer makes is assessed in terms of its impact on: (1) the system's functionality; (2) the development activities that will be needed; (3) the evidence that will be needed in the assurance case; and (4) the argument structure of the assurance case. The goals with this approach to architectural choices are twofold. The first is to develop the architecture so that it provides the evidence required in the assurance case. The second is to refine the assurance case as architectural choices are made so that the evidence which will be provided supports the claim that the system is adequately dependable.

Combining architecture development and dependability assurance in this way promotes detection—and thereby avoidance—of potential assurance difficulties as they arise. Inevitably, architectural choices are made without complete knowledge of their impact on subsequent development, and so it is always possible that a decision will have to be rethought. Nevertheless, by including explicit attention to dependability assurance goals while creating the system's architecture, the chances that a decision will not support the overall dependability goal are reduced. Where this is not done, there is the very real danger that inadequate dependability might only be revealed during evaluation carried out after development is complete.

Since dependability assurance is considered and addressed throughout development, ABD can increase the confidence that can be placed in an architecture. Furthermore, the increased efficiency of the development processes allows resource savings during development of typical critical systems when ABD is used. Nevertheless, it is not possible to show that architectures chosen with the help of the assurance case will always be superior to architectures chosen in an ad hoc manner because of the many variables involved in development. The architect will have been able to make better informed choices than that would have been possible otherwise, and so his or her goals are more likely to be achieved.

## 2 Assurance Cases

Assurance cases are the state of the art in rigorous but non-formal dependability argumentation and, as such, provide the foundation on which the ABD approach to architectural development rests. The most common use of assurance cases at present occurs in the documentation of *safety*, and safety cases have been built for a variety of production systems. In general, a safety case is “a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment” [2]. Special graphic notations have been designed to enable the documentation of assurance cases in a manner that is easy for humans to understand and that can be manipulated by machine. The most widely used of these notations is the Goal Structuring Notation (GSN) [17].

In its simplest form, an assurance case contains an instance of each of three essential elements: (1) an assurance goal or claim; (2) evidence that the goal has been satisfied; and (3) an argument linking the evidence to the goal in a way that leads one to believe that the goal is justified by the evidence. This basic structure is supplemented with a variety of other elements, including assumptions, justifications, and context,

and applied recursively to produce, for real systems, a hierarchic structure with the overall goal for the system at the root. The hierarchic structure makes the overall argument manageable at each level. Figure 1 illustrates the use of GSN in a simple *completely hypothetical* safety case. In the figure, the assurance goal is stated in the box at the top and the remainder of the figure documents the argument for belief in that goal.

The overall argument in an assurance case is a set of logical inferences that show why the evidence implies that the system's assurance goals have been met. The goals, evidence, and hence the assurance argument are specific to a particular system, and so each assurance case is unique. However, patterns have been developed for common argument fragments [10].

### 3 Assurance Based Development

Assurance Based Development integrates the various separate activities that occur in the construction of a critical system, i.e., it integrates requirements and context analysis, system development, and assurance case creation. The primary goal is to ensure that the choice of development techniques will allow evidence generated during development to be sufficient for the system's assurance case. Creating a system architecture

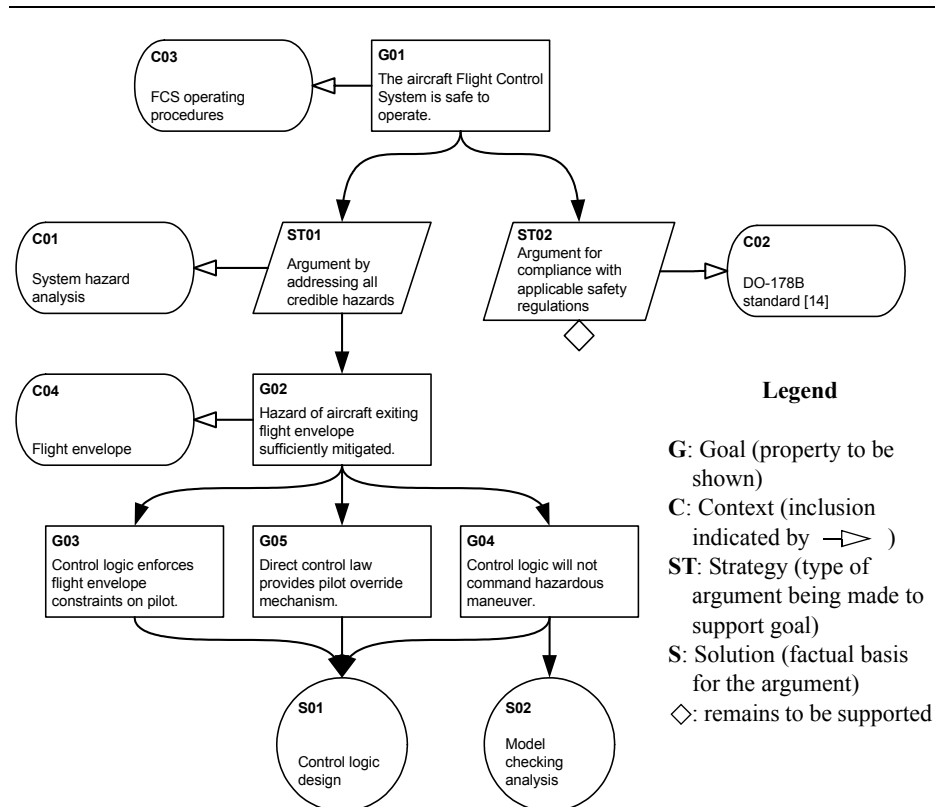
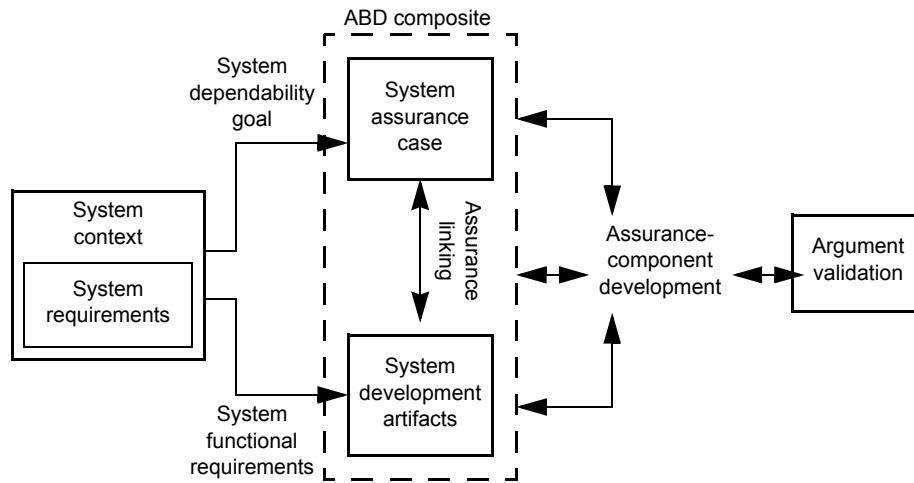


Figure 1. Example assurance case



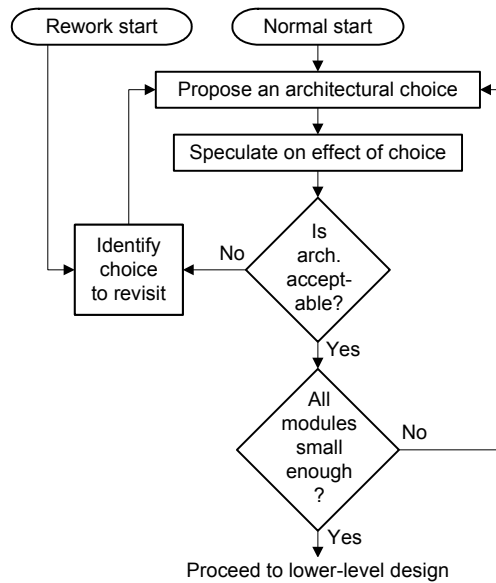
**Figure 2. Assurance Based Development**

is one stage of ABD. We describe the overall ABD process in this section, elaborating its implications for system architecture below.

The major components of Assurance Based Development and their high-level interactions are shown in Figure 2. Shown on the left of the figure are components labeled *system context* and *system requirements*, with the former enclosing the latter; the context in which a system operates influences the system requirements in many ways. The system requirements are used by both the system assurance case and the system development artifacts. The system requirements include the dependability requirements such as availability and safety, and thus determine the primary goal of the assurance case. The system requirements include the functional requirements also and so are the starting point for the development lifecycle.

At the center of the technique are the *system assurance case* and the *system development artifacts*. These two components are developed in parallel, and their development is coordinated using a technique that we refer to as *assurance linking*. Assurance linking ensures that assurance goals and development artifacts are coupled explicitly and systematically so as to reveal the evidence needed by the assurance case from the development artifact. Assurance linking enables developers to check that the properties possessed by development artifacts are the properties necessary to support the goal in the assurance case for which the development artifact provides evidence.

As an example of assurance linking, consider the problem of developing a software component in a safety-critical system such that the component meets an assurance goal of having a failure rate per unit time below some threshold  $p$ , where  $p$  is perhaps  $10^{-3}$ , i.e., not in the ultra-dependable range. Testing might be the basic strategy chosen to meet this goal. Such a goal requires several pieces of evidence if it is to be believed as part of an assurance case. These pieces of evidence are: (1) that the specification for the component is correct (a complex assurance subgoal); (2) that the component has been tested according to its test plan (a development subgoal); (3) that testing



**Figure 3. Traditional process for architecture development**

according to the test plan demonstrates in a statistically valid way that its failure rate is below the threshold (a developmental and documentation subgoal); (4) that the test cases were the result of a random process of selection from the expected operational environment (an analysis subgoal); and (5) that these items of evidence were recorded and reported accurately.

A technique called *ABD composite production* is the process of developing a component and its associated assurance case elements. The assurance case plays a predictive role in ABD since it is used to determine the necessary properties that each development artifact must have in order to support the system assurance argument.

Development of an architecture, therefore, consists of repeatedly making an architectural choice, assessing the value and suitability of the evidence that will result from it, and developing the associated assurance argument fragment. The assurance argument fragment is then analyzed to ensure that its derived premises are both satisfiable and practical, and that it is free of fallacious reasoning and other flaws.

If the synergy between the system and its assurance case is not present, then the basis for each development choice, including those made in architecture, will tend to be factors such as cost, experience and convenience although dependability will sometimes be considered. Thus there will be no guarantee that the choices made will be those that facilitate the system meeting its dependability goals nor that the evidence developed will be that needed in the assurance case.

#### 4 Existing Techniques For Architecture Development

In any discussion of architecture, it is important to have a precise definition of what the term means. In this paper, by the term architecture, we mean the following:

The architecture of a system is the system's fundamental organization, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. [1]

There is no rigorous, prescribed process for developing a system architecture or a software architecture. There are several principles that are known to promote certain useful qualities in an architecture, e.g., modularization that facilitates information hiding, and there are architectures that are commonly used because they possess important properties, e.g., redundancy allows certain types of fault to be masked.

In practice, most architectural development tends to be driven primarily by desired system functionality. The dominant technique used, therefore, is functional decomposition using informal “box-and-arrow” models. Once a model is thought capable of supplying the necessary functionality, attention turns to issues such as encapsulation and performance. Dependability as an architectural issue is often either: (1) assumed to be addressed by some architectural pattern; (2) an afterthought that has to be argued with an architecture that is *a fait accompli*; (3) argued in a fragmented and uncoordinated manner as development proceeds; or (4) some combination of items (1)-(3). The first approach leads to a partial argument where the contribution to dependability or lack thereof by many of the architectural decisions is missing. The second approach can lead to a lot of rework as architectural decisions are discovered that do not meet the needs of the assurance argument. Finally, the third approach leads to an incomplete and unsatisfactory argument.

Developers of safety-critical systems pay attention explicitly to system dependability both during development and during evaluation, and the process they follow shares some characteristics with ABD. However, although dependability goals influence architectural choices, the goals considered are typically just overall system reliability or availability targets, and the process tends to have the characteristics of item (3) above.

In the experience of the authors, for systems in general, architects often use an iterative process such as the one shown in Figure 3. In each iteration, the architect makes a tentative architectural choice based on functional decomposition and then examines the software as it would be given that choice using a box-and-arrow style of model. He or she speculates as to whether, with respect to obvious alternative choices, the new choice combined with the previous choices would:

- allow the resulting system to meet its functional requirements;
- allow partitioning of the software into modules that can be divided evenly over the available team members;
- result in a system with adequate performance; and
- result in software with acceptable volume and complexity.

An architect typically starts the development of an architecture with a choice that he or she hypothesizes will allow the required overall system functionality to be provided. Understandably, an architect usually proposes architectural choices with which his or her team is already familiar: a team that has used the same architectural pattern in the last several projects will often start by proposing that pattern for the next project. Clearly, they could not propose an architectural choice with which they were totally unfamiliar. Also, architects are hesitant to use architectural choices with which they

have little or no experience, as their unfamiliarity will make their conclusions about the acceptability of the choice tenuous, thus adding to perceived project risk.

Selection of architectural choices is guided by the architect's sense of the next most "pressing" issue. Once convinced, albeit informally, that their choice could supply the necessary functionality, the architect turns to issues such as performance, flexibility, maintainability or similar. Following that the architect might turn to mundane but important issues such as whether the components of the architecture can be implemented effectively by the available team structure.

This development process does not guarantee an acceptable architecture. As a tentative architecture becomes more complex, answering questions such as whether proper functionality will be supplied or whether acceptable performance will be achieved requires techniques such as architectural modeling and prototyping. In some cases, the answers are not obtained until the system is built at which point the "wrong" answer can be disastrous because selecting a new choice is either impractical or very expensive.

When the perceived drawbacks are costly enough, however, architects will revisit choices. They will replace them with alternatives and re-assess the system as a whole. They will then consider replacing any further choices that examination reveals to be sub-optimal given the repaired choice.

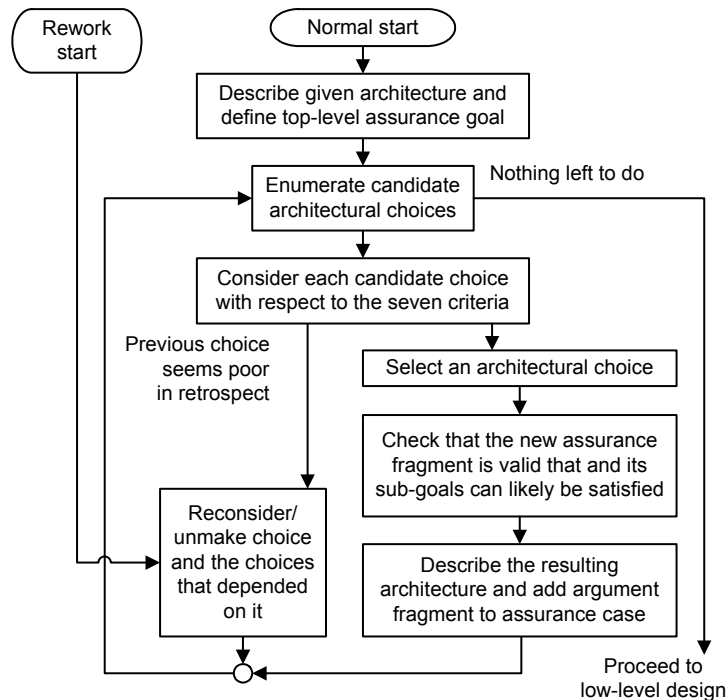
In some cases, architectural choices are affected by non-technical considerations such as standards, imposed choices, and workload balancing. External standards often have to be followed to facilitate certification in regulated industries such as medical devices. Similarly, development decisions such as the choice of target hardware platforms or the choice of software development tools are sometimes imposed as an enterprise-wide decision. Finally, limited development resources may force engineers to accept compromises that increase development risk, reduce functionality, or make maintenance or enhancement more difficult.

In the special case of safety critical systems, the general architectural approaches summarized above are sometimes enhanced by including consideration of the safety argument during development. For example, it has been advocated that developers create a preliminary safety case early in the system lifecycle and update it periodically as development progresses [9, 12]. ABD takes this idea to its limit by tightly coupling the development of the system and its assurance argument so as to make visible to the developer the assurance obligations incident upon each part of the system.

Attribute Driven Design (ADD) is an architectural development technique that is closely related to ABD [18]. In ADD, architectural choices are influenced by quality properties as well as more conventional software architecture considerations. The major difference between ABD and ADD is that ABD uses the rigorous argument of a safety case to fully document the rationale for believing that quality attributes have been achieved and guides development to ensure that this argument will be valid.

## **5 Architecture In Assurance Based Development**

As discussed in section 3, the fundamental concept underlying Assurance Based Development is that system development decisions are based on being able to meet



**Figure 4. Architecture development in ABD**

assurance goals and on supporting the associated assurance argument. The development of a system's architecture is a part of the overall development, and so the architectural decisions in ABD are based on meeting assurance goals. With this approach, the role of architecture in dependability is clear, and the architectural choices made can be optimized to achieve the desired dependability and the associated assurance without expending unnecessary cost. The ABD process is summarized in Figure 4.

The starting point, as with any development, is the *given architecture*. This is the high-level architecture within which the computing system will operate. The high-level architecture will define the functional requirements for the computing system and the associated dependability requirements. The top-level goal in the computing system's assurance argument is derived from these requirements. Thus, any system that demonstrably meets this goal solves the problem that it was created to solve, providing the desired functionality with the desired dependability when operated in the intended context.

The ABD process for developing a system architecture proceeds differently from the process discussed in the previous section. The architect begins by examining the top-level goal in the assurance case. This goal will include the dependability requirements as well as the functionality itself. The architect seeks an architectural choice that will allow the goal to be met. His or her first step is to make an architectural choice that results in a satisfactory argument that the goal has been met and sub-goals that can be addressed practically by subsequent development activities. Each choice is evalu-

ated in terms of seven criteria, which are discussed in detail in section 5.2. These criteria have been selected so as to help developers avoid making choices that will need to be re-visited because they make successful completion of the system and its assurance case impractical or even impossible.

Once the choice is made, the evidence that results from the choice becomes an integral part of the argument that the top-level goal of the assurance case is met. Thus acceptance of the choice by the architect requires that he or she develop the associated argument fragment for the assurance case. This argument fragment must show that the top-level goal will be met, provided the premises generated by the architectural choice are met.

As an example of the concept, consider the development of a simple autopilot system that has to provide an altitude-hold capability for a general-aviation aircraft. Such a requirement could be implemented as a control loop that reads a set of sensors, computes an adjustment to the actuators, and sends commands to the actuator servos according to a simple, periodic real-time schedule. The dependability requirements would be quite extreme, however, because safe flight depends on the system operating correctly when in use [14]. In this case, a system reliability of  $1 \times 10^{-7}$  over three hours together with a significant safety requirement might be required.

The given architecture for such a system would include much of the rest of the aircraft's avionics system. In particular, the mechanisms by which the sensor signals are made available and the actuators are commanded would be defined.

The architect faced with the stated assurance goal would need to make an initial architectural choice that would ensure hardware and software failure rates which, when suitably composed, would meet the goal. The choice, therefore, might include an NMR hardware system<sup>1</sup> combined with software developed with a rigorous process and a comprehensive test plan. The detailed characteristics of the selected NMR hardware system and of the software development process would be the evidence for the argument fragment, and the architectural choice would not be accepted unless this argument was considered adequate.

In the remainder of this section, we describe the Assurance Based Development of architecture in detail. In section 5.1 we discuss the determination of architectural choices and in section 5.2 selection from them. In section 5.3 we present the use of architectural choices, and we review the overall process in section 5.4. A detailed example of ABD applied to architecture is presented in section 6.

### 5.1 Candidate Architectural Choices

With dependability as a central criterion in making architectural choices, it is important to ensure that all possible candidate choices are considered. Several general textbooks have been written on architecture [15, 3, 4, 5], and a great deal of research has been conducted on architectural support for dependability [e.g., 16], and it is not necessarily the case that the architect will be familiar with the field. This research has

---

1. N Modular Redundancy (NMR) executes N replicates of a system in parallel on separate hardware and selects an output by voting. Defects in a minority of the replicates can be masked with this technique. N is often three giving Triple Modular Redundancy (TMR).

been motivated in most cases by seeking to achieve specific dependability metrics (e.g., reliability, availability and safety) within general system categories (e.g., servers, clients and embedded systems), and the relevant literature tends to be organized from these perspectives.

Determining the architectural choices in ABD is a two-phase process. In the first phase, the architectural choices that might be able to address the current subject assurance goal are enumerated. Care has to be taken to include: (1) all available general architectural patterns; (2) all architectural choices from experience with similar systems; and (3) architectural choices from beyond the architect's personal experience.

In the second phase, the functionality and assurance evidence that each choice can provide is elaborated. Because the candidate choices are necessarily generic, the architect must consider how the choice would apply to the situation at hand and determine the functionality and evidence that would result.

Patterns are a general and commonly used technique, and they have proven especially important in architecture and design. They can be used in Assurance Based Development where they consist of architectural choices coupled with assurance case argument fragments, the composite being used to capture experience thereby allowing future developers contemplating similar architectural decisions to benefit from that experience.

## 5.2 Selection Of An Architectural Choice

Selection of a suitable architectural choice from the enumerated candidate set is based on seven criteria (discussed below): functionality, subsequent restrictions, dependability, cost, feasibility, standards, and additional non-functional requirements. A candidate architectural choice can be rejected based on just one or several of the criteria, or it can be modified to suit the needs of the system under development if a change can deal with the problem.

Much of the pruning of the set will be based on the architect's experience. In many cases, an experienced architect might consider only a single candidate architectural choice in which he or she has considerable confidence. In such a case, these criteria are exit criteria from the selection process for that choice.

It might appear that, except for dependability, these criteria will have the same meanings and roles in ABD as they do in traditional architectural development. This is not the case, however, because several of the criteria can influence and can be influenced by the dependability argument. Note also that these criteria are not disjoint, and so evaluating a criterion cannot necessarily be done in isolation. We examine each criterion briefly with an emphasis on its overall role in dependability.

- **Functionality.** Once an architectural choice has been instantiated for the system under development, the architect needs to check by inspection, analysis, prototyping and/or modeling that there are no detectable deterrents to achieving the desired functionality. The functionality criterion is the same in ABD as in existing techniques.
- **Restrictions imposed on later choices.** To a greater or lesser extent, each architectural choice that is made restricts subsequent choices throughout the rest of software development. Making an architectural choice at one level of abstraction

generates subgoals for subsequent refinement, and those subgoals can only be met in certain ways. Of particular importance in this context is the possibility of an architectural choice restricting the selection of techniques later in the process that either facilitate dependability or contribute to its assessment.

- **Evidence of dependability.** Each system development choice must give rise to evidence that, along with an assurance strategy, is sufficient to argue that the assurance goal will be met.
- **Cost.** Clearly, any architectural choice has to be cost effective, not only in terms of software construction effort but in a complete sense. If provision of adequate evidence for the assurance argument would require resources beyond those available, the candidate architectural choice has to be rejected. The issue is not whether the system can be built within budget but whether the system can be built and have a satisfactory assurance case within budget. While cost is a consideration in all systems, regulations sometimes demand justification that risk has been reduced as low as reasonably practical (ALARP). In such systems, cost may directly appear in the assurance case in addition to providing guidance on selection.
- **Feasibility.** The architectural choice must not be infeasible. Moreover, it must not preclude the completion of an architecture that can be instantiated, the completion of a system that is fit for use in its intended context, or the creation of a convincing assurance case for that system.
- **Applicable standards.** Applicable standards have two effects on the selection of an architectural choice. First, a standard might preclude certain choices by definition. Second, standards might require certain development practices that restrict or preclude certain forms of evidence that would otherwise be required for the assurance case.
- **Non-functional requirements.** Non-functional requirements derive from stakeholder interests, and they have an effect that is similar to the effect of a standard. Non-functional requirements often prescribe certain aspects of development or certain characteristics of the desired system. Such prescriptions limit the available architectural choices and are likely to affect the assurance evidence in the same way that a standard can.

As an example of the application of these criteria, consider again the simple autopilot example mentioned earlier. Assume that the choices that the architect can make are: (1) a single processor running the entire application; (2) a pair of processors both running the entire application and comparing outputs; (3) three processors with each running the entire application and voting on their results (TMR); and (4) a distributed implementation in which several processors are connected together with a real-time bus and different parts of the application are run on different nodes.

The evidence for the assurance case that each choice provides would depend on the specific characteristics of the equipment chosen and the planned approach to the development of the software. The dependability requirements from the given architecture are such that options (1) and (2) might have to be rejected based on the dependability criterion. Option (4) might have to be rejected because of cost.

Applying these criteria can be quite involved since they are neither independent of each other nor independent of decisions at other points in development. Consider, for example, the applicable standards criterion. If such a standard prescribes use of a particular programming language, this might preclude the subsequent use of certain forms of static analysis that depend on certain language features (such as strong typing) or on the existence of a formal semantic definition of the language.

### **5.3 Using An Architectural Choice**

Once an architectural choice has been made, precise descriptions (i.e., specifications) of the choice itself and the assurance evidence that implementing it will provide constitute an ABD composite. The ABD composite documents the link between the choice and the evidence.

Once the ABD composite has been formed, the choice is integrated into the evolving system architecture. The architecture can be documented in any manner that is deemed appropriate. In particular, an architectural description language can be employed thereby facilitating a variety of analyses.

The next step is to document the argument fragment to which the evidence applies and to integrate the fragment into the evolving assurance case. As with the architecture itself, the assurance case can be documented in any suitable manner, but the use of GSN would be a likely choice.

The assurance case fragment to be added as the result of a choice identifies the affected development artifacts and describes the contribution that these artifacts will make to the argument. In some cases, the choice will introduce new goals, obligating the developers to supply specific evidence later in the process, while in others the choice will directly support a goal with evidence from a development artifact.

The role of the ABD composite is to document the link between the development stage where the evidence is created and the location in the assurance case where the evidence is part of the argument. As development proceeds, there is an obligation to ensure that the evidence is prepared as expected. Any changes in the anticipated development activity must be traced back to the assurance case so as to check that the effects of the change on assurance have been considered, and the mechanism that supports this traceback is the ABD composite.

It is possible that an architectural choice will prove unacceptable after it has been selected and subsequent choices have been made. To address this, a developer must isolate the problematic choice, select an alternative, and re-examine any choices made after the readdressed choice.

### **5.4 Termination Of The Architectural Development Process**

The ABD process continues as long as architectural choices are being made that produce subgoals which need to be refined using architectural techniques. Thus, as each architectural choice is made, new subgoals will be generated to support the assurance argument fragment that derives from the choice.

Each of these subgoals starts the process described in this section over again unless the architect is convinced that the subgoal is not best addressed by an architectural solution. The architect deems the architecture complete when, in his or her judgement, all of the modules in the system require no further decomposition, are

sufficiently well defined, and are cohesive enough that it is likely that those responsible for designing and implementing them will be able to do so successfully.

An assurance case for a complete architecture may contain unsubstantiated goals that could be addressed through architecture; these will be addressed by design, implementation, and verification choices as the ABD process proceeds. Many architectural patterns are also design patterns, and so it is likely that some goals could be addressed through either architecture or design. Because architecture is more centralized than design, goals that can be addressed through design should be left to designers in order to keep the architecture phase from becoming a bottleneck.

## 6 An Illustrative Example

In order to illustrate the process of developing an architecture using ABD, we present an illustrative example of the use of the technique on a realistic application. The process is quite extensive, and so in the summary we examine only two architectural choices. In addition, although the application is real, we have made a number of assumptions about aspects of the application that either have not been documented by the system developers or are necessary for ABD but not for the application in its present form.

The system we use for illustration is part of a software-based system for alerting pilots to *runway incursions* at airports. Lockheed Martin, in collaboration with the National Aeronautics and Space Administration (NASA), is developing a research prototype system known as the Runway Incursion Prevention System (RIPS) [7, 8] to address this realistic safety-related problem. The Federal Aviation Administration (FAA) defines a runway incursion as “any occurrence at an airport involving an aircraft, vehicle, person, or object on the ground, that creates a collision hazard or results in the loss of separation with an aircraft taking off, intending to take off, landing, or intending to land.” [7]. The RIPS system operates in the cockpit of an aircraft (referred to as *ownship*), collects information about the position of the aircraft and of other aircraft and traffic in the vicinity, examines that information for evidence of a runway incursion involving the ownship aircraft, and alerts the pilot to such incursions via an Integrated Display System (IDS) if a collision is possible.

Our illustrative example is based on a part of RIPS called the Runway Safety Monitor (RSM). The RSM was not developed using ABD, and so our example is strictly for purposes of illustration. Our work is not part of the RIPS development activity. In constructing the example, we have drawn upon the RSM documentation for descriptions of the problem to be solved, the sources of data available for the purpose of detecting incursions, and of the systems on board the aircraft and on the ground with which an incursion detection system might interact.

### 6.1 The Given Architecture

The RSM makes use of the existing systems on board the aircraft including a computer, the aircraft’s ground location system that provides the aircraft’s position, and broadcasts on the Automatic Dependent Surveillance - Broadcast (ADS-B) link that provides the positions of other traffic. These sources of data are known to be unreliable in that data might be unavailable for periods of up to several seconds because of limi-

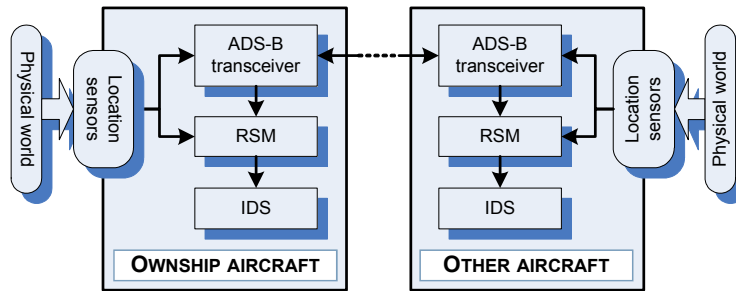


Figure 5. The given RSM architecture

tations in the basic equipment. This lack of reliability in the data is not a serious problem provided the pilot knows that RIPS is not able to report incursions.

While the decision to implement RSM as a software entity that uses this equipment is an architectural decision, it is an architectural decision at the level of RIPS rather than that of the RSM system. In effect, the architects of RIPS decided to delegate the task of alerting the pilot to a software sub-component rather than a separate system running on its own processors. The outcome of these decisions constitute the given architecture.

The given architecture is shown in Figure 5. The IDS system polls the RSM at a frequency of 1 Hz to determine whether a runway incursion involving ownship is in progress. To perform its computation, the RSM will need to know where the ownship aircraft (the aircraft it is installed in) is located, and where other aircraft that might conflict are located. It will obtain the former from the aircraft's ground location system and the latter from the contents of broadcasts on the ADS-B bus.

## 6.2 The Top Level Assurance Goal

The problem to be solved is to detect incursions involving ownship. The top-level goal of our assurance case states both the required functionality and dependability of the system as shown in Figure 6. For purposes of illustration, we have assumed dependability requirements for the RSM that place it in the ultra-dependable category and classify the system as safety critical.

In this example, we assume that the RSM is required to meet the following two requirements (recall that the data sources are unreliable):

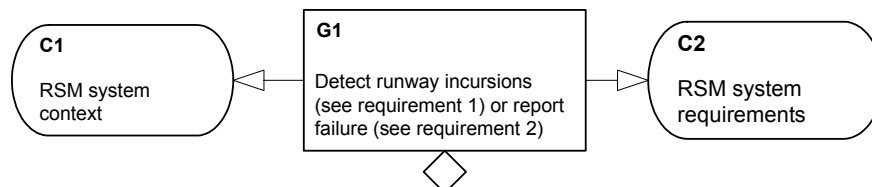


Figure 6. Top-level assurance case goal

- **Requirement 1:** If the quality of the supplied data is adequate, to detect runway incursions involving ownship within  $t$  time units after they begin with probability greater than or equal to  $p_0$ .
- **Requirement 2:** If the quality of the supplied data is inadequate, to report a failure of RSM with probability greater than or equal to  $p_1$  within  $u$  time units.

Note the inclusion in Figure 6 of the system's context in GSN. The details of the system's context are crucial to the proper refinement of the goal and the analysis associated with both the functionality and the dependability of the system.

### 6.3 The First Architectural Choice

There are many candidate architectural choices that meet the two requirements in the top-level goal. For example, the overall approach to the real-time requirements could be either sequential or concurrent, and if concurrent then either synchronous or asynchronous. The choice will be influenced, in part, by the services available from the target operating system, in part, by the anticipated verification approach, and by several other factors.

The requirement for the detection of missing or corrupt data can similarly be addressed using various architectural mechanisms. A number of different system modules could take action when data is missing, and data defects could be signaled by a data collection module by generating an event, by a time-out, or by using special coded data values. Feasibility is an important criterion in this aspect of selection because there has to be a high level of assurance that defective data will be detected and that the timing element of the requirements is met.

The experience of the authors leads us to select a sequential code implementation with each software module responsible for detecting and reporting errors in the data it handles. Choosing sequential code with distributed error detection allows us to divide the top-level goal into three concerns: 1) RSM primary functionality; 2) RSM timing, and 3) RSM detection of defective data. The resulting assurance case fragment is shown in Figure 7. An important item in this fragment is goal G2.4. This goal requires that evidence be supplied and that an argument developed which shows that the three modules do not interfere with each other. This is an important aspect of the verification that must be constructed if this architecture is used, yet this is not obviously so without the assurance case as a reference.

### 6.4 The Second Architectural Choice

The first architectural choice generated four subgoals, and in a complete application of ABD all four would be addressed. For purposes of illustration, we address only one, the RSM functionality (goal G2.1 in Figure 7).

There are many candidate architectures that might be used including several patterns, an object-oriented approach, and functional decomposition. We selected functional decomposition of the RSM functionality (see Figure 8) because it facilitates the use of some forms of static analysis including determination of worst-case execution time. That decision leads in this example to the following six modules:

- The *ownship runway locator*, which determines whether the aircraft in which the RSM is presently using a runway, and, if so, builds a model of that runway;

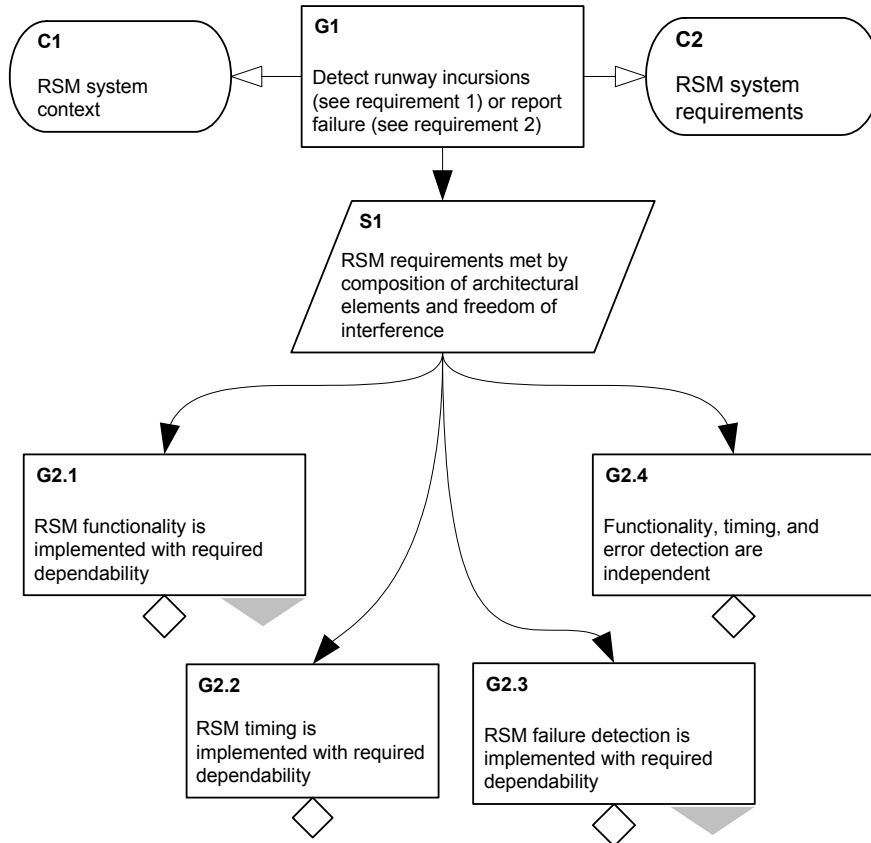


Figure 7. Argument fragment from first architectural choice

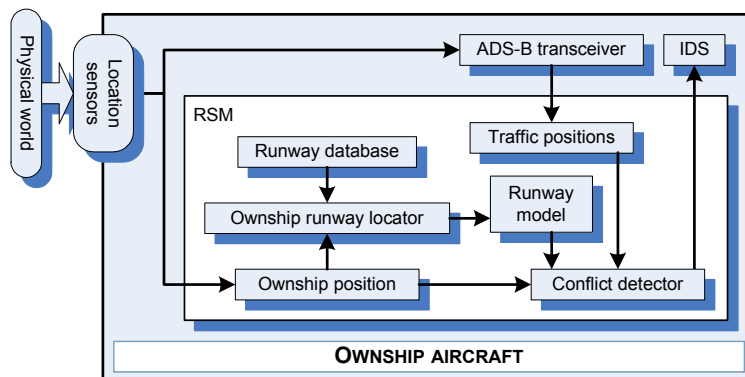


Figure 8. Functional decomposition of RSM functionality

- The *runway database*, which stores the location and necessary geometric details of all of the runways for which RSM service will be available;
- The *runway model*, which stores the geometry of the runway including the bounds of the incursion zone;
- The *ownership position* component, which collects information about the position of the aircraft from the aircraft's ground location system;
- The *conflict detector*, invoked if the aircraft is found to be using a runway, determines whether ownership is in conflict with any other monitored traffic within that runway's incursion zone, and
- The *traffic positions* component, which collects information about the position of other traffic within a specified region from ADS-B broadcasts.

The assurance case fragment that accompanies this architectural choice is shown in Figure 9. It details the assurance responsibility allocated to each of the new components listed above and how these responsibilities, if satisfied, demonstrate the satisfaction of sub-goal G2.1.

Although not shown, the detailed arguments for goals G2.2 and G2.3 in Figure 7 are similar to the argument for goal G2.1. The argument for goal G2.2 is facilitated by the decision to use functional decomposition for goal G2.1. To show that goal G2.2 is met requires several forms of evidence, including assurance that various modules will execute within specified time bounds. One of those modules is that which is associated with goal G2.1. Functional decomposition as the architectural choice for goal G2.1 eases the task of determining worst-case execution time (WCET) for that module. WCET is not easy to establish with any architecture and can be essentially impossible with some modern processors. However, assurance over timing is essential, and that makes many other candidate architectural choices unacceptable.

The argument for goal G2.4 will be different from goals G2.1, G2.2 and G2.3. This goal is concerned with the composition of the evidence from the other three. It is not sufficient to know that each of the other three goals will be met in order to use that evidence to argue that goal G1 will be met. Goal G2.1 might be met, for example but there might be side effects that impact the composition of goals G2.1, G2.2, and G2.3.

Turning now to the other selection criteria, we ask ourselves whether, given this architectural choice, it is likely that the system can be built within the specified budget, schedule, technology constraints, etc. At this point the architecture is not yet complete, much less the low-level design and implementation, so our assessment will be speculative—as would any such assessment at this point in the development of a system. Given our experience and knowledge and the proposed architecture as it stands, how likely do we think it is that we will encounter a difficulty that will force revisions that would cause the project's schedule to slip or, worse, cause the effort to fail completely?

To perform this assessment, we consider the components in our candidate architecture and the responsibilities upon them described in our assurance case fragment. Will it be possible, for example, to construct the ownership runway locator so that, provided the components it depends upon perform as described, the ownership runway locator demonstrably satisfies the goals with which it is associated? Given what we know about the probability of data errors, reasonableness checks on the incoming data cou-

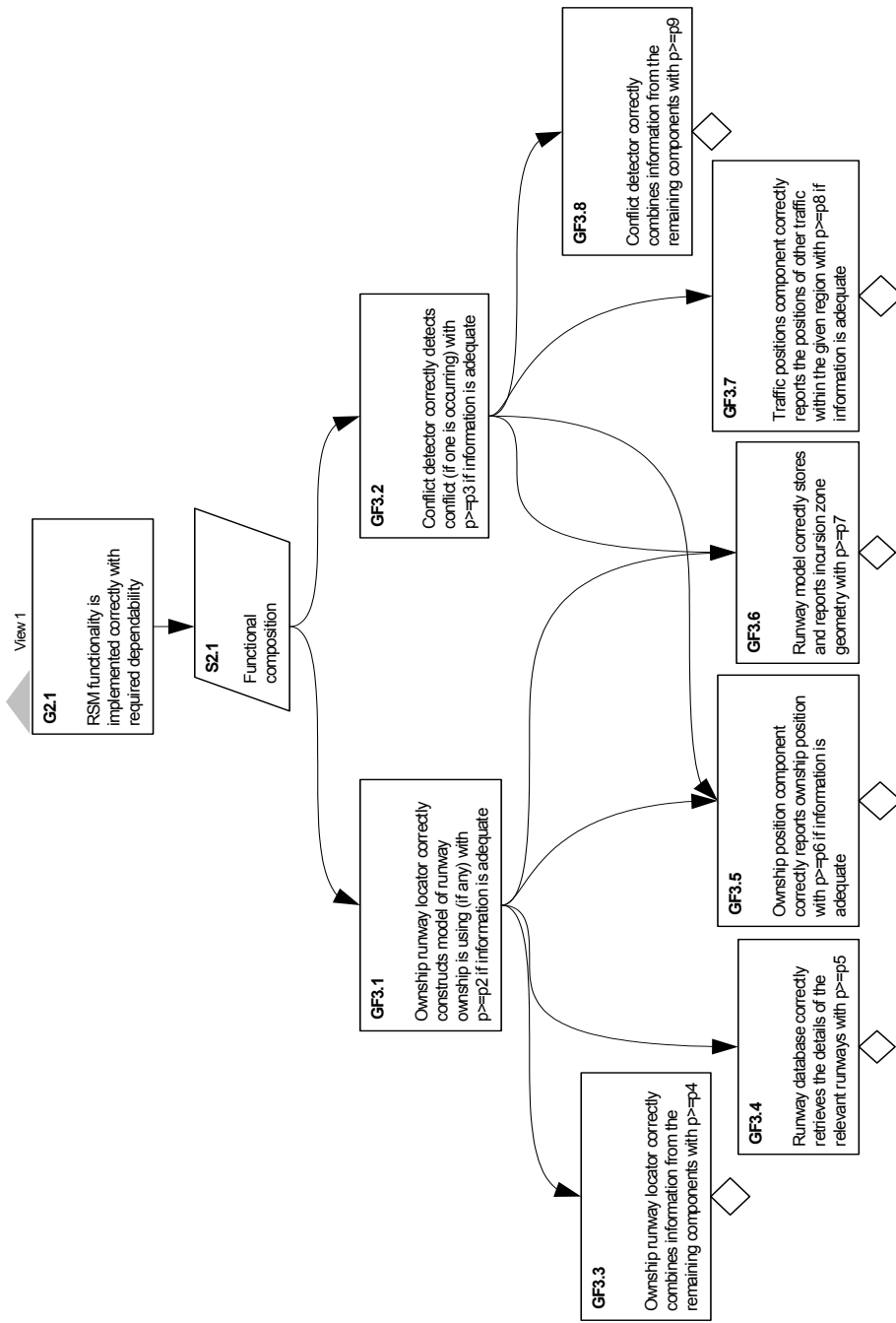


Figure 9. Assurance of functionality split across functional decomposition

pled with the use of formal techniques to implement and verify the algorithm seem like a plausible way to construct a component, later in the process, that can be shown to meet its goals using a software reliability modeling technique. Thus, we decide that the architectural choices we made are appropriate given the knowledge we have of the system at the time the architecture is created. Furthermore, although there is no way to know for sure that the architecture is the best one, we were able to assess it against the system's assurance goals, and that assessment gives us much higher confidence that the architecture is satisfactory than would experience and intuition alone.

## **7 Conclusion**

There are a number of choices that must be made when designing a system's architecture, and those choices can have a profound impact on the finished system's dependability. Currently, there is little guidance for making the right choices, given the level of dependability that must be met by the system. If system development is coupled with system assurance, however, the system's assurance case can guide architectural choices, providing concrete dependability criteria against which to gauge potential alternatives.

In this paper, we have explained the basic principles of Assurance Based Development, and shown how this development paradigm can be used to provide assurance case goals for architectural choices. We have presented an example system architecture and shown how evolving its assurance case in parallel with the architecture kept us continuously apprised of the specific dependability goals each part of our system was obliged to meet. Whereas with standard architecture development techniques we would have had to wait until system development was more complete to analyze the effect of our choices on the system's dependability, we were able to assess the choices against specific assurance case goals for the RSM.

Finally, software system architecture is currently very much an art, and the creativity in finding a good architecture is due in large part to the difficulty in creating general guidelines from a wide variety of systems. Because it is not clear whether the context in which one successful decision was made is similar to that in which a new choice must be made, whether the same choice should be made for the new system is likewise unclear. With ABD, the specific situation in which a choice is made is much more clearly defined because of the assurance goal that accompanies it. Thus, not only does ABD guide specific architectural choices, it helps lay a foundation for good architectural engineering.

## **Acknowledgements**

We thank David Green of Lockheed Martin for giving us extensive help in understanding the RSM and all of the associated artifacts. We are very grateful to NASA Langley Research Center for suggesting the use of the system for study. We appreciate William Greenwell's assistance with the assurance case material presented here and his contribution of the hypothetical safety case in Figure 1. This work was sponsored in part by NSF grant CCR-0205447 and in part by NASA grant NAG1-02103.

## References

- [1] ANSI/IEEE standard 1471-2000, "Recommended Practice for Architectural Description of Software-Intensive Systems -Description".
- [2] Bishop, Peter and Robin Bloomfield. "A Methodology for Safety Case Development." Proc. of the Sixth Safety-critical Systems Symposium, Birmingham, Feb. 1998. <<http://www.adelard.co.uk/resources/papers/index.htm>>
- [3] de Lemos, R., C. Gacek, A. Romanovsky (Eds.). Architecting Dependable Systems. Lecture Notes in Computer Science 2677. Springer 2003.
- [4] de Lemos, R., C. Gacek, A. Romanovsky (Eds.). Architecting Dependable Systems II. Lecture Notes in Computer Science 3069. Springer 2004.
- [5] de Lemos, R., C. Gacek, A. Romanovsky (Eds.). Architecting Dependable Systems III. Lecture Notes in Computer Science 3549. Springer 2005.
- [6] EUROCONTROL. "The EUR RVSM Pre-Implementation Safety Case," ver. 2.0. Document RVSM 691. 14 August 2001.
- [7] Green, D. F. "Runway Safety Monitor Algorithm for Runway Incursion Detection and Alerting." Technical report NASA CR-2002-211416. January 2002.
- [8] Green, D. F. "Runway Safety Monitor Algorithm for Single and Crossing Runway Incursion Detection and Alerting." Technical report NASA CR-2006-214275. February 2006.
- [9] Kelly, T. P. "A Systematic Approach to Safety Case Management." Proc. of SAE 2004 World Congress, Detroit, MI, March 2004.
- [10] Kelly, Tim, and J. McDermid. "Safety Case Patterns – Reusing Successful Arguments." Proc. of IEE Colloquium on Understanding Patterns and Their Application to System Engineering, London, Apr. 1998.
- [11] Kinnersly, S. "Whole Airspace ATM Safety Case - Preliminary Study." November 2001.
- [12] MoD, "00-56 Safety Management Requirements for Defence Systems," U.K. Ministry of Defence, Defence Standard, Issue 3, December 2004.
- [13] Nagra. "Project Opalinus Clay: Safety Report." Technical report NTB 02-05. December 2002.
- [14] RTCA. "Software Considerations in Airborne Systems and Equipment Certification," document RTCA/DO-178B. Washington, DC: RTCA, December 1992.
- [15] Shaw, M., and D. Garlan. Software Architecture: Perspectives On An Emerging Discipline. Prentice Hall, 1996.
- [16] Strunk, E. A. and J. C. Knight. "Dependability Through Assured Reconfiguration in Embedded System Software.", IEEE Transactions on Dependable and Secure Computing, Vol. 3, No. 3, pp 172-187 (July 2006)
- [17] Weaver, R. A. and T. P. Kelly. "The Goal Structuring Notation - A Safety Argument Notation." Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases, July 2004. <<http://www-users.cs.york.ac.uk/~tpk/dsn2004.pdf>>
- [18] Wojcik, R., F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. "Attribute-Driven Design (ADD), Version 2.0." Technical report CMU/SEI-2006-TR-023. November 2006.