Assurance Based Development of Critical Systems

Patrick J. Graydon John C. Knight
Department of Computer Science
University of Virginia
{graydon | knight}@cs.virginia.edu

Elisabeth A. Strunk
Software Systems Engineering Dept.
The Aerospace Corporation
elisabeth.a.strunk@aero.org

Abstract

Assurance Based Development (ABD) is the synergistic construction of a critical computing system and an assurance case that sets out the dependability claims for the system and argues that the available evidence justifies those claims. Co-developing the system and its assurance case helps software developers to make technology choices that address the specific dependability goal of each component. This approach gives developers: (1) confidence that the technologies selected will support the system's dependability goal and (2) flexibility to deploy expensive technology, such as formal verification, only on components whose assurance needs demand it. ABD simplifies the detection—and thereby avoidance—of potential assurance difficulties as they arise, rather than after development is complete. In this paper, we present ABD together with a case study of its use.

1. Introduction

Assurance Based Development (ABD) [9] is a novel approach to constructing critical computing systems. In this paper we discuss how it can help developers make development decisions that contribute to their systems' dependability arguments. In ABD, dependability, i.e., confidence that the system will meet its dependability goals when fielded, is evaluated throughout the development process. The system and its assurance argument are co-developed so that explicit criteria for the dependability impact of a development choice are available at the time the choice is made. Combining development and assurance in this way facilitates detection and avoidance of potential assurance difficulties *as they arise*, rather than after development is complete—when they are much harder to address. Here, we explain the kind of assurance goals that are generated at each step and how development decisions can be evaluated against those goals.

Knowing that a critical system is going to operate dependably in its expected environment is essential, yet current approaches to dependability assurance are ad hoc. There are many techniques available to developers of critical systems, but in most cases their benefits have been shown only in isolation and developers do not fully exploit the dependability benefits they bring. For example, developers might use formal methods for the "critical parts of the system," but are often unable to evaluate the ensuing effect on the dependability of the system as a whole. There is little incentive to use a technique whose overall benefit to the delivered system is unknown.

Our goal with ABD is to integrate the development of the system and its assurance argument, thus enabling assurance needs to drive development decisions. We avoid making assumptions about the structure of the system's functional documentation (since that varies widely from system to system), but for the system's assurance argument we use an *assurance case*. The assurance case is an argument that sets out the dependability goal for the system, the evidence needed to support it, and how that evidence is used to justify the claim. Safety cases, a specific form of assurance cases, are used widely in Europe; in some domains, their use is mandated by regulation.

While assurance cases are often created for critical systems, they are not always exploited to guide developers' choices. This is: (1) inefficient, because development steps might produce superfluous assurance or need to be revisited after development is complete; and (2) ineffective, because necessary development activities were omitted unintentionally. If the assurance case is produced near the end of development, there is a risk that the evidence produced during development will be insufficient. This, in turn, could force developers to produce additional evidence or pressure them into accepting an incomplete argument [4]. Recognizing these risks, some standards [7] and researchers [4] call for developers to construct safety cases early and

update them often. ABD takes this idea to its fullest extent, integrating safety case and system construction.

ABD combines assurance case and system development so that each supports the other. System development is tailored to support the assurance case, generating evidence for the argument at each step. The assurance case also supports the development process, guiding developers to make choices that make the system much more likely to be fit for its intended use.

We begin in section 2 by summarizing the assurance case concept and how it fits into ABD. We describe the ABD process itself in section 3, and we illustrate its use in section 4. We discuss related work in section 5 and present our conclusions in section 6.

2. Assurance cases and ABD

Assurance cases are the state of the art in rigorous but informal dependability argumentation and, as such, provide the foundation on which the ABD approach rests. Safety cases, a special form of assurance cases, have been built to document the *safety* of 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" [1]. Graphic notations have been designed to facilitate writing 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 is the Goal Structuring Notation (GSN) [10].

In its simplest form, an assurance case contains 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 evidence justifies the goal. This basic structure is applied recursively to produce, for real systems, a hierarchic structure with the overall goal for the system at the root. Evidence at one level becomes a goal at the next lower level, so that the argument is manageable at each level. Other elements that can appear in assurance cases are strategies, assumptions, justifications, and context. While the goals, etc., and hence the assurance argument, are specific to a particular system, patterns for common argument fragments have been developed [5].

In ABD, the assurance case and the system are co-developed, with each affecting decisions in the other. The *ABD process*, which determines how this synergistic activity is to be performed, is discussed in section 3.

Because the system and its assurance case are co-developed, the links between the assurance argument and the development artifacts that support each part of it must be documented. These *assurance links*, together with the development artifacts and assurance case fragments being linked, form an *ABD composite*. Figure 1 shows a fragment of an example ABD composite in which the assurance case is presented in GSN. In the figure, solution S1 shows evidence from a development artifact being applied to support goal G3, and, indirectly, to support the top-level assurance goal, G1.

In ABD, assurance case goals represent assurance obligations that must be addressed, directly or indirectly, by evidence from development artifacts. Goals are added to the assurance case diagram with an annotation indicating that they have not yet been addressed. As development progresses, these goals are addressed either by direct evidence from a development artifact or an argument fragment combining evidence from subgoals, each of which must itself be addressed.

The assurance case goals related to a development artifact set out the properties which the developer must ensure that the artifact possesses. Goal G2 in Figure 1, for example, could describe a property that a development artifact is to have. If each artifact is created so that it has the required properties and the assurance argument is valid, then the system developers can be confident that the system will meet its assurance goals.

As an example of a fragment of an ABD composite and assurance links, consider the goal of developing a software component in a safety-critical system. If the component must meet an assurance goal of having a failure rate per unit time below 10^{-3}, i.e., not in the ultra-dependable range, testing might be the basic technology chosen to meet this goal. Such a goal requires that the testing process provide several pieces of evi-

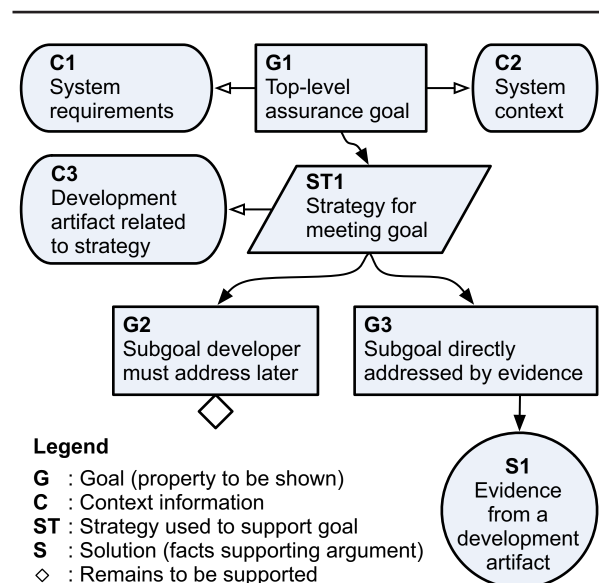

Figure 1. A fragment of an ABD composite

dence to support the assurance case. These pieces of evidence are: (1) that the documented test plan has been conducted as prescribed; (2) that the component's failure rate has been demonstrated in a statistically valid way to be below the threshold; and (3) that the test cases were the result of a random process of selection from the expected operational environment.

3. Assurance Based Development

The ABD process addresses assurance case goals through development choices, which provide evidence that is linked into the assurance case. Briefly, a developer applying ABD repeatedly examines the unsatisfied goal(s) in the assurance case, selects one, makes a *system development choice*—the selection of an architecture or design, the use of a particular tool or language, an implementation decision, or the selection of a verification or validation strategy—that addresses the goal, and modifies the development artifacts and assurance case accordingly. Because each choice may give rise to several new, unaddressed, goals, the process is applied repeatedly until no unaddressed goals remain.

ABD assumes the availability of system requirements, including functional requirements and dependability requirements such as availability and safety. In this paper, we assume that the requirements are correct and complete. We plan to consider the integration of requirements techniques with ABD in future work.

We also assume the availability of a description of the *given architecture*, i.e. the high-level architecture within which the computing system will operate. For example, a new anti-lock braking system is part of a larger automobile system that relies upon the braking system meeting certain functional, non-functional, and dependability requirements. To the braking system developer, the given architecture includes the braking system itself, the other vehicle systems with which the brakes must interact, and the interfaces between the braking system and the other components. The given architecture for a desktop application would include the operating system and the desktop's peripherals.

In section 3.1 we describe how the developer enumerates potential system development choices. In section 3.2 we describe how the developer ought to select from among the available choices. Finally, in section 3.3 we describe how this choice is recorded in the assurance case as an ABD composite.

3.1. Candidate development choices

It is important that a developer enumerating candidate system development choices cast a net wide enough to include at least one choice likely to lead to a system that meets its functional, cost, dependability, and other goals. Developers will consider familiar choices and may solicit suggestions from colleagues and team members, but these sources alone may be insufficient. Furthermore, while considering more choices increases the likelihood that at least one will be suitable, there are costs associated both with enumerating potential choices (as consulting reference material takes time) and with assessing each candidate choice. Developers need a way to enumerate a short list of choices likely to be acceptable for a given problem.

Patterns are a general and commonly used technique that has proven especially important in architecture and design. Experience in ABD could be captured by patterns of coupled system development choices and assurance case argument fragments, helping developers to quickly enumerate a set of candidate choices appropriate to a given problem. We leave effective recording and retrieval of patterns for future work.

3.2. Selection of a system development choice

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

Much of the pruning of the set will be based on the developer's experience. In many cases, an experienced developer might consider only a single candidate system development 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. Note 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.*** The system development choice must not obviously preclude achieving the desired functionality. This can be checked by inspection, analysis, prototyping and/or modeling.
- ***Restrictions on later choices.*** Each system development choice that is made affects the subgoals that are generated and thus restricts the available choices throughout the remainder of development. The system development choice should not preclude desirable choices later, particularly when the later choices support dependability.
- ***Evidence of dependability.*** Each system development choice must give rise to evidence that, along

along with an assurance strategy, is sufficient to argue that the assurance goal will be met.

- ***Cost.*** The system development choice must be cost-effective in a complete sense; that is, it must be possible to build both the system and a satisfactory assurance case within budget. If providing adequate evidence for the assurance argument would require resources beyond those available, a candidate choice must be rejected.
- ***Feasibility.*** The system development choice must not itself be infeasible or preclude completion of the system or a convincing system assurance case.
- ***Applicable standards.*** Applicable standards can: (1) preclude certain choices by definition; or (2) 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 often prescribe certain aspects of development or certain characteristics of the desired system. Such prescriptions limit the available system development 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 anti-lock braking system example mentioned earlier. Assume that the braking system's computations could be run on: (1) a single processor; (2) two processors whose outputs are compared; (3) three processors whose outputs will be voted on (TMR); or (4) many processors on a real-time bus, each running part of the computation.

The assurance case evidence that each choice provides would depend on the specific characteristics of the equipment chosen and the planned software development approach. If the dependability obligations of the hardware are stringent enough, options (1) and (2) must be rejected based on the dependability criterion. Option (4) would have to be rejected because of cost.

Applying these criteria can be quite involved since they depend both on each other and on decisions at other points in development. Consider, for example, the *applicable standards* criterion. If a relevant standard prescribes the use of a particular programming language, this might preclude the subsequent use of static analysis that depends on certain language features (such as strong typing) or on the existence of a formal semantic definition of the language.

3.3. Applying system development choices

Once made, a development choice is applied to the system and the assurance case updated to reflect its effect. The way in which the choice is applied to the development artifacts will depend upon the type of artifact. An architectural choice, for example, might be applied by modifying a description of the system's architecture in an architectural description language. The choice to use a particular programming language might be recorded in project standards documentation.

The assurance case extension resulting from 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. In section 4 we will illustrate this linking with a more concrete example.

4. An illustrative example

To illustrate the process of developing a system using ABD, we present a summary of the use of the process on a realistic application. Space considerations preclude us from describing every system development choice, and so we examine only a subset of them. We have selected a depth-first slice of the assurance argument so as to illustrate artifacts from most development phases: we illustrate the development decisions, evidence, and argument from the requirements level down to source code. 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 application we use for illustration is part of a research prototype for a software-based system for alerting pilots to *runway incursions* at airports. 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." [2]. The system, known as the Runway Incursion Prevention System (RIPS) [2, 3], is being developed by Lockheed Martin on a contract from the National Aeronautics and Space Administration (NASA).

The RIPS system operates in the cockpit of an aircraft (referred to as *ownship*). It collects information about the position of that aircraft and of other aircraft and ground 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).

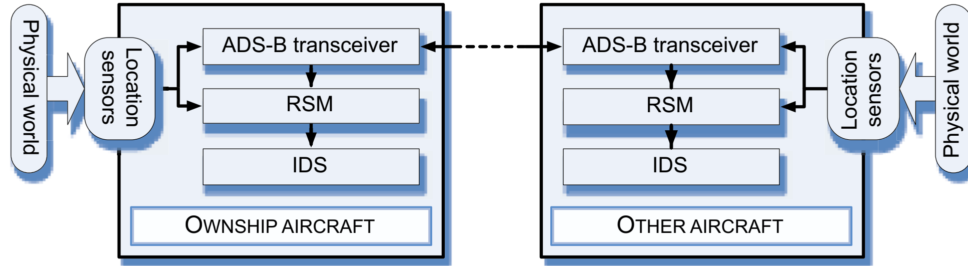

Figure 2. The given RSM architecture

Our illustrative example is based on a part of RIPS called the Runway Safety Monitor (RSM). Our work is not part of the RIPS development activity, and ABD was not used to develop the RSM. Our example is strictly for purposes of illustration. In constructing it, we have drawn upon the RSM documentation for: (1) the problem to be solved; (2) the sources of data available for the purpose of detecting incursions; and (3) the systems on board the aircraft and on the ground with which an incursion detection system might interact.

4.1. The given architecture

The RSM uses 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 aircraft. Limitations in the basic equipment may make these data unavailable for up to several seconds. This lack of reliability is not a serious problem provided the pilot knows that RIPS is inoperative.

The decision to implement RSM in software is an architectural decision at the level of the RIPS system: 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 result of this and other decisions by the RIPS team is the RSM's given architecture, shown in Figure 2. 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 ownship is located, and where other aircraft that might conflict are located.

4.2. The top level assurance goal

The top-level goal in Figure 3, G1, states the required functionality and dependability of the RSM. For purposes of illustration, we have assumed dependability requirements 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):

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

Note the inclusion in Figure 3 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.

4.3. The first system development choice

There are many candidate 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 by the services available from the target operating system and the anticipated verification approach amongst many 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 within the specified time limit.

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. While this choice is somewhat arbitrary, we note that in a complete application of ABD developers will balance the perceived risk of

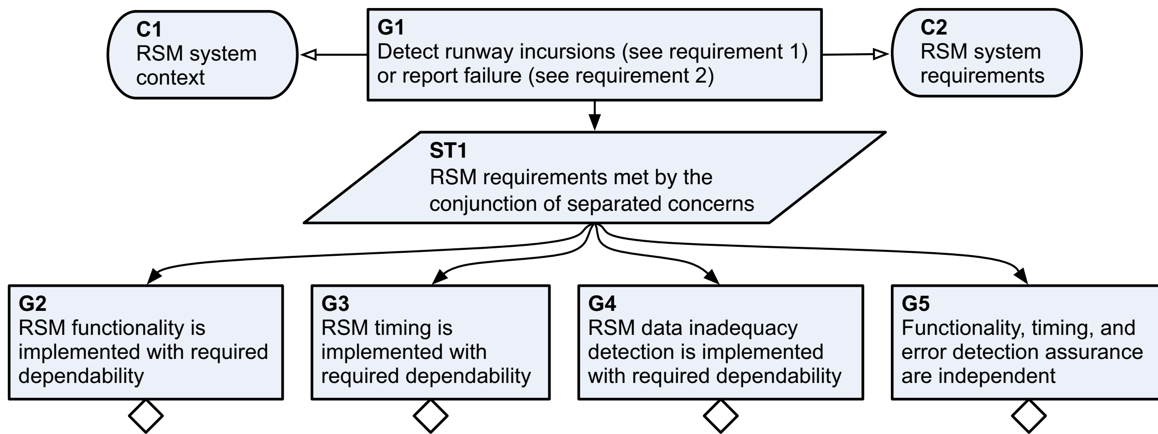

Figure 3. Assurance case after first choice

making (and thus having to redress) a poor choice against the time required to more thoroughly enumerate and evaluate alternatives.

Choosing sequential code with distributed error detection allows us to divide the top-level goal into three concerns and address each of these independently as shown in Figure 3. An important item in this fragment is goal G5. Without careful attention, developers might supply arguments for the satisfaction of goals G2, G3, and G4 that do not, together, imply the satisfaction of goal G1. If, for example, testing is used to show low probabilities that the RSM will fail to provide the required functionality (goal G2) or fail to meet its timing requirements (goal G3), then the selection of test cases will affect the way in which the probabilities are combined to justify meeting requirement 1 in goal G1. Goal G5 obliges the developer to carefully consider whether the manner in which the other goals are satisfied justifies concluding that G1 has been satisfied, and so acts as a reminder to the developer to consider an important but subtle consideration that could easily be missed if the assurance case were not used as guidance.

4.4. The second system development choice

The first choice generated four subgoals, each initially unaddressed as indicated by the diamond-shaped decoration. In a complete application of ABD all four would be addressed in an order chosen to minimize the risk of needing to readdress a choice. For purposes of illustration, we continue our example by addressing RSM failure detection (goal G4 in Figure 3).

There are many available candidate choices, including several architectural patterns, an object-oriented architecture, and functional decomposition. We selected functional decomposition because it facilitates the use of some forms of static analysis including determination of worst-case execution time. Our decomposition, recorded in architectural diagram form in Figure 4, contains the following six modules:

- the *ownship runway locator*, which determines whether ownship is presently using a runway, and, if so, builds a model of that runway;
- the *runway database*, which stores the location and necessary geometric details of all of the runways for which RSM service will be available;

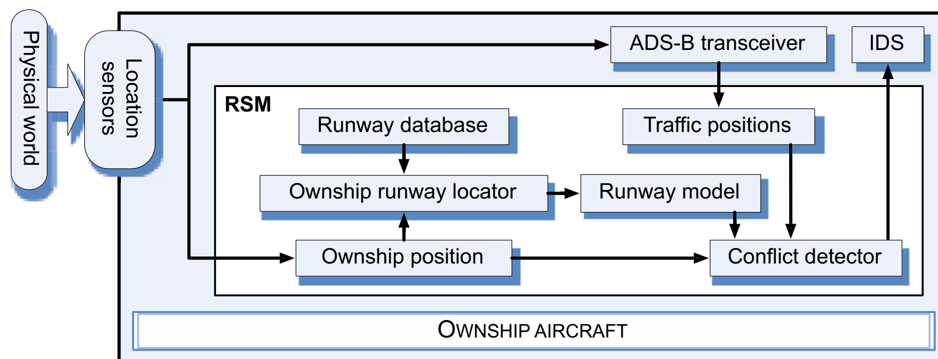

Figure 4. Functional decomposition of the RSM

- the *runway model*, which stores the geometry of the runway including the incursion zone's bounds;
- the *ownship position* component, which collects information about the position of the aircraft from the aircraft's ground location system;
- the *conflict detector*, which is invoked if the aircraft is found to be using a runway, and determines whether ownship 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.

Part of the assurance case fragment that accompanies this system development choice is shown in Figure 5. It details the failure detection responsibility allocated to each of the new components listed above and how these responsibilities, if satisfied, demonstrate the satisfaction of sub-goal G4. Note the context bubble C3, which clarifies strategy ST2 by describing the functional decomposition we have selected. This clarification has the effect of linking the assurance-case fragment shown in Figure 5 to the development artifact in which the choice was recorded.

Although not shown, the arguments for goals G2 and G3 in Figure 3 are similar to that for goal G4. The argument for goal G2 would show how responsibility for the RSM's functionality is partitioned across the system's modules, and the argument for goal G3 would show how the system meets its timing requirements by linking several forms of evidence, including evidence showing that various modules will execute within set time bounds. Functional decomposition as the system development choice for goals G2, G3, and G4 eases the task of determining worst-case execution time (WCET) for the system. 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.

Turning now to the other selection criteria, we ask ourselves whether, given this choice, it is likely that the system can be built within the specified budget, schedule, technology constraints, etc. Since not even the architecture is yet complete, our assessment must be speculative. Given our experience, knowledge, and the system as proposed, it seems likely that we will be able to find acceptable ways to satisfy the as-yet unaddressed subgoals, and so we accept this choice for now.

4.5. The third system development choice

The application of the assurance case fragments associated with our choice of functional decomposition has, at this point, satisfied goals G2, G3, and G4 from Figure 3, thus removing their diamond-shaped decorations. Goal G5 and the new subgoals introduced by our choice remain unsatisfied. We continue our illustration

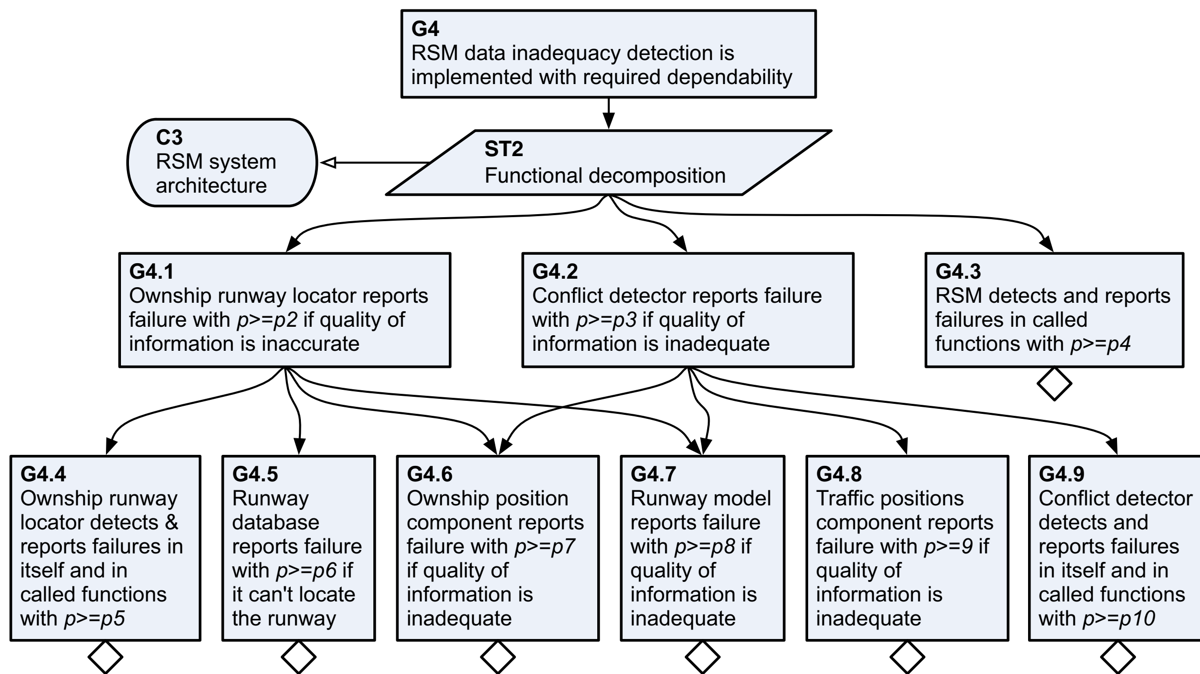

Figure 5. Assurance case fragment from second choice

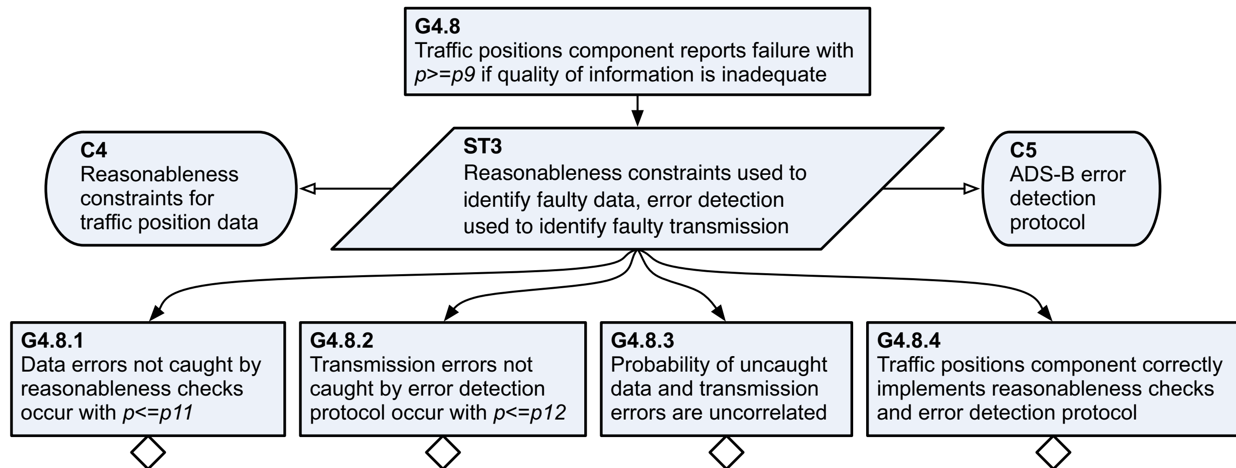

Figure 6. Assurance case fragment from third choice

by addressing subgoal G4.8. The traffic positions component must detect the cases in which the information received over ADS-B from the other aircraft is inadequate and in those cases report failure with probability at least p_9. Traffic position information could be compromised in several ways: 1) the other aircraft could report data incorrectly, 2) the data could be corrupted in transit, and 3) the data could be stale because up-to-date information was not received in time. The developer must ensure that the probabilities of these events occurring undetected, either alone or in combination, is low enough that the target probability is met.

Let us assume that the target probability is high, and that the error detection and correction mechanism used on ADS-B broadcasts is sufficient for our purposes. The design of the traffic positions component must sufficiently mask the remaining types of faults.

One way to address the threat of incorrect information being reported by other aircraft is to impose reasonableness criteria on the data. There are limits to the acceleration of aircraft, and so if the data representation of an aircraft's positions changes too quickly, we can conclude that the data are faulty and report the unavailability of traffic position information.

Alternatively, we could incorporate a redundant source of traffic position information such as radar or a camera with which to compare the ADS-B data (as is done in the actual system). Since the given architecture does not include such an additional information source, however, we have little choice but to detect faults in the information we have. Accordingly, we choose to:

- track the traffic positions we obtain from ADS-B broadcasts over time, computing the velocity and acceleration of each target aircraft;
- perform reasonableness assessment on these to detect incorrect information in the broadcast;
- check the error detection fields of each ADS-B message we use to detect corrupt messages; and
- use position estimation to address the threat of stale data.

An assessment of this choice will help us to decide whether reasonableness constraints are likely to be sufficient. Figure 6 shows the argument fragment that accompanies the choice to use reasonableness criteria. If we can satisfy subgoal G4.8.1 by showing that the probability of incorrect data not being caught by our criteria is sufficiently low, then we can make the choice and proceed; if not, we must seek a change to our previous choices or to our project's givens that would make the project tractable.

4.6. The fourth system development choice

Assuming that we find—or chose to assume, at the risk of rework or project failure if we are wrong—that it is possible to satisfy subgoal G4.8.1, we proceed by seeking ways to satisfy the remaining subgoals. One such subgoal is G4.8.4, which claims that the implementation of the traffic positions component performs the described computation.

If our architecture and design were sufficient to tolerate a small number of residual faults in the traffic positions component, we could elect to implement the algorithm in the programming language of our choice and rely upon testing to confirm that our implementation has the desired behavior. If we had elected earlier to use a safety kernel [11] in our architecture, for example, we might now be faced with such a goal. We assume to the contrary that the needed assurance

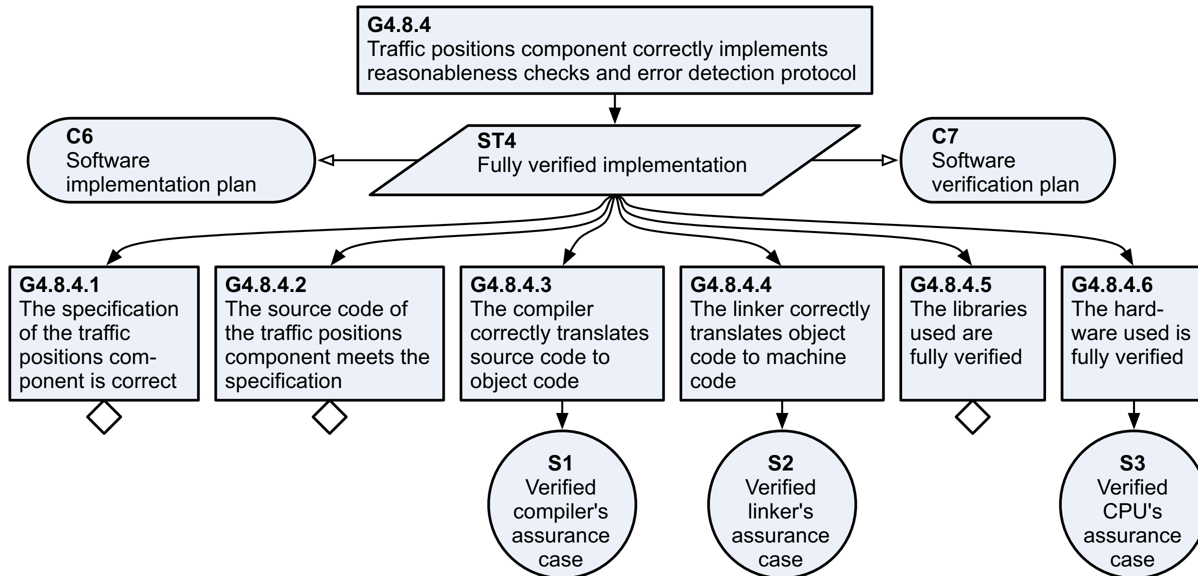

Figure 7. Assurance case fragment from fourth choice

makes testing infeasible, and so we must either seek an alternative choice or else revise an earlier choice so as to lower the target probability.

One choice that would allow us to meet this goal is to use a fully-verified implementation of the traffic position component, resulting in the assurance case fragment shown in Figure 7. If we make this choice, we must write a formal specification of the chosen algorithm, implement it, and argue that:

- the specification correctly formalizes the described computation;
- the source code we write meets the specification;
- the compiler we use correctly translates the source code into object code;
- the linker we use correctly translates the object code into machine code;
- the libraries we use (if any) are fully verified; and
- the hardware we run on correctly implements its instruction set architecture.

Some of these sub-goals can be addressed with direct evidence. The assurance case provided by the vendors of the tools we use, for example, constitute evidence in this assurance case of our claim that these tools have been fully verified.

4.7. Re-addressing a choice

At this point in our example we have chosen an architecture, selected a design for detecting and reporting errors in data from an unreliable subsystem, and chosen an implementation strategy for one submodule. We may use this strategy for other submodules with similar assurance needs, potentially saving costs by re-using the same technique. We are not compelled to do so, however: if a cheaper technique meets another module's assurance needs, we are free to use it.

Several unaddressed goals remain, and in a complete application of ABD, we would continue the process until all sub-goals had been addressed, directly or through argument, by evidence. We would then examine the assurance case for flaws in the argument, and redress these flaws. (The techniques a developer would use to validate the assurance case argument are beyond the scope of this work.)

At any point in the process, a developer may discover that a previous choice leads to an unsatisfiable goal and so requires redress. If, for example, we could find no suitable fully-verified compiler, our fourth choice would need readdressing. If no other implementation strategy could be found to address goal G4.8.4 in Figure 6, we may even have to readdress the architecture to weaken goal G4.8.4 by reducing the dependability obligation imposed on the implementation.

5. Related work

Some standards governing software development, such as DO-178B [8], offer the same prescription to all applicable systems. The ABD process instead compels the developer to assess the dependability needs of each part of a system and make system development choices accordingly; the developer can thus economize in some parts of the system while remaining assured that the system as a whole will be adequately dependable.

Other safety-critical software development work is assessed via a safety case. Some standards [7] and researchers [4] call for safety cases to be constructed early and updated often during system development and subsequent change. Continuing with this idea, ABD interleaves development and assurance case construction tightly so that the assurance case can be used to drive system development.

Problem-Oriented Software Engineering [6] aims to create software and an argument that it is fit for use. In POSE, the system requirements are documented using Problem Frames and progressively transformed, via a series of justified transformations, into an implementable specification. This is one technique that could be used to argue (as one must) that if the system meets its requirements the user will be satisfied.

6. Conclusion

Many choices must be made at each stage of system development, and these can profoundly impact the finished system's dependability. Currently, there is little guidance for making choices that give assurance that the system meets it dependability goals. If the system and its assurance case are co-developed, however, the assurance case can provide 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 system development choices. We have presented an example system and shown how its assurance case drove specific choices we made in its development by allowing us to analyze the effect of our choices on the system's dependability as we made them. While it is not possible to show that the choices we have made are optimal because of the many variables involved in development, we feel that our choices were better informed than they would have been otherwise and so are more likely to achieve our goals.

Acknowledgements

We thank David Green of Lockheed Martin for extensive help in understanding the RSM and its 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. This work was sponsored in part by NSF grant CCR-0205447 and in part by NASA grant NAG1-02103.

References

[1] Bishop, P. and R. Bloomfield. "A Methodology for Safety Case Development." <http://www.adelard.co.uk/resources/papers/index.htm>

[2] Green, D. F. "Runway Safety Monitor Algorithm for Runway Incursion Detection and Alerting." Technical report NASA CR-2002-211416. January 2002.

[3] Green, D. F. "Runway Safety Monitor Algorithm for Single and Crossing Runway Incursion Detection and Alerting." Technical report NASA CR-2006-214275. February 2006.

[4] Kelly, T. P. "A Systematic Approach to Safety Case Management." Proc. of SAE 2004 World Congress, Detroit, MI, March 2004.

[5] Kelly, T., and J. McDermid. "Safety Case Patterns – Reusing Successful Arguments." Proc. of IEE Colloquium on Understanding Patterns and Their Application to System Engineering, London, April 1998.

[6] Mannering, D., J. G. Hall, and L. Rapanotti. "Relating Safety Requirement and System Design through Problem Oriented Software Engineering." Technical report 2006/11, Open University, September 2006.

[7] MoD, "00-56 Safety Management Requirements for Defence Systems," U.K. Ministry of Defence, Defence Standard, Issue 3, December 2004.

[8] RTCA. "Software Considerations in Airborne Systems and Equipment Certification," document RTCA/DO-178B. Washington, DC: RTCA, December 1992.

[9] Strunk, E. and J. Knight. "The Essential Synthesis of Problem Frames and Assurance Cases." Proc. of 2nd International Workshop on Applications and Advances in Problem Frames, co-located with 29th International Conference on Software Engineering, Shanghai, May 2006.

[10] 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.

[11] Wika, K.G. and J.C. Knight. "On the Enforcement of Software Safety Policies." Proc. of 10th Annual IEEE Conference on Computer Assurance (COMPASS '95), Gaithersburg, MD, June 1995.