Abstract
Problem frames and assurance cases are two current research areas that can improve—and have improved—system
dependability, in critical and noncritical systems alike. While these two techniques are effective separately, their syn-
thesis is much more powerful. This paper describes the synthesis of these two techniques and the rationale behind the
synthesis, the particular pieces that influence each other, and the beginning of a process to integrate the two in soft-
ware system development. A detailed example of the application of the synthesis is also provided.

Keywords and phrases:
Problem frames, assurance cases, safety cases.
The Essential Synthesis of Problem Frames and Assurance Cases

1. Introduction

Both problem frames (Jackson, 2001) and assurance cases (Weaver & Kelly, 2004) support the development and assurance of dependable software. Problem frames were originally created, and are primarily used, to help developers elicit and structure software system requirements. They support software assurance by clarifying requirements and their system context in a way that enables developers to make rigorous arguments for system validity in the context of the system’s problem. Assurance cases were developed to provide a means of documenting an argument that a system possesses a specific dependability attribute. While these attributes are also requirements of the system, they are generally only a part of the entire problem-frames process. The most common use of assurance cases at present is in the provision of arguments for safety at the system level.

These are two important technologies that appear to be related only marginally, if at all. Each makes a valuable contribution to system dependability, but we claim that a carefully formulated combination of the two provides a value over and above their individual contributions. This increased value arises because each technology enhances the other.

The combination enhances assurance cases by providing additional structure and rigor. The current formulation of assurance cases leaves many aspects of the argument excessively informal. This degree of informality is, in fact, a weakness because it increases the possibility that the assurance argument will be flawed. A suitable formulation of certain parts of the system using problem frames enables substantial rigor to be used in the informal elements of assurance cases with which we are concerned.

The combination enhances problem frames by providing a rigorous statement establishing satisfaction of the frame concern. Showing that the frame concern has been addressed is imprecise in the current formulation of problem frames, and the necessary precision and completeness are supplied by the basic structure of assurance cases.

With these two ties between assurance cases and problem frames, we have concluded that the two technologies are, in fact, strongly related. This relationship is such that the synthesis of the two is an extremely valuable structure. In this paper we discuss the relationship between problem frames and assurance cases, and we show how a composite of the two provides considerable benefit. We begin by presenting some background information on assurance cases. We then discuss how problem frames can enhance assurance cases and vice versa. We briefly describe a software
development approach that exploits the integration of the two, give an example to illustrate the various concepts in the synthesis of these two techniques, and present our conclusions.

2. Assurance Case Background

Assurance cases are the state of the art in rigorous dependability argumentation. The most common use of assurance cases at the moment occurs in the documentation of safety (Kelly, 1998), and safety cases have been built for a variety of production systems. Graphical 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) (Weaver & Kelly, 2004). The argument in Figure 1 illustrates the use of GSN in a hypothetical safety case.

In general, a safety case, an example of an assurance 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” (Bishop & Bloomfield, 1998). In its simplest form, it contains an instance of each of three essential elements: (1) a safety 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 applied recursively to produce, for real systems, a hierarchic structure with the overall goal for the system at the root. The overall goal is

---

Legend

G: Goal (property to be shown)
C: Context (inclusion indicated by \( \rightarrow \))
ST: Strategy (type of argument being made to support goal)
S: Solution (factual basis for the argument)
\( \diamond \): remains to be supported

Figure 1. Example Safety Case, Expressed in GSN
decomposed into more specific subgoals along with a strategy for arguing the validity of the overall goal from the validity of the subgoals, with evidence supporting the argument included where the goals are specific enough to accommodate it. Accompanying this basic safety-case structure are assumptions that have to be made and context information that provides details of terms and concepts used in the safety argument. An example of an assumption that might have to be made is a restriction on the operating environment, and an example of context information is a standards document that was followed in system development.

An assurance goal is typically a nonfunctional requirement such as a dependability requirement. Functional requirements tend not to be considered in an assurance case unless they relate to other, nonfunctional requirements. We believe that the emphasis on separating functional and nonfunctional requirements weakens the overall assurance argument, and that problem frames are the most promising approach to structuring an argument for correctness of functional requirements.

3. Goals and Context of the Assurance Case

The overall quality of both developed systems and assurance cases is limited by the quality of their stated requirements. For systems, those requirements are those described in the problem frames approach. For assurance cases, the system requirements define the goal that the assurance case must show for implementation assurance.

Because of their role in helping to document and structure requirements, problem frames can contribute significantly to the state of the art in assurance cases. Specifically, problem frames can be used to: (1) create and structure the implementation goal of the assurance case; and (2) structure the assurance case context.

3.1 Assurance Case Goals

Typically, the top-level goal of an assurance case is a crucial dependability requirement of the system to be assured. Safety and security are the most common goals; the goal as stated might be something like “the system is acceptably safe to operate.” What safety means for the system, and what level of safety is acceptable, is part of the context for that goal.

Such a goal is important, particularly in terms of requirements elicitation: the safety requirements of the system are some of the most important and must be considered with great care. However, assurance cases as they stand focus on the solution to a problem and not the problem itself.
We believe that it is more effective to state the goal of the assurance case in terms of the problem to be solved than to state it in terms of the developed system (the solution to the problem). Dependability concerns, such as the reliability concern (for which there already exists a discussion (Jackson, 2001)), can be used to express the dependability properties that might normally be separated into their own assurance case. By ensuring that the problem, including all of its functional and dependability concerns, is addressed, we can assure the specific dependability concerns in a more robust way.

The new top-level assurance goal in a system’s assurance case then becomes “the problem that is addressed by the system is solved.” When elaborated using problem frames, this top goal is split into two subgoals. The first subgoal is that the requirement for the problem is a valid representation of the problem. The second subgoal is that the frame concern is addressed, i.e., that the machine solves the problem. Further subgoals about correctness of implementation will branch from this second subgoal.

3.2 Assurance Case Context

There is an absence in the assurance case literature of a systematic method to capture context necessary to support the assurance argument. We use GSN as an example notation because it represents the state of the art, and so limitations in GSN are limitations in other notations as well. Context enters into an assurance case documented in GSN explicitly as a first-class node in an assurance case graph, and also implicitly via assumptions, definitions of domain terminology, strategies for decomposing arguments, and justifications for certain types of inferences. Without explicit acknowledgement, it is difficult to understand the basis for contextual aspects of the argument or to evaluate whether the claims based on context are reasonable.

Although GSN represents context explicitly, its treatment of context is otherwise mostly unstructured. Context nodes may refer to external documents such as domain standards, definitions, or any other domain-related facts, so their semantics are expressive but vague. Moreover, the contribution of a context node to an argument, such as which assumptions it enables the argument to make, is typically unstated and left to the reader to judge. Finally, unlike context diagrams, GSN does not provide support for expressing relationships among contextual elements. Consequently, the reader of an assurance case may have difficulty in discerning the contribution of context to an assurance case and in evaluating the validity of the contextual basis of an argument.
Basing the assurance case goal on the problem structure introduces a way to more elegantly and efficiently document the assurance case context, in two ways. First, the problem frame’s context diagram can identify how the various domain and standards documents are linked to the system requirements. The diagram conveys the domains with which the system will interact, enabling the assurance case to document which domains the system influences directly and which domains it ultimately influences but over which it does not have direct control. The structural depth of the diagram allows: (1) relationships to be clearly depicted; (2) domain-specific terms to be documented with other concepts in their particular domains; (3) references to other sources to be included together with the context pieces necessary to understand those sources; and (4) the distinctions between direct and indirect influence to be made.

The second reason that problem frames can improve assurance case contextual structure is that the structure and decomposition of the context can be reflected in the structure and decomposition of the frame. Each machine is connected to certain domains, and it is within those domains that the context lies. The information captured in the context diagram can be linked to the design choices made in a system—and thus to the assurance arguments that are made about that system—through the structure provided by the frame. Without this structure, evaluation of the system with respect to its context is much more difficult. Since the assurance case is an informal argument that the system solves this problem, structuring context in a comprehensible way and clearly linking it to the parts of the system to which it is relevant is essential.

4. Assurance-Based Development

The synthesis of assurance cases and problem frames leads to a new way of looking at software system development. In current practice, a system is usually developed before its assurance case. Others have suggested that the two be developed in a roughly parallel fashion (Kelly, 2004), but the synthesis that we describe here implies that they should be developed together explicitly and tied very closely.

The synthesis that is the subject of this paper is part of a new method we have created, called assurance-based development (ABD), in which the separate parts of the typical construction of an assured system—requirements and context analysis, system development, and assurance case creation—are integrated. Integration enables the whole
system development process to benefit from the careful thought given to system dependability, thought that is required to create the system’s assurance case.

The major components of assurance-based development and their high-level interactions are shown in Figure 2. At the center of the technique are the system assurance case and the system development artifacts. Because we emphasize the integration of these two components, we call the combination of the two the ABD composite. The two components are developed in parallel, and their development is coordinated to ensure that assurance goals and development artifacts are coupled explicitly and systematically. The coupling will thus reveal the evidence that the assurance case needs the development artifact to provide. This in turn determines the development activities that need to be followed to allow the assurance case to be populated with appropriate evidence. The benefit of this coupling can be seen in the argument that it provides for selection of development processes. Selection of a specific technique in software development is often ad hoc, whereas in assurance-based development this selection is determined by the evidence that it will produce as well as the artifacts developed.

Shown at the top of Figure 2 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. Because they include the dependability requirements, such as safety, they determine the primary goal of the assurance case. Because they also include the functional requirements, they are the starting point for the development lifecycle. Determining the system context accurately and fully is a major element of assurance-based development.

The synthesis of problem frames and assurance cases forms the core of ABD. Problem frames provide the basic structuring mechanism for the requirements of the system, and also for its context. The assurance case goals and con-
text are based on the system’s problem frame requirements. The development process carries the problem from the frame through to its software solution, arguing rigorously using an assurance case that the solution satisfies the problem, both in validity and verifiability.

5. Assuring Satisfaction of the Frame Concern

5.1 The Role of Assurance Cases

A general way to determine whether the frame concern is satisfied for any particular system has not been completely researched. Using software architecture to elaborate and support the necessary argument has been studied (Hall et al, 2002), but this is a general method for incorporating specific techniques, each of which in turn must be shown to support the frame concern in some specific way.

In principle, many functional requirements can be formalized and some non-functional requirements can be quantified (such as reliability and availability), and this suggests that determining whether the frame concern is satisfied could be tackled by analytic methods. This would still leave those non-functional requirements that cannot be formalized—such as requiring that a system be “user friendly”—without a satisfaction argument. Similarly, although a dependability requirement might be stated formally, the argument that such a requirement has been addressed often has to have informal as well as formal elements. This occurs because of the wealth of informal factors that contribute to the satisfaction of the requirement, e.g., confidence in the completeness of the hazard analysis process, which can only be informal.

Assurance cases, in particular safety cases, were developed specifically to support informal argument in a rigorous way. This approach is an ideal technique for dealing with frame concerns, and we believe that assurance cases are the best method for constructing the overall frame concern argument.

Assurance cases do not constrain the arguments that can be made about the frame concern. Thus, for example, an assurance case could be used to document an argument that the frame concern is modified or addressed by the software architecture used in the system’s design thereby allowing existing work to be incorporated. This argument should include either a justification that the architecture fully satisfied the frame concern or a strategy for decomposing the frame concern that shows what portion of it is addressed by the architecture.
5.2 Creating the Frame Concern Argument

The argument justifying the frame concern is not the entire assurance case, although it is a significant part of it. The complete system assurance case has to be a composite that includes the arguments justifying all of the non-functional requirements and documenting all of the required evidence, context, and assumptions. Thus, despite the synthesis that we advocate, the assurance case for a specific system will be built using appropriate existing techniques, and it will be documented using whatever notation is desired. The synthesis affects the content of the assurance case, not the process by which it is built.

Key components of any assurance case are the goals that have to be satisfied. The root node in an assurance case, for example, is the high-level assurance goal that the system has to meet. The basic structure of an assurance case is to show that high-level goals have been met by defining lower level goals together with a strategy for combining these lower-level goals, evidence, assumptions and context into the necessary argument.

Figure 3 shows the key elements of the problem frame and frame concern structures together with the general form that the top parts of an assurance case might take. The association between the two is indicated by the colors and patterns of the various components of the two parts of the figure. Thus, for example, requirements are shown in the problem frame with a box that is shaded with diagonal lines and the same pattern is used in the top-level goal in the assurance case.
Various elements of the problem frames and associated frame concerns for a system contribute details and structure to the system assurance case. More specifically, the following table shows a set of general but informal associations that hold between the elements of a problem frame and the elements of an assurance case:

<table>
<thead>
<tr>
<th>Element of Problem Frame</th>
<th>Element of Assurance Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Goals</td>
</tr>
<tr>
<td>Specification</td>
<td>High-level evidence and lower-level goals</td>
</tr>
<tr>
<td>Frame concern structure</td>
<td>Argument strategies</td>
</tr>
<tr>
<td>Domain knowledge</td>
<td>Context and assumptions</td>
</tr>
</tbody>
</table>

**Table 1: Correspondence Between Elements Of Frame Concern And Assurance Case**

With this table of associations as the basis, the elements of a system’s problem frame are incorporated into the system’s assurance case, where each element has a type in the assurance case that corresponds to its type in the problem frame. As the elements are added, the argument for the frame concern is developed within the assurance case using existing techniques for assurance case construction.

The assurance case structure fits this approach well when the top-level goal is a statement of the system requirements. The argument, then, is one supporting the claim that the requirements have been satisfied. The evidence to be used in the argument will come from the system, because the system specification, design, and implementation are what actually fulfill the requirements. Strategies can be used to document design decisions, as well as the flow of the overall argument.

### 6. Example: Runway Safety Monitor

To illustrate how assurance cases and problem frames can work together, we provide the requirements and specification aspects of the ABD composite for a system called the *Runway Safety Monitor* (RSM). The RSM is a prototype program constructed by NASA for placement in aircraft cockpits to detect runway incursions at airports. An incursion occurs if there is some obstacle on a runway that could interfere with aircraft using that runway. Incursions have become an increasing problem at busy airports, and this prototype was constructed to evaluate and demonstrate technology to alert pilots to potential incursions. Here, we focus on the problem of constructing the RSM’s model of the runway, which encompasses the runway itself and the space around the runway in which traffic is monitored.

This example steps through the different elements of the problem frame and relates them to the assurance case, as shown in Figure 4. Section 6.1 explains the overall system requirements and presents the context diagram for the sys-
tem as a whole, and Section 6.2 sets up the system’s overall problem frame. The example continues in Section 6.3 where, in order to limit length, we focus solely on the runway modeler subproblem and presents its various elements: a set of requirements adapted from the documentation we had available (Section 6.3.1); the subproblem’s problem frame and its frame concern (Section 6.3.2); the mappings between requirements and machine phenomena (Section 6.3.3); the subproblem specification (Section 6.3.4); and finally, the runway modeler’s assurance case (Section 6.3.5).

The order of these different sections reflect the general sequence in which the ABD composite would be constructed. The process begins with elicitation of requirements and the associated context, and these are structured in a problem frame and the frame concern is elaborated. The system’s specification is then developed and the problem frame refined into the frames for the various subproblems. Finally, the top level elements of the assurance case are developed based on all the artifacts developed previously.

6.1 RSM Requirements and Context Diagram

Because the RSM system is a prototype, it does not have a formal requirements document. However, after initial flight tests of the system, its designer wrote a technical report detailing the system and much of the rationale behind it (Green, 2002). We used this report as our requirements statement. The fragments that discuss the overall requirements are (we use ellipses to indicate omitted text):

The Runway Safety Monitor (RSM) was developed by Lockheed Martin in support of NASA’s Runway Incursion Prevention System (RIPS) research. ... RSM was developed as a component of the Integrated Display System (IDS), an experimental avionics software system... The advanced capabilities of IDS and RSM provide pilots with enhanced situational awareness, supplemental

---

Figure 4. Runway Modeler Example
guidance cues, a real-time display of traffic information, and warnings of runway incursions in order to reduce the possibility of runway incursions while also improving operational capability.

A runway incursion is defined by the Federal Aviation Administration (FAA) 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.” In other words, a runway incursion occurs when the use of a runway results in a conflict between an aircraft taking off or landing and other traffic, i.e., aircraft, vehicle, person, object, that may lead to a collision or loss of separation.

The context diagram for the RSM system is shown in Figure 5. In the diagram, the RSM program is the machine, and the elements of the context in which it operates are positioned around it. The “Runway” is the airport runway; for simplicity in our example we assume there is only one, but realistic runway topologies are addressed by the prototype system. The “Integrated Display System” is the pilot display mechanism. The “Ownship” is the aircraft running the RSM program in the diagram, and “FMS Data” is sensor data that the Ownship has available. “Traffic” is all other vehicles in the area of interest, and “ADS-B Data” is a communication infrastructure that allows all vehicles in the area of interest to exchange their state information (location, speed, etc.).

6.2 RSM Problem Frame

In providing information to the pilot, the RSM fits the information display frame. It must bring together several different elements: the geometry of the runway and the space within which incursions will be monitored; the state of the aircraft performing the monitoring operation (Ownship); and the state of the other objects in the system (Traffic). The decomposed frame diagram for this problem is presented in Figure 6.
This RSM decomposition follows the context diagram, so that the phenomena associated with each element of the context can be documented and analyzed together. Furthermore, the different subproblems in the RSM can be solved separately; this enables the assurance argument to be partitioned into arguments about each of the subproblems, together with an obligation to show that the partitioning is appropriate and does not lose critical detail.

6.3 Runway Modeler Subproblem

Continuing our example with just the runway modeler subproblem, we make two further simplifying assumptions. First, we assume that each airport has only one runway. Second, we assume that the runway thresholds are given in the coordinate system used for the calculations rather than in latitude and longitude. The information available to us does not contain the details we would need had we not made the assumptions. Apart from these assumptions, our model reflects the actual RSM requirements.

6.3.1. Runway Modeler Requirements. There is no specific requirements document for the runway subproblem. Rather, the requirements are woven through the technical report in a number of places because of the explanatory nature of the document. We needed a statement of requirements that we could use to show how the frame concern is satisfied, and so we reorganized the text from the report to suit our purposes. We also italicized the terms in the

![Figure 6. Problem Frame Decomposition of RSM Problem](image-url)
**Context.** A major concept of the RSM algorithm is the use of *runway incursion zones*. A runway incursion zone is a software derived three-dimensional invisible zone (not shown on displays). There is one incursion zone for each runway. The horizontal dimensions of a runway incursion zone overlay the associated *runway*. The incursion zone boundaries and zone altitude form a long rectangular shaped box that defines the 3-dimensional space where runway incursions are monitored.

**Requirement - Overall.** The RSM provides the x,y *Cartesian coordinates of the zone* used for monitoring traffic. Accurate placement of the incursion zones is critical for correct performance of the algorithm and prevention of false alarms.

**Requirement - Sides.** The width or *sides of the zone* extend a constant *side* distance from both sides of the runway. The sides of a zone are set to be near the *hold short positions* but not too close to set off false alarms when aircraft are stopped at the hold line. The zone edge value should be less than the shortest distance to the hold line, and should take into consideration possible errors in the traffic data for latitude/longitude positions. The incursion zone is wider at the *approach ends* than at the *runway thresholds* and appears to fan out toward the ends. This difference is due to the allowance of up to a *two-dot ILS localizer deviation error* on approaches.

**Requirement - Ends.** The length or *ends of the zone* extend a constant *end* distance from both ends (thresholds) of the runway. The ends of zones vary based on the intersection of the *ILS glide slope path* with the incursion zone *altitude*. This distance is determined by calculating the position where the glide slope path intersects the runway incursion zone altitude.

**Requirement - Altitude.** The third (vertical) dimension of the incursion zone is a *constant* *vertical* value for the *altitude of the zone* above the runway surface. This defines the maximum altitude above ground level (AGL) over the runways and approach areas below which a traffic conflict would be considered a runway incursion. The value can be changed in the RSM configuration file based on different airport conditions and requirements.

---

6.3.2. Runway Modeler Problem Frame. The problem frame for the runway modeler subproblem, along with its frame concern, is summarized in Figure 8. The phenomena associated with the frame are classified into A, B, C, and D phenomena and are listed in Figure 9.

Like all frame concern statements, the frame concern for the runway modeler effectively says that the system must satisfy its requirements. As is also fairly common, the runway modeler’s frame concern statement, as initially formulated, is relatively vague, and is not sufficient to determine whether the concern has actually been met. The synthesis that we describe deals with this by: (1) mapping the initial frame concern into a more detailed statement based on the requirements; and (2) organizing this statement into a form suitable as a top-level goal in the assurance case.
6.3.3. Runway Modeler Domain Knowledge. Two types of domain knowledge are used in our example. One type is knowledge that helps refine requirements from ones that are too abstract to be implemented by a machine to those that can be translated directly into a specification (in terms of the reference model of Gunter et al. (2000), it links phenomena in \( e_h \) with phenomena in \( e_v \)). We do not list this knowledge separately, because it can be found in the assurance case (below) as justifications and evidence.

<table>
<thead>
<tr>
<th>A Phenomena</th>
<th>B Phenomena</th>
<th>C Phenomena</th>
<th>D Phenomena</th>
</tr>
</thead>
<tbody>
<tr>
<td>runway</td>
<td>incursion zone</td>
<td>runway</td>
<td>incursionZone</td>
</tr>
<tr>
<td>constant [side] distance</td>
<td>coordinates of the zone</td>
<td>width</td>
<td>extBeginSide1</td>
</tr>
<tr>
<td>sides of the runway</td>
<td>sides of the zone</td>
<td>hdg</td>
<td>extBeginSide2</td>
</tr>
<tr>
<td>constant [end] distance</td>
<td>ends of the zone</td>
<td>glideslope</td>
<td>beginSide1</td>
</tr>
<tr>
<td>thresholds of the runway</td>
<td>altitude of the zone</td>
<td>distToLHS</td>
<td>beginSide2</td>
</tr>
<tr>
<td>constant [vertical] value</td>
<td></td>
<td>beginVertex</td>
<td>endSide1</td>
</tr>
<tr>
<td>ILS glide slope path</td>
<td></td>
<td>endVertex</td>
<td>endSide2</td>
</tr>
<tr>
<td>hold short positions</td>
<td></td>
<td>IZEndDist</td>
<td>extEndSide1</td>
</tr>
<tr>
<td>approaches</td>
<td></td>
<td>IZAlt</td>
<td>extEndSide2</td>
</tr>
<tr>
<td>two-dot ILS localizer deviation error</td>
<td></td>
<td>localizerError</td>
<td>altitude</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ILSLocDist</td>
</tr>
</tbody>
</table>

Figure 8. Runway Modeler Frame Concern Summary

Figure 9. Runway Modeler Phenomena
**Figure 10. Runway Modeler Domain Description (B and D phenomena)**

<table>
<thead>
<tr>
<th>B Phenomena</th>
<th>D Phenomena</th>
</tr>
</thead>
<tbody>
<tr>
<td>incursionZone</td>
<td>incursion zone</td>
</tr>
<tr>
<td>altitude</td>
<td>altitude of the zone</td>
</tr>
<tr>
<td>extendBeginSide1, beginSide1, endSide1, extendEndSide1, extendBeginSide2, beginSide2, endSide2, extendEndSide2</td>
<td>coordinates of the zone</td>
</tr>
</tbody>
</table>

**Figure 11. Runway Modeler Domain Description (A and C Phenomena)**

<table>
<thead>
<tr>
<th>A Phenomena</th>
<th>C Phenomena</th>
</tr>
</thead>
<tbody>
<tr>
<td>runway</td>
<td>runway</td>
</tr>
<tr>
<td>constant [vertical] value</td>
<td>IZAlt</td>
</tr>
<tr>
<td>ILS glide slope path</td>
<td>glideslope</td>
</tr>
<tr>
<td>constant [end] distance</td>
<td>IZEndDist</td>
</tr>
<tr>
<td>constant [side] distance</td>
<td>distToLHS - IZLHSDist</td>
</tr>
</tbody>
</table>
The second type of knowledge corresponds to the *domain description* of Jackson. It links world phenomena with machine phenomena, so that a correlation can be drawn between the requirements and specification. The domain description is shown in Figures 10 and 11.

All of the domain knowledge of both types that is documented here was obtained from a combination of the technical report (Green, 2002) and the RSM source code.

---

**Figure 12. Runway Modeler Specification**

```plaintext
coord: TYPE = [# x: real, y: real #]
angle: TYPE = (h: nonneg_real | h < 360)
pos_acute_ang: TYPE = [a: angle | 0 < a AND a < 90]
latitude, longitude: TYPE

% Defined elsewhere
distance(a, b: coord) : real
% returns a point c s.t. b between a and c
addPtOnLine(a, b: coord, dist: real) : coord
% add a perpendicular at pi/2 or 3pi/2 degrees to ab of distance dist with endpoint b
addPi_2Perp(a,b:coord, dist:real) :coord
add3Pi_2Perp(a,b:coord, dist:real) :coord

runway : TYPE = [# width         : posnat,
                 hdg           : angle,
                 glideslope   : pos_acute_ang,
                 distToLHS    : nat,
                 beginVertex  : coord,
                 endVertex    : coord,
                 IZEndDist    : real,
                 IZAlt        : real,
                 IZLHSDist    : {n:nat | n<distToLHS},
                 localizerError: int,
                 ILSLocDist   : int #]

incursionZone : TYPE = [# extBeginSide1 : coord,
                       beginSide1   : coord,
                       beginSide2   : coord,
                       endSide1     : coord,
                       endSide2     : coord,
                       extEndSide1  : coord,
                       extEndSide2  : coord,
                       altitude     : nonneg_real #]

izone_init(r: runway) : incursionZone =
(Let
  zoneCenterBegin: coord = addPtOnLine(r`endVertex, r`beginVertex, r`IZEndDist),
  zoneCenterEnd: coord = addPtOnLine(r`beginVertex, r`endVertex, r`IZEndDist),
  edgeDist: nat = r`distToLHS - r`IZLHSDist,
  extDist: real = (r`IZAlt / tan(r`glideslope)) - r`ILSLocDist,
  zoneCenterBeginExt: coord = addPtOnLine(r`endVertex, zoneCenterBegin, extDist),
  zoneCenterEndExt: coord = addPtOnLine(r`beginVertex, zoneCenterEnd, extDist),
  extendRadius: real =
    (350 * (distance(r`beginVertex, r`endVertex) + r`ILSLocDist + extDist)) /
    (distance(r`beginVertex, r`endVertex) + r`ILSLocDist) IN

  (# beginSide1 := add3Pi_2Perp(zoneCenterEnd, zoneCenterBegin, (r`width/2)+edgeDist),
   beginSide2 := addPi_2Perp(zoneCenterEnd, zoneCenterBegin, (r`width/2)+edgeDist),
   endSide1 := addPi_2Perp(zoneCenterBegin, zoneCenterEnd, (r`width/2)+edgeDist),
   endSide2 := add3Pi_2Perp(zoneCenterBegin, zoneCenterEnd, (r`width/2)+edgeDist),
   extBeginSide1 := add3Pi_2Perp(zoneCenterEnd, zoneCenterBeginExt, extendRadius),
   extBeginSide2 := addPi_2Perp(zoneCenterEnd, zoneCenterBeginExt, extendRadius),
   extEndSide1 := addPi_2Perp(zoneCenterBegin, zoneCenterEndExt, extendRadius),
   extEndSide2 := add3Pi_2Perp(zoneCenterBegin, zoneCenterEndExt, extendRadius),
   altitude := r`IZAlt #))

```

---
6.3.4. Runway Modeler Specification. We formally specified the runway modeler in PVS (Owre 2001). The specification (barring certain standard syntactic elements) is presented in Figure 12. The specification includes placeholders for the phenomena for some runway, but does not include the runway phenomena themselves; the types in the specification must be instantiated for each specific runway. Figure 13 shows the specification instantiated with the characteristics used for the runways at Dallas/Fort Worth (DFW), where the flight tests of the RSM were held. The information shown in Figure 13 was taken from the technical report and the source code data files for DFW.

6.3.5. Runway Modeler Assurance Case. With the necessary details of the requirements and the specification developed for the example, we now describe the assurance case. The assurance case begins with the statement of requirements (R) and refines those requirements into a specification (S) using domain knowledge (K) that links world phenomena to machine phenomena. Essentially, this takes the $S, K \models R$ formula—which is a verification argument—and shows that it holds through forward refinement (Zave 1997). The forward refinement process is not formal, so justifications are included where the reasoning behind the refinement steps is not immediately obvious.

The overall runway modeler requirement and the initial breakdown of the requirement into subgoals are shown in Figure 14 (recall that our statement of the requirements is shown in Figure 7). In the figure, the initial goal of the assurance case corresponds to the text in our requirements statement for the overall modeler requirement (which, in turn, corresponds to text in the technical report). The material that introduces the context requirements is included as a context box in the argument (labelled C01 in Figure 14).

The initial strategy for showing that the requirements have been met, as shown by the strategy box (labelled ST01), is to look at each of the dimensions of the zone. This strategy is informed by the context of the requirements (as indicated by the arrow linking the strategy to the context). The context sets the zone to be a 3-dimensional shape, so the strategy creates 3 subgoals, one for each dimension. In the figure, each subgoal has a diamond beneath it, indicating that it will be filled out in another place in the argument (or, in the case of this paper, in a subsequent figure).

DFW_35C : runway =
(# width := 150,
hdg := 350,
glideslope := 3,
distToLHS := 9050,
beginVertex :=
(# x := 1200.28, y := -5.65404 #),
endVertex :=
(# x := 1977.51, y := 11209.34 #),
IZEndDist := 1000,
IZAlt := 400,
IZLHSDist := 40,
localizerError := 350,
ILSLocDist := 1000 #)

Figure 13. Runway Modeler For Dallas/Fort Worth
Figure 15 shows the argument supporting goal G1 in Figure 14. (From this point, we choose only part of the argument to elaborate for reasons of space.) G1 is too vague to formalize directly into a specification in terms of the machine phenomena, and so we bring in the additional requirements specifically about the sides of the zone as context. We document this strategy as “requirements elicitation and refinement,” since in practice they might be obtained by returning to a customer for further details. These additional requirements enable us to break down G1 into more specific subgoals, G1.1 - G1.3.

Continuing our process of goal refinement, we expand G1.2 (the most complex of the three subgoals of G1). At this point in the argument, the goal is specific enough that it can be mapped into system phenomena using the domain description shown in Figures 11 and 10. The description itself is brought into the argument as context (box C1.2) at this step.

Using the description, we split G1.2 into two further subgoals, G1.2.1 and G1.2.2. At this stage of the argument, we must be very careful: the informal information contained in the goal probably assumes many things about the real world that are not stated directly. For example, G1.2.1 requires that the incursion zone be outside the runway proper; the natural-language requirements text implies this, but it is not brought out until the requirements are formalized. The problem of capturing implicit assumptions at this step is the same one that occurs at any stage of requirements; it

---

Legend

G: Goal (property to be shown)
C: Context (inclusion indicated by  \(\rightarrow\) )
ST: Strategy (type of argument being made to support goal)
S: Solution (factual basis for the argument)
: remains to be supported

Figure 14. Runway Modeler Assurance Case
is not a by-product of our specific approach. For particularly complex formalizations, additional goals might be added to ensure that sufficient thought was given to the assumptions that might have been implicit.

The last step of the argument requires knowledge of data that can vary from runway to runway. To denote this, we use a strategy saying that we will defer the rest of the argument to the particular installation of the system. We leave the goal unsupported at this point, enabling it to be instantiated in different ways for different installations.

Figure 16 shows the argument for runway 35C at Dallas/Fort Worth. The specifics for this part of the argument were taken from the DFW instance specification (shown in Figure 13), along with flight test analysis presented in the technical report. This step of the argument is where the goals are given solutions, evidence to directly support the argument. G1.2.1-DFW (the version of G1.2.1, specialized for runway 35C at DFW) requires only one element to its solution: the formal definition of the IZLHSDist symbol. G1.2.2-DFW includes flight test results to indicate what
the exact value for IZLHSDist should be, and then shows that the variable takes on that value, given the instance specification. Together, these two solutions show that the goal is met.

7. Conclusion

While problem frames and assurance cases have been developed and applied separately, their synthesis is much more powerful. Each brings a solution to a problem present in the other. Problem frames have no systematic and comprehensive way to show that the frame concern has been satisfied, and assurance cases provide a rigorous approach to developing this argument. Assurance cases rely on a wealth of requirements-related material in their content, but there is no systematic and comprehensive way to include and utilize this information in the present formulation of assurance cases. Problem frames provide a suitable mechanism.

In this paper we have described this synthesis and presented the rationale behind the synthesis, the particular pieces of the two technologies that influence each other, and the beginning of a process to integrate the two in software system development. To illustrate the ideas we have presented a detailed example based on a prototype avionics system. The benefits of the synthesis can be gained quickly and easily by developers using problem frames and assur-
ance cases because the synthesis can be accomplished with the artifacts that they must create in the standard course of system development.

Acknowledgments

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. We also benefited from recent discussions on a similar topic, and on the example, with colleagues at Politecnico di Milano. This work was funded in part by NSF grant CCR-0205447 and NASA grant NAG1-02103.

References


ZAVE, P., AND M. JACKSON (1997) Four dark corners of requirements engineering, ACM Transactions on Software Engineering and Methodology (TOSEM) 6 (1)