Author's pre-print version.
The definitive version of this article appears in the Proceedings of the 2026 AIAA SciTech Forum.
DOI: 10.2514/6.2026-1251

AdvoCATE: The Assurance Case Automation Toolset at Age 14

Ewen Denney* and Ganesh Pai†
KBR Inc., NASA Ames Research Center, Moffett Field, California, 94035

Assurance cases (ACs) are finding increasing adoption in many safety- and security-critical domains as a paradigm focused on communicating to stakeholders why a system or service can be relied upon, through an explicit statement of assurance objectives and rationale that shows how those objectives are met. ACs give the flexibility to both keep pace with technological change, e.g., with the rapid introduction of new technologies such as machine learning (ML), and to tailor verification efforts to be proportional to the levels of risk posed. Since the Assurance Case Automation Toolset (AdvoCATE) was first introduced to the community in 2012, it has inspired the development of various similar toolsets and capabilities for creating ACs. AdvoCATE has continued to evolve from its initial focus on creating structured arguments to capture assurance rationale, using the Goal Structuring Notation (GSN), to encompass a broader suite of features for safety risk management, including hazard analysis, risk modeling using barrier models captured using Bow Tie Diagrams (BTDs), capturing requirements, and organization of substantiating evidence as evidence logs. We use the assurance of an autonomous visual landing capability with a machine learnt component as a running example to showcase the utility and applicability of AdvoCATE.

I. Introduction

Consider the following scenario: a new technical system has been engineered to provide a service, e.g., cargo transportation using an uncrewed electric aircraft or a self-driving road vehicle. But before you can deploy this system you need approval from a regulatory authority who needs to have confidence that your system and service are safe and secure. More generally, the stakeholders of your system/service need to be convinced that it can be relied upon.

The *assurance case* (AC) paradigm is one way to communicate why one can be justifiably confident in trusting a critical system or service. The necessity for ACs was recognized well over two decades ago, and they have since been steadily adopted in many safety- and security-critical domains and organizations, including at NASA. For instance, the NASA System Safety Handbook [1, 2] places particular focus on ACs as a central organizing concept. ACs are also a core element of the more recent "objectives-driven, risk-informed, case-insured" approach to assurance at NASA [3, 4].

Although there are many notions of what an assurance case is, a common definition, due to [5], is "a reasoned and compelling argument, supported by a body of evidence, that a system, service or organization will operate as intended for a defined application in a defined environment". This is, itself, a generalization of the *safety case* concept, i.e., an AC focused on safety [6]. Based on that definition, evidently assurance *arguments* tend to be a major focus in an AC. They usually comprise: (i) explicit claims about the intended properties of a system or service in which confidence is necessary, and which serve as the conclusions of the argument; and (ii) structured reasoning comprising a series of statements linking those claims to *evidence*, i.e., verifiable statements of fact linked to information arising from the development of the system/service, which serve as premises in the argument.

As such, research surrounding ACs (including our own earlier work) has focused on assurance argumentation. However, informed by our practical experience, our viewpoint has evolved to recognize that the information upon which an argument rests is as crucial, if not more so. Indeed, as will be described later in this paper, other forms of reasoning that may not (or need not) take the form of an explicit argument, may better engender confidence [7]. Our key observations here are that: (i) what an AC needs to deliver is rationale for confidence (with that rationale itself exhibiting characteristics such as validity), and (ii) arguments serve as one amongst a core set of artifacts answering that need. Thus, for what follows, our preferred notion of an AC is a comprehensive, defensible, and valid *justification* that a system or service will function as intended for a defined application and operating environment [8].

*Technical Leader and Sr. Technical Fellow, Mission Technology Solutions (Science & Space), AIAA Member.
†Principal Researcher and Sr. Technical Fellow, Mission Technology Solutions (Science & Space), AIAA Associate Fellow.

A. Tool Support for Assurance Case Development

To both further (our) research in AC development, and support practical engineering work requiring ACs, in 2012 we introduced AdvoCATE, the assurance case automation toolset to the community [9]. As mentioned earlier, the initial focus was on developing graphical representations of structured arguments using the Goal Structuring Notation (GSN) [5], and various related capabilities including: integrating the results of formal verification [10]; abstraction using hierarchy [11], patterns [12], and modularity [13]; and retrieval of the relevant assurance-related information through queries and views [14].

The broad goal, overall, has been to provide automation support for creating and assembling assurance arguments (albeit primarily in an aviation context) [15], supported by a methodology where an AC evolves alongside the system or service for which it is developed, from concept into operation [8]. Not only have we ourselves used AdvoCATE to create ACs—in particular to support regulatory approval of uncrewed aircraft systems (UAS) operating beyond visual line of sight (BVLOS) [16, 17]—but also, over the years, the toolset has been distributed both nationally and internationally to a wide user base that includes both government entities and the safety-critical industry, in the aviation, automotive, cybersecurity, manufacturing, maritime, medical device, mining, military, nuclear, space, telecommunications, and rail transportation domains. Additionally, since its introduction, AdvoCATE has inspired various efforts at tool support for development of AC arguments [18–20]*.

B. This Paper

The need for consistency between a system and its AC, as the former evolves through life [22], has subsequently motivated an expansion of our original focus in developing AdvoCATE. Specifically, our toolset is now additionally concerned with creating and managing the supporting information necessary for assurance, including other mechanisms to communicate assurance rationale. This revised focus has been both reinforced and sharpened with the introduction of machine learning (ML)—and more generally learning-enabled functionality—into critical systems [23].

The toolset has been reengineered and remodeled from the version described in earlier papers, but retains key functionality, such as structured arguments. In this paper, we will provide the first description of other features, including hazard and requirements logs, improved and extended bow tie diagrams, and evidence and tools logs. At the core of all these features is an integrated assurance metamodel, which allows AdvoCATE to keep these diverse artifacts consistent, and enables traceability both between distinct assurance artifacts, as well as between distinct representations: textual using domain-specific languages (DSLs), tabular, and graphical. This assurance model AdvoCATE enforces numerous well-formedness constraints on the structure of the assurance case, but does not require the use of formal languages to describe any of the constituent assurance artifacts. Formal methods can, however, be used to create evidence (see Section VII).

To illustrate the key features of AdvoCATE, we use a running example of an Autonomous Visual Landing (AVL) system, we describing the operational concept, also identifying the system boundary and its interfaces to the environment (Section II). Due to space constraints, we do not give all details of the methodology, so here we give a high-level overview of the AC development methodology that we typically follow with AdvoCATE:

- First, we conduct a hazard analysis, either as a Preliminary Hazard Analysis (PHA) or a Functional Hazard Analysis (FHA) based on the application. Either of the PHA or FHA can be documented in a tabular *hazard log* in AdvoCATE (Section III).
- Next, we perform a system-level risk analysis and assessment. If the initial risk levels are deemed to be unacceptable, then we identify additional mitigations—barriers providing a risk reduction function—until the residual risk is at acceptable levels. We accomplish this, in part, through *risk scenario* modeling, as an additional safety technique (Section IV).
- Then, the needs of the *safety system*, comprising a collection of the identified barriers, are captured in the *requirements log*, which can include both safety and assurance requirements, as well as requirements on design (Section V).
- Thereafter, we capture assurance rationale in the form of GSN *arguments* for various assurance concerns (Section VI), including, but not limited to: (i) validity of the mitigations specified for identified hazards, (ii) rationale for initial risk levels, (iii) why identified mitigations reduce risk, (iv) rationale for residual risk levels, and (v) assumptions made in the scenario-based risk analysis.
- Lastly, substantiating evidence is documented in the *evidence log* (Section VII).

We conclude the paper in Section VIII.

*The review of AC tools listed in [20] also includes various tools that were developed prior to AdvoCATE, e.g., [21].

II. Running Example: Autonomous Visual Landing

We consider a running example of an Autonomous Visual Landing (AVL) application from our prior work [24]. In this application, assurance of ML-based functionality providing perception and landing decision support is required.

A. Operating Concept

To land, the aircraft must enter a standard traffic pattern which consists of defined phases and *legs* at which the pilot in command (PIC) makes specific decisions for safe landing. For this example, when the aircraft is on the *final* leg of the *approach* phase, the ML-based functionality is invoked. The aircraft is required to be aligned with the runway, and descending at a steady rate, i.e., following a typical *glide path* with a fixed angle, usually between $3^\circ - 5^\circ$) to the horizontal plane of the runway.

The decision whether to continue to land or to go-around, i.e., abort the landing, can be made by the PIC at any point in the aircraft descent trajectory, depending on the advisory received from the ML-based function on the (perceived) state of the runway (e.g., the potential for a collision hazard from a runway incursion), and the stability of the aircraft configuration as it descends. Various additional assumptions and constraints apply for this Concept of Operations (CONOPS), details of which are not in scope for this paper, but have been described in more detail in [24].

B. System Description

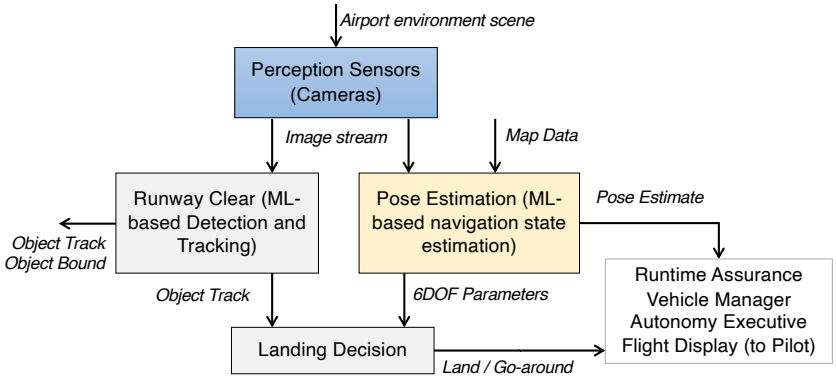


Fig. 1 High-level functional architecture for an ML-based Autonomous Visual Landing capability.

As shown by the high-level functional architecture (Fig. 1), the AVL capability is aided by two functions implemented using ML:

- *Runway Clear* (RWYCLR), responsible for perception, detection, and tracking of potential collision hazards, and for estimating the future position of the detected hazards relative to the aircraft after it lands;
- *Pose Estimation* (POSESTM), responsible for determining, respectively, the three dimensional orientation of the aircraft relative to the runway, and its translational position relative to the touchdown location on the runway. Effectively this is a sub-function for the *Localization* function of the aircraft.

For this paper, we mainly focus on POSESTM, in particular, its contribution to overall landing safety. POSESTMreceives a stream of images of a defined resolution at defined rate as input, along with external map data. In response it produces the Six Degree of Freedom (6-DOF) aircraft pose estimate, along with the pixel locations of the *keypoints* of the runways, i.e., specific points of interest in an image or scene, e.g., the 4 end-points of the runway. For simplicity, we assume a physical architecture comprising dedicated subsystems to which each of the functions aiding the AVL capability are allocated. That is, we assume that POSESTM is allocated to a *Pose Estimation Subsystem*. For additional details of the example—which are, again, out of scope for this paper—refer to [24].

III. Hazard Logs and Risk Assessment

A. Overview

AdvoCATE has a tabular representation for the *hazard log*, to incrementally record the results of a PHA or FHA. The hazard log can be populated by stepping through a series of *hazard views*—accessible through a drop-down menu—each

of which is focused on one specific step of the hazard analysis.

Specifically, the *hazard identification view* shows only those columns of the hazard table that are most relevant to recording the identified hazards. Thereafter, the *risk analysis view* shows the columns required for establishing the initial risk level posed by the (effects of) the identified hazards in the context of pre-existing mitigations. The *risk assessment view* then shows the columns needed to specify new mitigations for those hazards whose initial risk level is unacceptable, and to determine the residual risk level after the introduction of those new mitigations. Lastly, the *mitigation requirements view* shows those columns needed to specify the requirements corresponding to the new mitigations defined in the risk assessment view. These requirements also appear in the requirements log (Section V), where more details can be provided.

Figs. 2 and 3 show (screenshots of) excerpts from the hazard identification and risk assessment views, respectively, for the AVL example. As shown in the figures, a hazard analysis in AdvoCATE is organized by a *hazardous activity* (shown as a column in the table), and the related *system states* and *environmental conditions* (selected via a drop-down menu, alongside hazard views). Hazardous activities represent operational behaviors that inherently pose risk, such as an aircraft landing operation. Each such activity can be further characterized in terms of system states, representing the evolution of the system during that activity. Additionally, environmental conditions capture the operating context for the hazardous activity and the constituent system states. Thus, in AdvoCATE, a hazard represents a particular *loss of control* event (which may be a functional failure condition), that poses a potential for harm when it occurs in a specific system state of a hazardous activity, under the stated environmental conditions.

Hazardous activities are a central organizing element of risk scenario modeling (described next in Section IV), which relates hazards to their causes, effects, and mitigations as captured in the hazard log. In effect, hazardous activities provide a means to connect operational safety assessment to an FHA, which rather focuses on system functions.

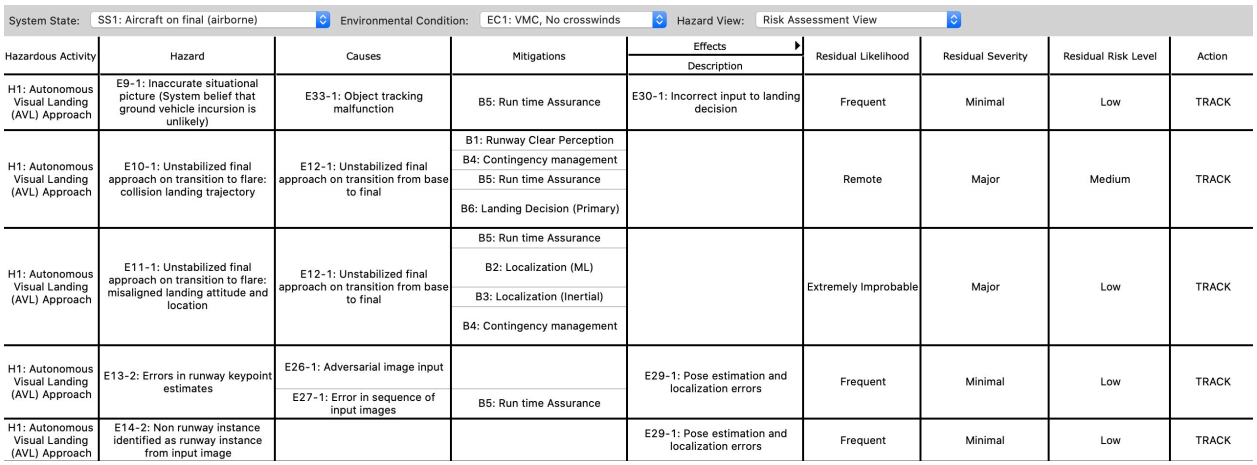
Hazards can be associated with the functional and physical elements of the system architecture. Within AdvoCATE, we formulate the latter using a simple DSL (detailed omitted here) as hierarchical breakdowns of the physical and functional architectures, which are then *allocated* to the identified hazards via the corresponding column in the hazard log (see Fig. 2). Additionally, for each physical/functional element, the DSL allows specifying the respective physical failure modes, or functional failure conditions. Those can then be selected as the causes of the hazard, as appropriate.

System State: SS1: Aircraft on final (airborne) Environmental Condition: EC1: VMC, No crosswinds Hazard View: Hazard Identification View

Hazardous Activity	Hazard	Allocation	Condition	Hazard Type	Causes	Effects
						Description
H1: Autonomous Visual Landing (AVL) Approach	E9-1: Inaccurate situational picture (System belief that ground vehicle incursion is unlikely)	autonomousVisualLanding: AF1: Autonomous landing with vision-based sensing	wrongWorldModel: AVL model of world inconsistent with ground truth	Safety	E33-1: Object tracking malfunction	E30-1: Incorrect input to landing decision
H1: Autonomous Visual Landing (AVL) Approach	E10-1: Unstabilized final approach on transition to flare: collision landing trajectory	aircraft: Aircraft	collisionCourseApproach: Landing trajectory on a collision course	Safety	E12-1: Unstabilized final approach on transition from base to final	
H1: Autonomous Visual Landing (AVL) Approach	E11-1: Unstabilized final approach on transition to flare: misaligned landing attitude and location	aircraft: Aircraft	misalignedApproach: Aircraft misaligned with runway	Safety	E12-1: Unstabilized final approach on transition from base to final	
H1: Autonomous Visual Landing (AVL) Approach	E13-2: Errors in runway keypoint estimates	RunwayLocalization: SF2.1.1: Runway Localization	poseEstimationMalfunction: Pose Estimation malfunction	Safety	E26-1: Adversarial image input E27-1: Error in sequence of input images	E29-1: Pose estimation and localization errors
H1: Autonomous Visual Landing (AVL) Approach	E14-2: Non runway instance identified as runway instance from input image	RunwayLocalization: SF2.1.1: Runway Localization	incorrectKeypoints: Incorrect estimates of runway keypoints	Safety		E29-1: Pose estimation and localization errors
H1: Autonomous Visual Landing (AVL) Approach	E28-1: Error in T-3DOF aircraft pose estimate	primaryStateEstm: SF2.1: Primary Pose (state) Estimation and Localization		Safety	E29-1: Pose estimation and localization errors	
H1: Autonomous Visual Landing (AVL) Approach	E24-1: Wrong object tracked	Tracking: SF1.2: Object tracking	wrongTrack: Wrong object tracked	Safety	E21-1: Wrong object detected	E33-1: Object tracking malfunction

Fig. 2 AdvoCATE screenshot: FHA excerpt for the AVL example showing the hazard identification view.

Hazards are also associated with *risk levels*, that are determined according to a risk acceptance matrix, based on the combination of the likelihood and severity of the corresponding worst-case consequences. Fig. 3, shows a *residual risk level*, which represents the risk that remains after the correct implementation and verification of the necessary hazard mitigations. Not shown is an associated *initial risk level*, which represents the unmitigated risk posed by a hazard.



Hazardous Activity	Hazard	Causes	Mitigations	Effects: Description	Residual Likelihood	Residual Severity	Residual Risk Level	Action
H1: Autonomous Visual Landing (AVL) Approach	E9-1: Inaccurate situational picture (System belief that ground vehicle incursion is unlikely)	E33-1: Object tracking malfunction	B5: Run time Assurance	E30-1: Incorrect input to landing decision	Frequent	Minimal	Low	TRACK
H1: Autonomous Visual Landing (AVL) Approach	E10-1: Unstabilized final approach on transition to flare: collision landing trajectory	E12-1: Unstabilized final approach on transition from base to final	B1: Runway Clear Perception B4: Contingency management B5: Run time Assurance B6: Landing Decision (Primary)		Remote	Major	Medium	TRACK
H1: Autonomous Visual Landing (AVL) Approach	E11-1: Unstabilized final approach on transition to flare: misaligned landing attitude and location	E12-1: Unstabilized final approach on transition from base to final	B5: Run time Assurance B2: Localization (ML) B3: Localization (Inertial) B4: Contingency management		Extremely Improbable	Major	Low	TRACK
H1: Autonomous Visual Landing (AVL) Approach	E13-2: Errors in runway keypoint estimates	E26-1: Adversarial image input E27-1: Error in sequence of input images	B5: Run time Assurance	E29-1: Pose estimation and localization errors	Frequent	Minimal	Low	TRACK
H1: Autonomous Visual Landing (AVL) Approach	E14-2: Non runway instance identified as runway instance from input image			E29-1: Pose estimation and localization errors	Frequent	Minimal	Low	TRACK

Fig. 3 AdvoCATE screenshot: FHA excerpt for the AVL example showing the risk assessment view.

AdvoCATE provides a collection of dashboards that give summary information about the various facets of the assurance case, and allow simple creation and editing. Fig. 4 shows excerpts from the hazards and evidence dashboard pages (see also Section VII). Other dashboard pages, not shown here, are provided for arguments, patterns, safety architecture, requirements, and tools.

B. Example

In the context of the AVL example, main safety assurance objectives are to provide sufficient confidence that under all specified operating conditions:

- neither the intended behavior of PosEstm nor its failure conditions lead to an unacceptable outcome, and
- PosEstm does not exhibit any unintended behavior that could lead to an unacceptable outcome at a rate more frequent than that corresponding to an acceptable risk level—or equivalently, a Target Level of Safety (TLOS)—for those outcomes.

The unacceptable outcomes to be avoided, as identified from an FHA are: Controlled Flight Into Terrain (CFIT); a landing in an area other than the intended runway; and a runway excursion. These outcomes are causally preceded by *landing safety hazards*, which are, as mentioned earlier, a combination of uncontrolled system states and specific environmental conditions that occur during landing (a hazardous activity). For the operational context relevant to PosEstm, the relevant landing safety hazard is, primarily, an *unstable approach*, i.e., when the aircraft does not maintain its essential flight parameters (such as its attitude, landing configuration, speed, descent rate, and power settings) within the limits established for an airworthy aircraft type design.

We assume here that RwyClr has established that there is no collision hazard in the landing trajectory. We additionally assume for simplicity that only a single object in the airport environment can pose a collision hazard at any given time. Thus, from a system safety standpoint, an additional unacceptable outcome to be avoided during landing (to which RwyClr contributes) is collision with objects on the ground, such as a ground vehicle on the runway or taxiway, or another aircraft. Referring to the functional architecture of Fig. 1, the LndDesc function uses the pose estimate and the track of a detected object to inform the decision/advisory of whether to land.

IV. Risk Scenario Modeling

Risk scenarios and *safety architectures* created within AdvoCATE offer a different approach from arguments to substantiate the safety-related claims in an AC. Although, they are closely related to the arguments created within AdvoCATE, both from a conceptual, and from a modeling standpoint. In particular, risk scenarios are views of a safety architecture (described in more detail subsequently in this section). Conceptually, the latter implicitly embodies the *defense-in-depth* and *risk-based mitigation* argumentation patterns. That is, the argument underlying a safety architecture is that layers of independent mitigations, each of which provide a risk reduction function, collectively reduce the risk of the effects of the identified safety hazards to an acceptable level. From a modeling standpoint, the elements of a safety architecture in AdvoCATE can be related to the other assurance artifacts, including arguments. For example, an

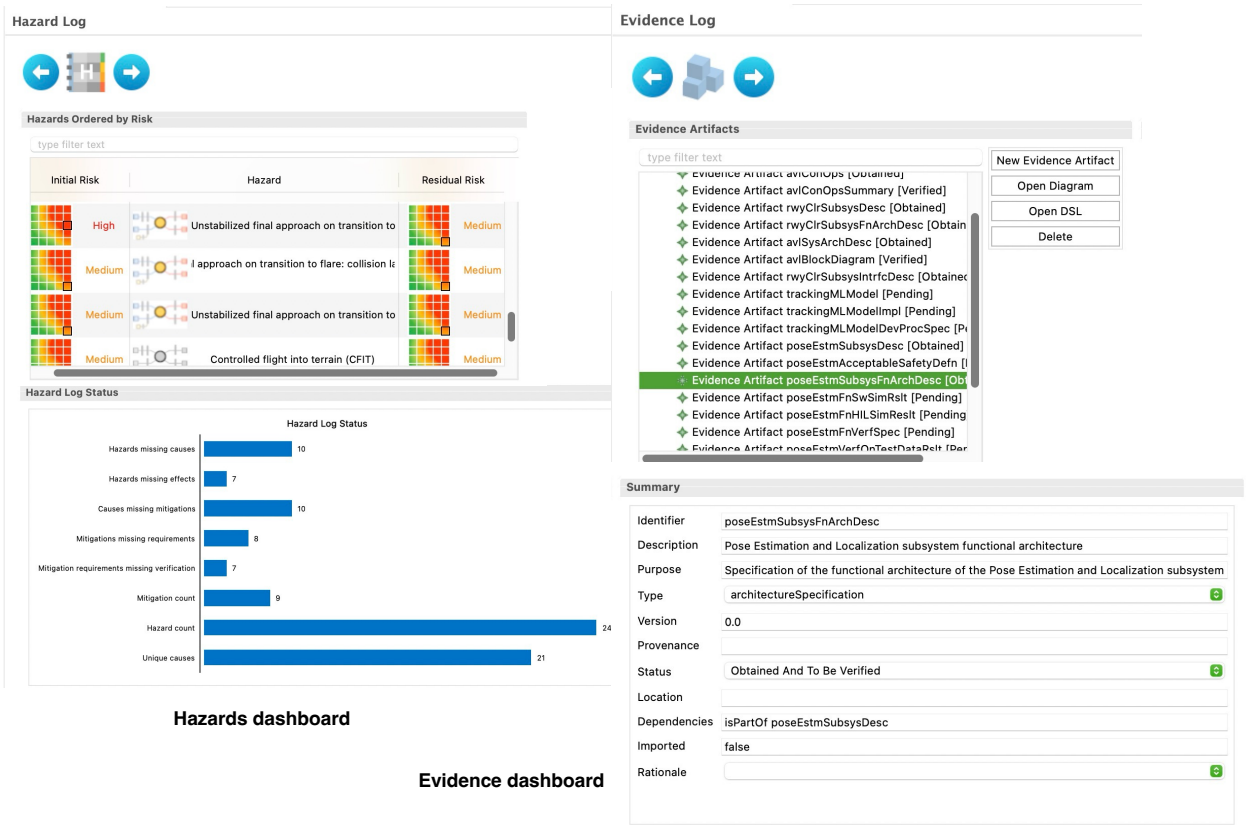


Fig. 4 AdvoCATE dashboard elements showing summary information from the hazard log and evidence log.

argument can be created to substantiate the claim that the mitigations in a safety architecture are independent.

A. Risk Scenarios

The goal of risk scenario modeling is, effectively, to characterize the various operational scenarios of a system elaborating, (i) the conditions leading to a loss of control and eventually to harm/loss; (ii) the invocations of the mitigation mechanisms; and (iii) the conditions that compromise the mitigations.

Risk scenarios are causal event chains showing how initiating (operational) events lead to loss of control events (hazards) that eventually manifest as the undesired effects to be avoided. In AdvoCATE, we use *barrier models*, represented using Bow Tie Diagrams (BTDs) to capture risk scenarios. The underlying metamodel has a close connection to that used as the basis for the hazard log. Thus, a risk scenario allows us to relate the functional hazards, their precursors, contributing failure conditions, and their inter-relations identified via the FHA.

Fig. 5 shows a fragment of a risk scenario in the BTD notation, for the AVL example discussed in Section II. This BTD shows *threat* (i.e., initiating) events (e.g., 'Adversarial Image Input') leading to the *top event*[†] 'Errors in Runway Keypoint Estimates', that eventually leads to the terminating consequence event (effect) 'Pose Estimation and Localization Errors'. Thus, this risk scenario partially models function failure conditions due to the propagation of sensor errors across the function interface to the ML model that implements the *Runway Localization* and *Keypoint Estimation* sub-functions of PosEstm.

Fig. 5 also shows a series of (hierarchically organized) mitigations, (known as *barriers* and *controls*, respectively, in BTD terminology), that are meant to reduce risk by either preventing or recovering from the identified hazard. For example, Run time Assurance is a barrier function that serves to mitigate the risk posed by the errors in a sequence of input images. More specifically, this barrier invokes a control: a monitoring capability to observe sequences of input images to detect errors, and out-of-distribution (OOD) inputs. Likewise, when errors in keypoint estimation inevitably

[†]A top event in BTD terminology corresponds to a hazard in FHA.

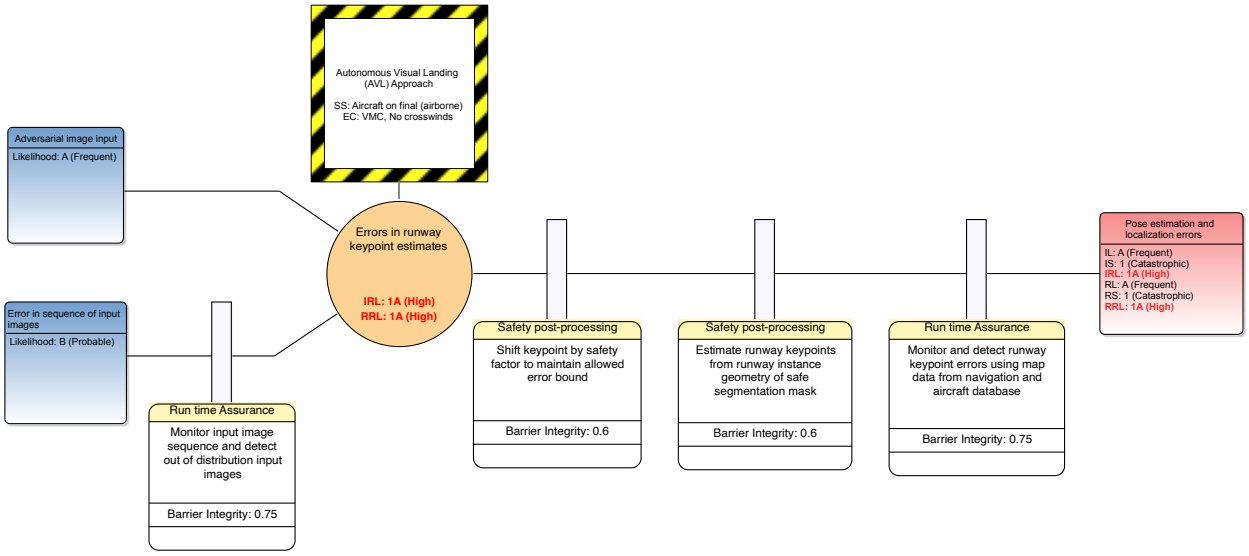


Fig. 5 BTD of a risk scenario for keypoint estimation errors in the AVL example.

occur, additional mitigations are to be invoked, including:

- Safety Post-processing, which involves two controls: (i) shifting keypoints by a safety factor such that they are within a predetermined error bound of the ground-truth keypoint; and (ii) estimating new, corrected, keypoints from the runway instance that is enclosed by a safe segmentation mask; and
- Run time Assurance, which involves monitoring and detecting keypoint errors that may persist despite safety post-processing, by using the map data input from the *Navigation and Aircraft Database*.

The intended risk reduction functionality of barriers and their controls can be associated with requirements, which can be added directly to the relevant diagram elements in AdvoCATE. Those in turn populate the requirements log (Section V), which also contains additional relevant information.

BTDs in AdvoCATE have an underlying risk assessment model [7]. In effect, it gives an assessment of the initial and residual risk levels for each consequence event based on: (i) its respective initial and residual severities; (ii) and the probabilities of occurrence of its precursor threat events; and (iii) the integrities of the mitigating barriers and controls.

Like requirements, barriers and controls can both be associated with evidence that substantiates the associated assumptions, e.g., about their integrity, and the probabilities of their precursor threats. This evidence, in turn, populates the evidence log (Section VII), where additional relations between evidence items can also be provided.

B. Safety Architectures

There can be a plurality of risk scenarios for a given system and its operations. By constructing BTDs of these risk scenarios, each modeling the different causal chains of events and the associated mitigations, and composing those risk scenarios, a new, larger, overarching model can be formed which we refer to as the *safety architecture*. It specifies at a system level:

- the collection of mitigations used to manage all the identified hazards, and their causes and effects; and
- the collection of circumstances (scenarios) under which those mitigations are invoked.

Fig. 6 gives a zoomed-out fragment of the AVL safety architecture. The shaded rectangular region reflects the portion of the safety architecture that we have modeled as the BTD in Fig. 5.

Just as system-level scenarios can be modeled, we can create scenario models of the mitigation mechanisms in AdvoCATE. In other words, these are lower-level scenarios focusing on barrier functions that are deployed in the wider system. In fact, the BTD in Fig. 5 is actually a barrier-focused scenario model and the barrier function in question is Localization, of which Pose Estimation is a sub-function. In Fig. 6, this relationship of the barrier-focused scenario to the higher-level scenario is shown as an *escalation* link from the (consequence) event to a (higher-level) barrier in the safety architecture. The semantics are that the effect Pose Estimation and Localization Errors (the node labeled E) in the barrier-focused scenario defeats the Localization barrier (the node labeled L) at a system level. That barrier, in turn, is meant to recover from an unstable final approach (the node labeled U) in the system-level risk scenario, without

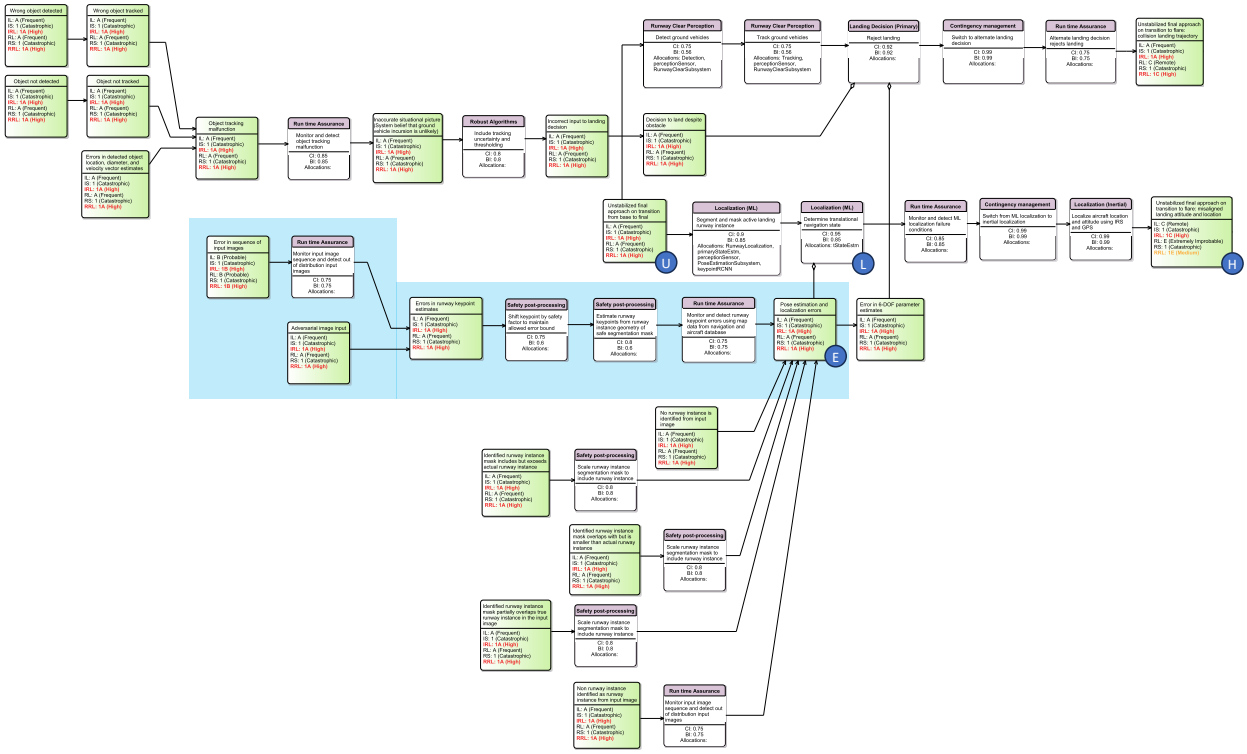


Fig. 6 Safety architecture fragment for the AVL example. The shaded rectangular region corresponds to the risk scenario of Fig. 5.

which the risk of the unstable approach persisting (the node labeled H) will increase. For more details on the specifics of safety architecture development, see [7].

C. Phased Scenarios

The initiating and terminating events in a BTD can be further analyzed by extending the event chain in either direction, i.e., by specifying precursor events to the threats and successor events to the consequence event, as well as mitigations/barriers to address each of those events. This analysis (i.e., extending the event chain) can be performed in the same BTD or by creating new BTDs in which the event being analyzed is set either as a threat, top, or consequence event. Consequently, we can model and link scenarios corresponding to distinct operational phases. For instance, the safety architecture fragment of Fig. 6 is in fact for the *approach* phase, and in particular for the *final* leg (see Section II.A, and Fig. 7).

By creating and linking such safety architecture fragments for different phases of landing, we can: (i) understand how the risk profile for an operation can change across the phases, (ii) provide assurance of safety risk mitigation through the different phases of landing, and (iii) create a safety architecture for the landing operation from the point the ML-based functionality is invoked until the operation is completed.

In AdvoCATE, we can then abstract this safety architecture for the landing operation in the form of a *phases view*, as shown in Fig. 7. This view allows us to hierarchically model the phases of the landing operation, the specific legs that constitute each phase, and the portion of safety architecture that is applicable to those legs. The links between the legs and phases model how the consequence events of one phase initiate the risk scenario(s) of the subsequent phase(s). Effectively, the phases view gives us a mechanism to model, analyze, and provide assurance of phased missions.

V. Requirements Log

As with the hazard log, the *requirements log* has a tabular depiction in AdvoCATE. This contains, in addition to the requirements description statements, information on the type of requirements, their allocation (to the elements of

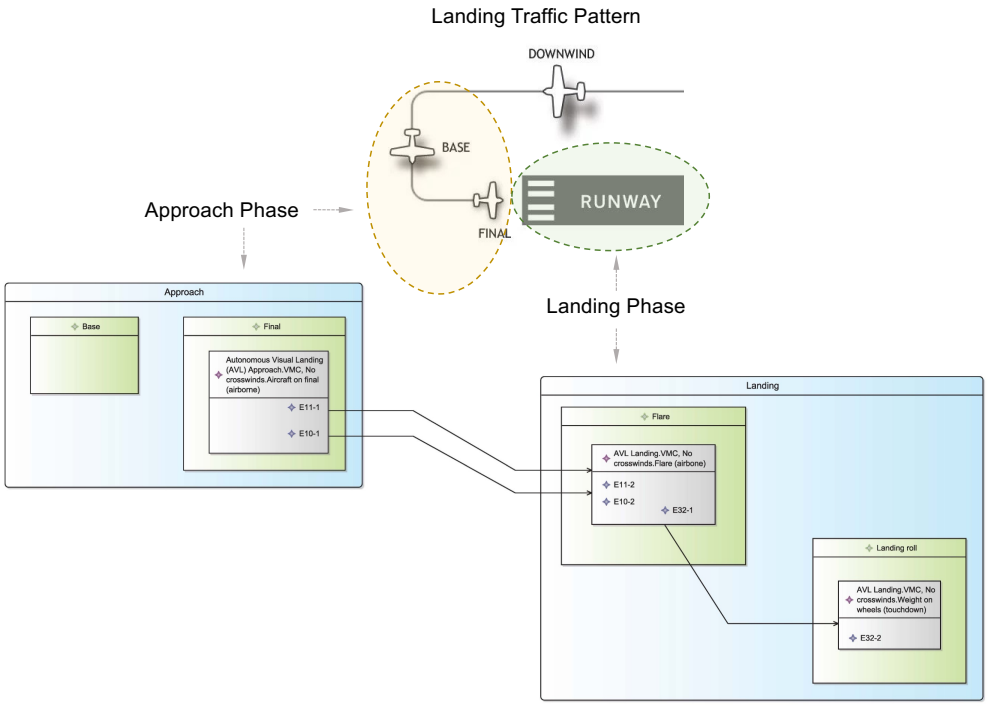


Fig. 7 Phases view in AdvoCATE modeling phased risk scenarios.

the functional and physical architectures), proposed *verification methods*, the results of applying those methods, i.e., *verification allocations*, and relations between the requirements. These aid assurance activities such as ensuring internal consistency amongst the requirements, traceability between requirements, verification methods, and the evidence that results from applying those methods, i.e., the *verification allocation*.

Within a safety-focused AC, requirements are mostly used to specify the safety system, i.e., how to implement the barriers and controls. Requirements can be entered directly in the requirements table, but can also be added directly within the respective elements of the BTDs. They will then be added to the requirements log. Requirements can also be added via the Mitigation Requirements view of the hazard log.

Fig. 8 shows (a screenshot of) an excerpt of the requirements log for the running example. An example functional requirement, as shown, concerns the capability of the runway localization function to provide a so-called *segmentation mask* of the active landing runway from input images, i.e., to provide an identification of all the pixels on an input image that correspond to the active landing runway. This requirement is *derived* from its parent requirement on PosEstm, which is both a functional and a safety requirement. Such associations between requirements are captured in the *relations* column of the requirements log, whereas the *allocations* column indicates the particular physical or functional architecture element with which the requirement is associated.

Also as shown in the figure, a requirement can have multiple verification methods, each of which can have multiple verification allocations (although Fig. 8 shows each verification method associated with a single verification allocation). Verification allocations are documented in more detail in the evidence log (described in Section VII).

VI. Capturing Assurance Rationale

AdvoCATE facilitates creating structured arguments using the Goal Structuring Notation (GSN) as a means of capturing the assurance rationale required to connect elements of an AC, such as (i) linking evidence artifacts to claims; (ii) justifying assertions about evidence artifacts; (iii) substantiating assumptions used in the risk analysis, (iv) reasoning about the effectiveness of barriers in the safety architecture, (v) clarifying the context in which the claims made and the substantiating evidence supplied should be interpreted, and so on.

Assurance arguments for complex functionality may not only leverage reasoning at different levels of the system hierarchy, but also may require reasoning about a diversity of related concerns. In part, this depends on the level of risk

ID	Description	Type	Allocation	Verification Method	Verification Allocation	Relations
PE-SF21-R0001	The pose estimation subsystem shall provide the capability to detect, classify, and localize the active landing runway from a stream of HD images of the airport landing environment	Functional & Safety	PoseEstimationSubsystem: SS2: Pose Estimation	VM1: Software Simulation	poseEstmFnSwSimRslt: Pose Estimation and Localization function software simulation results	Derives PE-SF211-R0003
						Assumes OPRENV-R0001
						Assumes OPRENV-R0002
				VM2: HIL Simulation	poseEstmFnHILSimReslt: Pose Estimation and Localization function hardware-in-the-loop (HIL) simulation results	Assumes OPRENV-R0003
						Assumes OPRENV-R0004
PE-SF211-R0003	The runway localization function shall segment and mask active landing runways from input images	Functional	RunwayLocalization: SF2.1.1: Runway Localization	VM3: ML item verification on test dataset	poseEstmVerfOnTestDataRslt: Verification of Pose Estimation item on test dataset	Derived from PE-SF21-R0001
				VM4: ML Item verification on simulated data	poseEstmVerfOnSimDataRslt: Verification of Pose Estimation item on simulated data	Parent of PEL-SF21-R0001
INL-R0002	Inertial localization shall continuously determine the attitude, location, and velocity of the ownship	Safety	aircraft: Aircraft	VM6: Flight testing - experimental trials	fltTestRsltINS: Inertial navigation flight testing experimental results	
CNTMGR-R0004	The contingency manager shall switch from the pose estimation and localization function to inertial localization when the pose estimate fault flag is set	Safety & Functional	aircraft: Aircraft	VM7: Architecture model verification	contMgrMdlVerfRslt: AVL architecture model verification results for contingency manager switching properties	
				VM8: Contingency manager subsystem verification	contMgrSubSysVerfRslt: Contingency manager subsystem verification results	

Fig. 8 AdvoCATE screenshot: Excerpt of the requirements log for autonomous visual landing.

posed by that functionality. For instance, safety of functionality that poses a low level of risk may be substantiated by a *shallow* argument that relies only on evidence from, for example, system-level verification. On the other hand, when increased assurance is necessary for functionality that poses higher risk, the assurance argument may require additional evidence from lower levels of the system hierarchy, e.g., that the design of the functionality meets the requirements, that the implementation satisfies the constraints of the design and meets the requirements, and that the integration of the implementation into the wider system meets the system level requirements.

In general, the comprehensibility of such arguments can be improved by providing a *big picture* overview of the assurance argument that abstracts from specific details. AdvoCATE provides functionality for creating such abstractions, as well as detailed arguments, as discussed next.

A. Argument Architecture

The *argument architecture* is a high-level tree structure that relates individual sub-arguments, each of which address a defined assurance scope. Fig. 9 shows the argument architecture for PosEstm, abstracting the overall rationale for the claim that PosEstm is safe for use (shown as the root of the tree), wherein each node of the tree represents abstractions of concrete arguments, as shown in Fig. 10 and Fig. 11.

In the figure, each node label indicates the scope of the sub-argument that the node encapsulates, and its color corresponds to system hierarchy as follows: function/subsystem (blue), sub-function/item (yellow), model (green), data (mauve). Links between nodes represent *support* relations, i.e., an argument in a lower-level node supports the argument of the higher-level node(s) to which it is connected. For instance, the argument for satisfaction of the functional safety requirements is supported (in part) by the argument that the Perspective n-Point (PnP) solver is accurate. The former is an argument at the system level, while the latter is an argument about the sub-function.

The notation used for each node of the argument architecture in Fig. 9 is due to [25], and it indicates the *splitting* of a large argument structure into smaller portions. Thus, the argument architecture here is created from a collection of such *split* sub-arguments. It is worth noting that such a structure can be seen as a type of *module view* of an overall argument, with each node of the view corresponding to a single argument module [13], that contains a single argument.

B. Structured Arguments

Fig. 10 shows the top-level of the argument architecture (highlighted by the dotted oval region) abstracting the decomposition of the main safety claim (the rectangular *goal node* G30) into three sub-claims (the goal nodes G29, G31, G32). The necessary context of the main claim also has been clarified (as shown as the rounded rectangular nodes C19, C57, C20). Additionally, the relevant assumptions (shown as the oval nodes A4, A6, and A7) also are stated.

The main safety claim as shown states that PosEstm is acceptably safe for use. Here, "acceptably safe" is defined in terms of the unacceptable outcomes (to which PosEstm contributes) not occurring more frequently than the rate corresponding to the safety target considered acceptable for those outcomes. That safety claim has been decomposed into

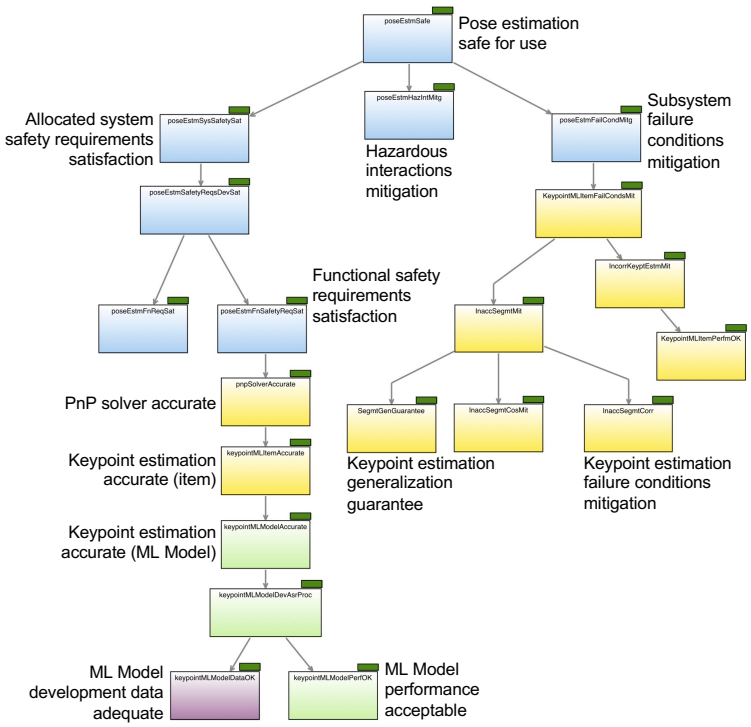


Fig. 9 Argument architecture for PosEstm showing the high-level structure of the overall safety argument.

lower-level sub-claims, where *decomposition* represents an inference rule of the argument (shown as the parallelogram node S14). The resulting sub-claims are:

- G31: PosEstm satisfies its allocated system (safety) requirements;
- G32: All identified failure conditions of PosEstm are sufficiently mitigated; and
- G29: All identified hazardous interactions of PosEstm are sufficiently mitigated.

The first two relate to the intended behavior and the corresponding failure conditions not leading to an unacceptable outcome, i.e., satisfying its functional safety requirements, whereas the third relates to PosEstm not exhibiting unintended behavior with unacceptable consequences.

Fig. 11 similarly shows an argument fragment pertaining to a claim of *generalization*, abstracted by the leaf node of the argument architecture highlighted by the dotted circle. Specifically, the main claim of this argument (goal node G101) invokes the mean average precision (mAP) metric of the item implementing ML-based keypoint estimation. This claim is then decomposed into to sub-claims about the values of those metrics as determined from verification in a real and unconstrained environment (G51), in a real but controlled environment (G52), and in simulation (G53). For each of those sub-claims, the necessary contextual information (C36, C37) and assumptions (A2, A3) also have been stated.

Here, we have only given a brief description of each of the arguments for clarity, although more details on the specifics of the overall assurance argument can be found in [24].

Additionally, although not described here, AdvoCATE also provides functionality to specify *argument patterns* [12], which abstract arguments in an orthogonal way to argument modules. Specifically, patterns abstract the content of an argument and its reasoning structure, whereas modules allow encapsulation of one or more arguments addressing a common assurance concern. Patterns can be composed [26] and either reused in other ACs, or used in the current AC to raise the level of abstraction at which reasoning is represented. Ultimately, AdvoCATE supports *round-trip engineering* of arguments, whereby arguments are constructed from a combination of instantiated patterns and bespoke argumentation; edits can be made to the generated arguments, and those edits optionally propagated back to the source patterns.

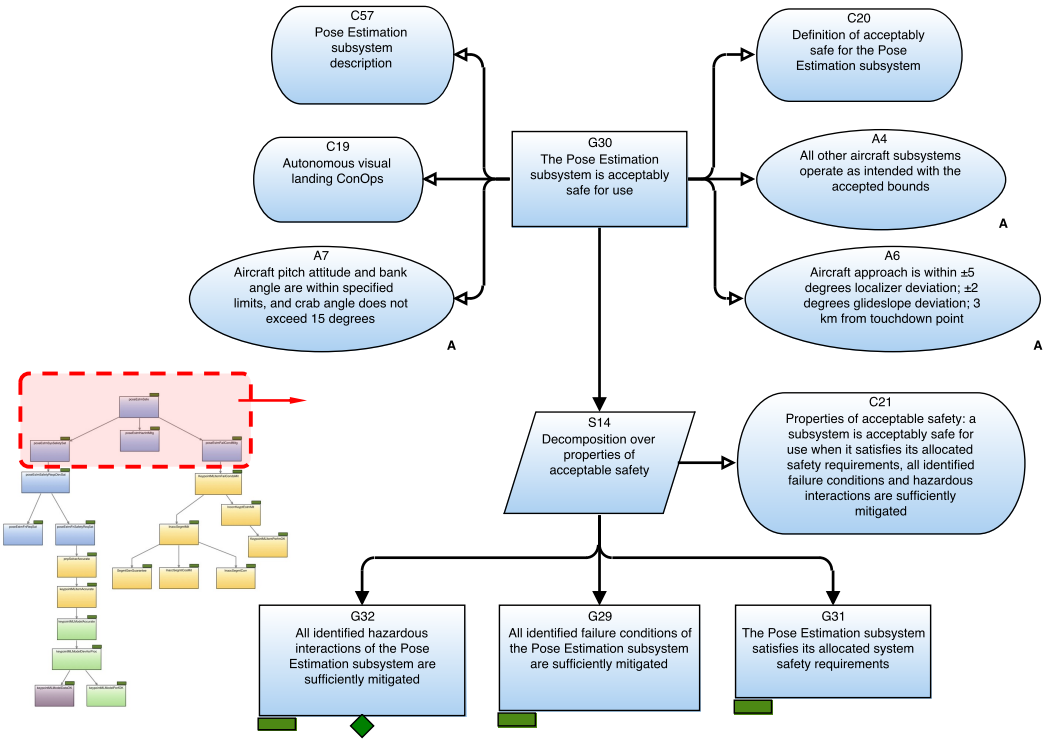


Fig. 10 Fragment of top-level GSN safety argument structure for PosEstm created using AdvoCATE.

VII. Assurance Substantiation with Evidence

An AC consists of *reasoning* (e.g., risk analysis and arguments) and substantiating *evidence*. Evidence is used in various places in the ACs we create in AdvoCATE, principally as substantiating *solution nodes* in a GSN structure, and to justify data in the risk analysis.

Evidence refers to one or more lifecycle artifacts (the products of a process, method, or tool) accompanied by verifiable *evidence assertions* (qualitative or quantitative statements of fact), which directly or indirectly provide confirmation that a claim in an AC can be considered to be true; for example, that a product or service meets its requirements (and, therefore, can be relied upon). An evidence assertion is thus an attribute of a lifecycle artifact, rather than the artifact itself, and serves to link that artifact to the argument where it is invoked as evidence.

Here, we use *evidence* in a general sense to encompass all external artifacts that are linked to the AC in a justifying or contextual role; for example, verification artifacts such as test data, simulation results, and the application of formal methods; manufacturer data sheets, formal specifications and models; and documentation such as user guides, test plans, concepts of operations, etc. Evidence can be obtained from development and verification activities, and can be obtained both at design time, and after deployment during operations. In practice, evidence tends to be obtained in parallel with the creation of the AC, as system development and verification activities proceed.

A. Evidence Log

In AdvoCATE, the *evidence log* documents all the evidence used in the AC, as well as any evidence for which there are plans for future use. We refer to an individual item of evidence as an *evidence artifact* (or simply "evidence", for short). An evidence artifact can be designated as *pending* while the AC is still being developed, to indicate that it is planned to be obtained. The evidence log also captures key attributes of an evidence artifact, such as its description, its *type* (e.g., measurement, or analysis), its provenance, status (e.g., pending, to-be-verified, or verified), as well as its *dependencies* with other evidence artifacts. This information is also summarized in the evidence dashboard, part of which is shown in Fig 4.

Fig. 12 shows the argument architecture for the running example, annotated with a selection of evidence artifacts associated with different nodes, i.e., the arguments that each node abstracts. As shown, evidence comprises lifecycle

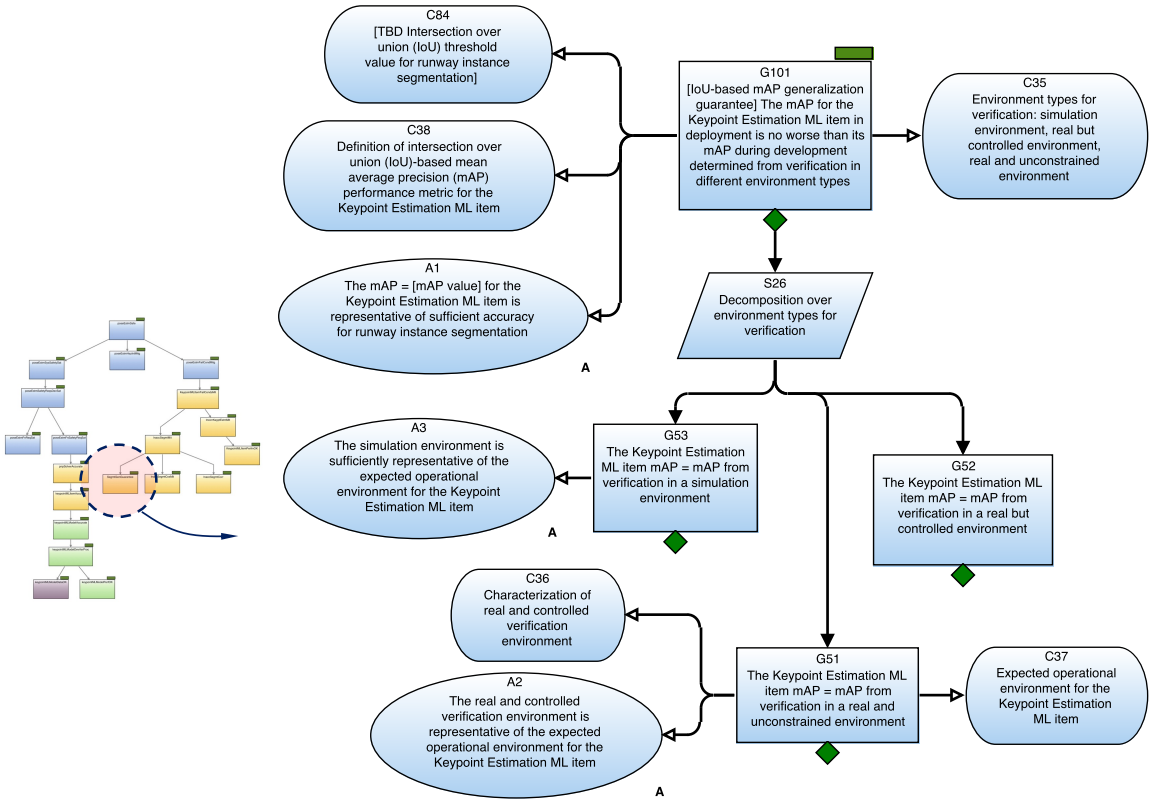


Fig. 11 Fragment of assurance argument for ML-based keypoint estimation generalization in the AVL example.

data obtained from system development, safety assessment, learning, and implementation processes. The evidence artifacts have been grouped here according to their provenance. Moreover, depending on the evidence assertion, an evidence artifact may be invoked in different sub-arguments of the overall argument.

If an evidence artifact is self-contained, then it will have no dependencies on other evidence. Often, however, evidence is created from other evidence artifacts using some tool—for example, simulation results are created using a simulator from a model and an initial configuration. Such relations between evidence artifacts can be modeled in AdvoCATE via an *evidence dependency graph*, and also depicted graphically. Fig 13 shows a simple fragment of such a graph for our running example. As shown in the figure, each node of the graph represents a concrete evidence artifact giving its attributes, while the annotated links indicate the dependencies. For instance, the interface specification for the RwyClrIs both a part of the associated subsystem description, and derived from the system architecture specification of the AVL function.

Just as evidence can be associated with arguments in order to substantiate claims, so arguments can also be associated with evidence in order to justify properties of evidence. AdvoCATE enforces circularity checks to prevent cyclic associations and arguments. Focusing on evidence lends itself to a bottom-up style of argument creation, where we first identify the key evidence, which assurance claims the evidence directly supports (i.e., the evidence assertions), and then work back to determine how the evidence was created, and which assumptions it relies on; while a top-down style will start with assurance claims and work towards the evidence that substantiates those claims. In practice, often a combination of these styles is used.

B. Tools Log

The *tools log* documents external tools and methods which are used to create evidence artifacts. Such tools can be informal methods, such as code review and inspection, or the application of formal methods, such as a static code analyzer, to other artifacts. In both cases, the tool or technique is characterized in terms of its inputs and outputs, each of which is either a type of evidence artifact or a primitive type. A *tool specification* gives assumptions on the inputs and guarantees on the outputs. A *tool use* represents an application of a tool to concrete evidence artifacts. The tools

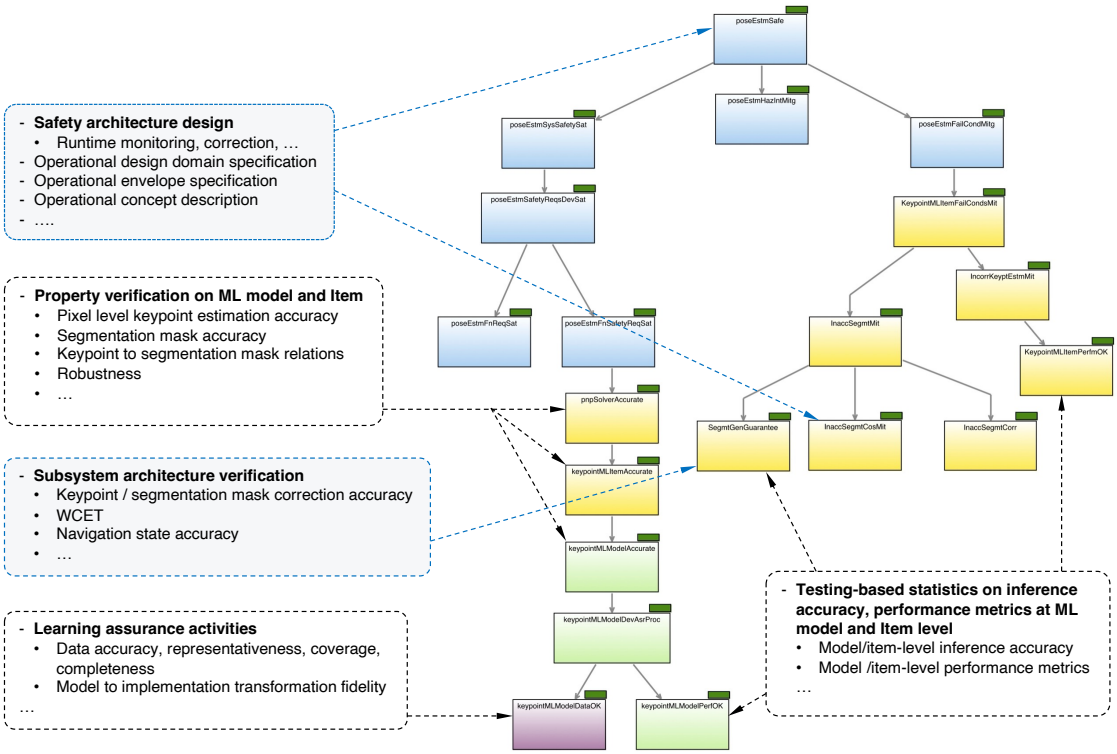


Fig. 12 Selection of evidence artifacts supporting the PosEstm safety argument.

and evidence logs are thus interrelated and capture the chain of dependencies through which evidence is constructed, along with the supporting assumptions.

For our running example, some suitable tools that could be used include runtime monitors to provide evidence that properties continue to hold during operations, test case generators to create suites of test cases, and theorem provers to prove soundness of model implementations. By associating argument patterns with tool specifications, tool uses can be used in generate argument fragments which reason about the properties of the tool outputs (the tool guarantees providing the corresponding evidence assertions). Just as tools can be recursively chained together, so their patterns can be composed to construct an argument that reasons over the construction of the evidence that is ultimately constructed. See [27] for an example of creating an argument from a tools log.

VIII. Conclusion

We have described the current suite of functionality for AdvoCATE, an assurance case automation toolset based on a metamodel that integrates a variety of assurance concepts that enable consistency and traceability. Since the tool supports a variety of modeling notations, different users are free to pick and choose whatever suits their needs. There are users who solely create arguments, others who only model risk scenarios using Bow Tie Diagrams (BTDs), and some who only use the hazard log feature.

ACs complement formal methods, but AdvoCATE, itself, is rigorous, rather than formal, and the tool enforces approximately 100 well-formedness rules. As ACs are primarily a means of communication to assurance stakeholders, their value lies in assembling, integrating, and justifying evidence, rather than formal analysis and, instead, formal evidence generated by tools can be integrated [27].

In ongoing work, we are extending the tool to better support operational assurance. This differs from our earlier work on dynamic assurance, where we looked at dynamically generating arguments [23, 28]. Our focus now is on integration of measures, metrics, indicators into arguments [29] as a means of determining the operational validity of an assurance case. We are currently reengineering the third generation of the tool using modern cloud-based implementation frameworks, in order to directly support collaborative development, assurance case assessment, and more naturally support tool interoperability.

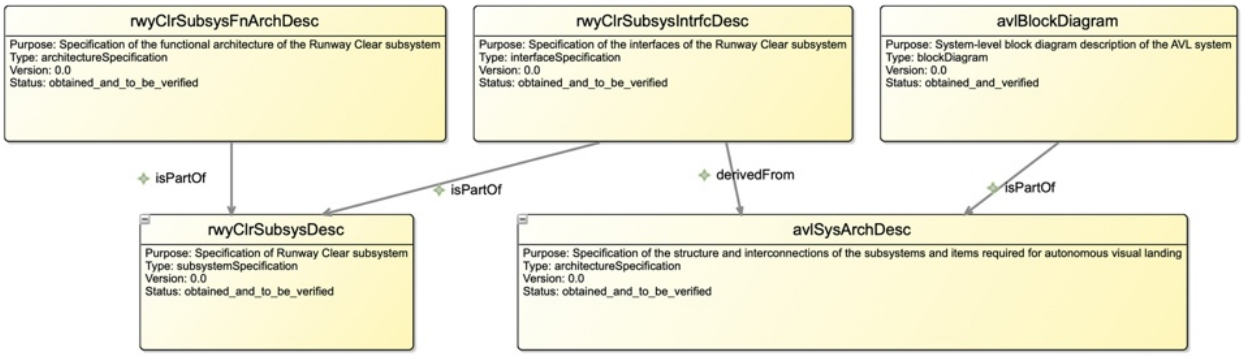


Fig. 13 AdvoCATE Screenshot: Evidence Dependency Graph

Acknowledgments

This work was performed under Contract No. 80ARC020D0010 with the National Aeronautics and Space Administration (NASA), with support from the System-wide Safety project, under the Airspace Operations and Safety Program of the NASA Aeronautics Research Mission Directorate; the DARPA Assured Autonomy Program under contract FA8750-18-C-0094; and the FAA Aviation Research Software and Systems Program (ARSS) for Complex Digital Systems, under contract 692M15-23-T-00014.

References

[1] Dezfuli, H., Benjamin, A., Everett, C., Smith, C., Stamatelatos, M., and Youngblood, R., "Volume 1: System Safety Framework and Concepts for Implementation," NASA System Safety Handbook NASA/SP-2010-580, NASA, Nov. 2011. URL https://ntrs.nasa.gov/citations/20120003291.

[2] Dezfuli, H., Benjamin, A., Everett, C., Smith, C., Stamatelatos, M., and Youngblood, R., "Volume 2: System Safety Concepts, Guidelines and Implementation Examples," NASA System Safety Handbook NASA/SP-2014-612, NASA, Nov. 2014. URL https://ntrs.nasa.gov/citations/20150015500.

[3] Forsbacka, M. J., and Helton, D. M., "Evolution of NASA's Nuclear Flight Safety Program to Infuse Risk Leadership and Assurance Framework Concepts," *Journal of Space Safety Engineering*, Vol. 10, No. 1, 2023, pp. 95–102. https://doi.org/10.1016/j.jsse.2022.11.003.

[4] Dezfuli, H., Everett, H., Youngblood, R., and Forsbacka, M., "Implementing an Objectives-Driven, Risk-Informed, and Case-Assured Approach to Safety and Mission Success at NASA," *Proceedings of the 17th International Conference on Probabilistic Safety Assessment and Management (PSAM) & Asian Symposium on Risk Assessment and Management (ASRAM)*, Sendai, Japan, 2024. URL https://ntrs.nasa.gov/citations/20240006733.

[5] The Assurance Case Working Group (ACWG), "Goal Structuring Notation Community Standard Version 3," UK Safety Critical Systems Club, SCSC-141C, May 2021. URL https://scsc.uk/r141C:1.

[6] UK Ministry of Defence (MOD), "Safety Management Requirements for Defence Systems," Defence Standard 00-56, Issue 7, 2017.

[7] Denney, E., Pai, G., and Whiteside, I., "The Role of Safety Architectures in Aviation Safety Cases," *Reliability Engineering and System Safety*, Vol. 191, 2019. https://doi.org/10.1016/j.ress.2019.106502.

[8] Denney, E., and Pai, G., "Tool Support for Assurance Case Development," *Journal of Automated Software Engineering*, Vol. 25, No. 3, 2018, pp. 435–499. https://doi.org/10.1007/s10515-017-0230-5.

[9] Denney, E., Pai, G., and Pohl, J., "AdvoCATE: An Assurance Case Automation Toolset," *SAFECOMP 2012 Workshops—Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR)*, Lecture Notes in Computer Science (LNCS), Vol. 7613, edited by F. Ortmeier and P. Daniel, Springer-Verlag, 2012. https://doi.org/10.1007/978-3-642-33675-1_2.

[10] Denney, E., Pai, G., and Pohl, J., "Heterogeneous Aviation Safety Cases: Integrating the Formal and the Non-formal," *17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Paris, France, 2012, pp. 199–208. https://doi.org/10.1109/ICECCS20050.2012.6299215.

[11] Denney, E., Pai, G., and Whiteside, I., "Hierarchical Safety Cases," *Proceedings of the 5th NASA Formal Methods Symposium*, LNCS, Vol. 7871, edited by G. Brat, N. Rungta, and A. Venet, Springer-Verlag, 2013, pp. 478–483. https://doi.org/10.1007/978-3-642-38088-4_37.

[12] Denney, E., and Pai, G., "A Formal Basis for Safety Case Patterns," *Computer Safety, Reliability and Security. SAFECOMP 2013*, Lecture Notes in Computer Science (LNCS), Vol. 8153, edited by F. Bitsch, J. Guiochet, and M. Kaâniche, Springer, 2013, pp. 21–32. https://doi.org/10.1007/978-3-642-40793-2_3.

[13] Denney, E., and Pai, G., "Towards a Formal Basis for Modular Safety Cases," *Computer Safety, Reliability, and Security. SAFECOMP 2015*, Lecture Notes in Computer Science (LNCS), Vol. 9337, edited by F. Koornneef and C. van Gulijk, Springer, 2015, pp. 328–343. https://doi.org/10.1007/978-3-319-24255-2_24.

[14] Denney, E., Naylor, D., and Pai, G., "Querying Safety Cases," *Computer Safety, Reliability and Security. SAFECOMP 2014.*, Lecture Notes in Computer Science, Vol. 8666, edited by A. Bondavalli and F. D. Giandomenico, Springer, 2014, pp. 294–309. https://doi.org/10.1007/978-3-319-10506-2_20.

[15] Denney, E., and Pai, G., "Automating the Assembly of Aviation Safety Cases," *IEEE Transactions on Reliability*, Vol. 63, No. 4, 2014, pp. 830–849. https://doi.org/10.1109/TR.2014.2335995.

[16] Berthold, R., Denney, E., Fladeland, M., Pai, G., Storms, B., and Sumich, M., "Assuring Ground-based Detect and Avoid for UAS Operations," *Proceedings of the 33rd IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2014, pp. 6A1–1–6A1–16. https://doi.org/10.1109/DASC.2014.6979492.

[17] Denney, E., and Pai, G., "Safety considerations for UAS ground-based detect and avoid," *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016, pp. 1–10. https://doi.org/10.1109/DASC.2016.7778077.

[18] Matsuno, Y., "D-Case Communicator: A Web Based GSN Editor for Multiple Stakeholders," *Computer Safety, Reliability, and Security. SAFECOMP 2017*, Lecture Notes in Computer Science (LNCS), Vol. 10489, edited by S. Tonetta, E. Schoitsch, and F. Bitsch, Springer, 2017, pp. 64–69. https://doi.org/10.1007/978-3-319-66284-8_6.

[19] Diemert, S., Goodenough, J., Joyce, J., and Weinstock, C., "Incremental Assurance Through Eliminative Argumentation," *Journal of System Safety*, Vol. 58, No. 1, 2023, pp. 7–15. https://doi.org/10.56094/jss.v58i1.215.

[20] Roback, K., "Review of Potential Assurance Case Tool Options for DoD," Technical Report IDA D-33524/2, Institute for Defense Analyses, Alexandria, VA, January 2024. URL https://apps.dtic.mil/sti/trecms/pdf/AD1211550.pdf.

[21] Adelard, Part of NCC Group, "Assurance and Safety Case Environment (ASCE)," , 2025. URL http://www.adelard.com/asce/.

[22] Denney, E., Habli, I., and Pai, G., "Dynamic Safety Cases for Through-life Safety Assurance," *2015 IEEE/ACM 37th International Conference on Software Engineering (ICSE 2015)*, Florence, Italy, 2015, pp. 587–590. https://doi.org/10.1109/ICSE.2015.199.

[23] Asaadi, E., Denney, E., Menzies, J., Pai, G., and Petroff, D., "Dynamic Assurance Cases: A Pathway to Trusted Autonomy," *IEEE Computer*, Vol. 53, No. 12, 2020, pp. 35–46. https://doi.org/10.1109/MC.2020.3022030.

[24] Denney, E., and Pai, G., "Assurance-driven Design of Machine Learning-based Functionality in an Aviation Systems Context," *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, 2023. https://doi.org/10.1109/DASC58513.2023.10311282.

[25] Spriggs, J., *GSN - The Goal Structuring Notation: A Structured Approach to Presenting Arguments*, Springer London, 2012.

[26] Denney, E., and Pai, G., "Composition of Safety Argument Patterns," *Computer Safety, Reliability and Security. SAFECOMP 2016.*, Lecture Notes in Computer Science, Vol. 9922, edited by A. Skavhaug, J. Guiochet, and F. Bitsch, Springer, 2016. https://doi.org/10.1007/978-3-319-45477-1_5.

[27] Sljivo, I., Denney, E., and Menzies, J., "Guided Integration of Formal Verification in Assurance Cases," *Formal Methods and Software Engineering. ICFEM 2023.*, Lecture Notes in Computer Science (LNCS), Vol. 14308, edited by Y. Li and S. Tahar, Brisbane, Australia, 2023. https://doi.org/10.1007/978-981-99-7584-6_11.

[28] Denney, E., Habli, I., and Pai, G., "Dynamic Safety Cases for Through-life Safety Assurance," *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, IEEE Press, Florence, Italy, 2015, pp. 587–590. https://doi.org/10.1109/ICSE.2015.199.

[29] Denney, E., and Pai, G., "Reconciling Safety Measurement and Dynamic Assurance," *43rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2024)*, Lecture Notes in Computer Science (LNCS), Vol. 14988, edited by A. Ceccarelli, M. Trapp, A. Bondavalli, and F. Bitsch, Springer, 2024. https://doi.org/10.1007/978-3-031-68606-1_4.