AFRL-RI-RS-TR-2023-162

QUASAR: QUANTIFIABLE ASSURANCE CASES FOR TRUSTED AUTONOMY

KBR WYLE LLC

SEPTEMBER 2023

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE	**2. REPORT TYPE**	**3. DATES COVERED** START DATE	END DATE
SEPTEMBER 2023	FINAL TECHNICAL REPORT	APRIL 2018	JANUARY 2023

4. TITLE AND SUBTITLE
QUASAR: QUANTIFIABLE ASSURANCE CASES FOR TRUSTED AUTONOMY

5a. CONTRACT NUMBER FA8750-18-C-0094	**5b. GRANT NUMBER** N/A	**5c. PROGRAM ELEMENT NUMBER** 62303E
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER** R2HG

6. AUTHOR(S)
Ewen Denney, Rebecca Lee, Ganesh J. Pai, and Irfan Šljivo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
KBR Wyle LLC (previously SGT, Inc.)
Building N269, Room 234, Mailstop 269-2, NASA Ames Research Center
Moffett Field CA 94035

8.PERFORMING ORGANIZATIONREPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA I2O
675 North Randolph St
Arlington VA 22203-2114

10.SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11.SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-RI-RS-TR-2023-162

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# AFRL-2023-0749/AFRL-2023-3593.
Date Cleared: 31 August 2023

13. **SUPPLEMENTARY NOTES**

14. ABSTRACT
The goal of QUASAR was to develop techniques and tool support for the construction of dynamic assurance cases (DACs) that address both static design-time concerns and dynamic operational concerns and, for the latter, to provide quantifiable and executable measures of confidence in appropriate assurance properties of the target platforms. In particular, the project focused on applying these techniques to learning-enabled components (LECs) that comprise key parts of autonomous systems, and to that end, we used these techniques to construct platform-specific assurance cases in collaboration with our TA4 partners, Boeing (air domain) and Northrop Grumman (undersea domain).

15. SUBJECT TERMS
Dynamic Assurance Case, Assured Autonomy, Assurance Case, Assuring Learning-enabled Components

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**
a. REPORT U	**b. ABSTRACT** U	**C. THIS PAGE** U	**SAR**	**107**

19a. NAME OF RESPONSIBLE PERSON	**19b. PHONE NUMBER** *(Include area code)*
WILLIAM E. MCKEEVER	**N/A**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 SUMMARY

This report describes work done by KBR as a Technical Area (TA) 3 performer of the QUASAR (Quantifiable Assurance Cases for Trusted Autonomy) project in the Assured Autonomy Program.

The goal of QUASAR was to develop techniques and tool support for the construction of dynamic assurance cases (DACs) that address both static design-time concerns and dynamic operational concerns and, for the latter, to provide quantifiable and executable measures of confidence in appropriate assurance properties of the target platforms. In particular, the project focused on applying these techniques to learning-enabled components (LECs) that comprise key parts of autonomous systems, and to that end, we used these techniques to construct platform-specific assurance cases in collaboration with our TA4 partners, Boeing (air domain) and Northrop Grumman (undersea domain).

We give an overview of these DACs and describe how they address the challenge problems provided by TA4. The static components of a DAC were constructed for the Autonomous Visual Landing air domain challenge problem (Boeing, CP3.1) and, in part, for the collision avoidance undersea domain challenge problem (Northrop Grumman, CP4). The former concerns assurance of aircraft navigation state estimates produced by an LEC used for autonomous landing based on optical sensing, whereas the latter concerns assurance of an LEC that filters the input of a forward-looking sonar intended for detecting obstacles in the forward path of an autonomous underwater vehicle (AUV). The main assurance artifacts that constitute the static DAC components include a hazard log (documenting the identified hazards), a requirements log (capturing the related safety requirements), a safety architecture (representing the system-level organization of the identified safety mitigations), assurance rationale captured in the form of structured arguments, and an evidence model describing the evidence necessary to substantiate the safety claims made.

The executable part of a DAC that provides real-time confidence is given as an assurance measure. A high-level assurance measure serves as an assurance metric for the overall mission. To formulate this, we first identify mission objectives, then link those objectives to events in the safety architecture, using the structure of the safety architecture to form a probabilistic graphical model. A mission objective is a quantifiable goal that is necessary for overall mission success. A low-level assurance measure provides uncertainty quantification for individual system components, such as LECs, or sensors that produce a real-time signal. We describe the techniques used to construct the assurance measures for the undersea domain challenge problems (CP4 – collision avoidance, CP6 – operating under degraded modes), and the air domain challenge problem (CP3.1 – autonomous visual landing), respectively.

Claims in an assurance case are ultimately substantiated by evidence, which can take many forms, but one important source of evidence is provided by formal verification of the target system. We describe how we reason about such verification as part of the overall platform assurance cases, by developing assurance models of formal verification techniques in the form of tool specifications. We show how we modeled the application of the tools developed by our TA1 collaborators from UC Berkeley (VerifAI, applied to Boeing CP 1.1 and CP 2.1) and Imperial College (Venus, applied to Boeing CP 2.2) to the corresponding challenge problems, and created assurance arguments from this.

We describe the technology developed to create these assurance cases that was implemented in the AdvoCATE assurance case tool. First, we describe the assurance case metamodel

that defines the structure of assurance cases supported by AdvoCATE. The metamodel enforces the overall structure of the assurance case, but additional well-formedness and internal consistency properties have been implemented as so-called validations, and we give an overview of several of these. The core AdvoCATE assurance model can be extended by user-defined ontologies. These enhance our model-based approach to assurance by allowing the formulation of domain-specific extensions to the underlying models. We map the core DAC components into a derived ontology, and allow domain-specific additions. We describe a query language that can be used to specify and extract assurance artifacts within the assurance case that satisfy given properties. Queries can use both the core metamodel and ontological extensions.

As assurance cases grow in size and complexity, it becomes increasingly challenging both to manage their development, and to understand and assess them. Given the communicative role of assurance cases, the latter problem is probably the more serious. In addition to user-definable views (developed in earlier phases of the project), AdvoCATE now additionally provides several built-in structuring mechanisms that allow assurance cases to be structured and abstracted into meaningful fragments. We describe an argument architecture (splits) view, which allows large arguments to be decomposed into meaningful fragments, the provenance view, which provides a graphical representation of the assurance artifacts from which an argument is created, as well as other associated assurance artifacts and their relation to the argument, and the phases view, which shows the high-level hierarchical architecture of the assurance architecture.

In addition to these graphical views, we also describe tabular views. A table view specification language allows the generation of tabular views that display designated assurance artifacts from the assurance case in tabular form. A variation of such tables is the traceability matrix, which allows users to depict relations between two sets of artifacts, such as hazards and requirements. Besides static views, it is also possible to define dynamic views of arguments. These can be used to update argument status in real-time, while maintaining argument structure. Status can be used for different purposes, such as showing which part of an argument is currently "active" (that is, which branches reason about the effectiveness of the safety measures currently being applied). It can also be used to display confidence in argument claims.

To that end, we extended the argument view mechanism that was developed in earlier phases of the project to be able to access real-time values provided to AdvoCATE via external ports. The views are updated as data is received. By integrating this with the output of assurance measures, we are able to use these dynamic argument views to visualize LEC confidence and its bearing on an argument. Dynamic views are specified by associating formulas with selected argument nodes, and mapping these formulas to visualizable node properties, such as colors. Both table view specifications and dynamic argument views leverage the query language.

Lastly, we conclude by discussing scope for further automation, in particular for streamlining the generation of assurance measures from the other assurance artifacts, and high-level modeling of formal verification tools that could be used to re-generate assurance artifacts when the context of the assurance case changes, as well as the potential for additional tool support to improve the overall assurance lifecycle.

2.0 INTRODUCTION

The problem area addressed by this research project is the development of dynamic assurance cases (DACs) for learning-enabled cyber-physical systems (LE-CPSs). To ensure safety and mission success when using LE-CPSs, assurance of their *fitness for purpose* is required, i.e., their behavior is as intended, is predictable, and that their functions and services can be trusted. Moreover, assurance is required that upon deviating from the required envelope of behaviors and level of fitness, appropriate countermeasures can feasibly intervene. These requirements pose unique challenges from the standpoint of assurance technologies. Specifically, the state of the art in *assurance case* (AC) technology is insufficient in its current capabilities to address LE-CPS assurance, owing to their:

- limited expressivity, i.e., critical assurance information not captured in the form of arguments is obscured;
- limitations in scope, e.g., life-cycle/system safety concerns are not sufficiently treated, while formal assurance cases largely concern only assurance of functional correctness;
- inherently static nature, due to which they are ill-suited for operational assurance, which is inherently dynamic; and
- misplaced focus in assurance quantification, by not considering the quantification of system properties. Moreover, the automated construction, manipulation, assessment, and evolution of ACs consistent with system adaptation require a formal basis.

The overarching goal is to develop, implement, and demonstrate an integrated framework, along with supporting infrastructure—i.e., languages, tools, and technologies—to produce DACs that give (quantified) justification for confidence in the design and operation of LE-CPSs, both during design-time and then continually during operations. The focus is on providing (and updating) assurance of safety and functional correctness whilst accounting for the evolution of both the system and its environment.

3.0 METHODS, ASSUMPTIONS AND PROCEDURES

3.1 Dynamic Assurance

For the subsequent discussion, the ensuing terminology is relevant: *assurance* is the provision of justified confidence that an *item*—i.e., a component, system, or service—possess the required assurance properties. An *assurance propert*y is a logical, possibly probabilistic characteristic associated with assurance concerns (or assurance objectives), i.e., functional capabilities and dependability attributes. An *assurance claim* results from applying one or more assurance properties to a particular item. Practically, an assurance claim can be considered to be equivalent to an assurance requirement that has been (or will eventually be) substantiated by concrete evidence.

First, we introduce a concept of *assurance measure*, which characterizes the extent of confidence in an assurance property through a probabilistic quantification of uncertainty. It encodes a baseline level of acceptable risk for a deployed LE-CPS, based on a suitably abstract, probabilistic model of the same. Our concept of dynamic assurance integrates assurance measures into an LE-CPS, to facilitate run-time confidence assessment of its assurance properties.

3.1.1 Assurance Measures

An assurance measure provides real-time feedback to autonomous systems on the uncertainty in various systems components. The work completed in this program has been focused on assuring LECs, but assurance measures can be developed for any system that provides real-time signal output. During the program, we have developed a multi-level approach to assurance measures. The *high-level assurance measure* is an assurance metric for the mission. For this we identify *mission objectives*, link those objectives to events in the *assurance architecture* (see Section 3.3.2), and use the relationships in the safety architecture to form a probabilistic graphical model. A mission objective is a quantifiable goal that is necessary for overall mission success, such as the avoidance of collisions. Probabilities for nodes in the graphical model may be computed from low-level assurance measures, and conditional probability tables will be computed from data. The high-level assurance measure concept has been developed in Phase 3, and is illustrated in this report using examples based on the challenge problems we addressed in Phase 3 (see Section 4.2 for more details on the application of assurance measures to the challenge problems of this program).

The low-level assurance measures provide uncertainty quantification for individual system components. Such system components include LECs as well as any other sensors that produce a real-time signal, such as sonar or cameras. For any assurance measure, there are two key elements to monitor, which are addressed by anomaly detectors and an inference module (Figure 1).

The first is the input to the system (input anomaly detector). If the LE-CPS is operating in an unknown environment, the system may not perform as expected. The second element is the performance of the system in response to a given input (inference and inference anomaly detector). Even if the LE-CPS is in an environment for which it was designed, it may underperform in certain scenarios. For each assurance measure, we seek to quantify the uncertainty for each of these components by providing out-of-distribution detection as well as epistemic uncertainty measurements for system performance.

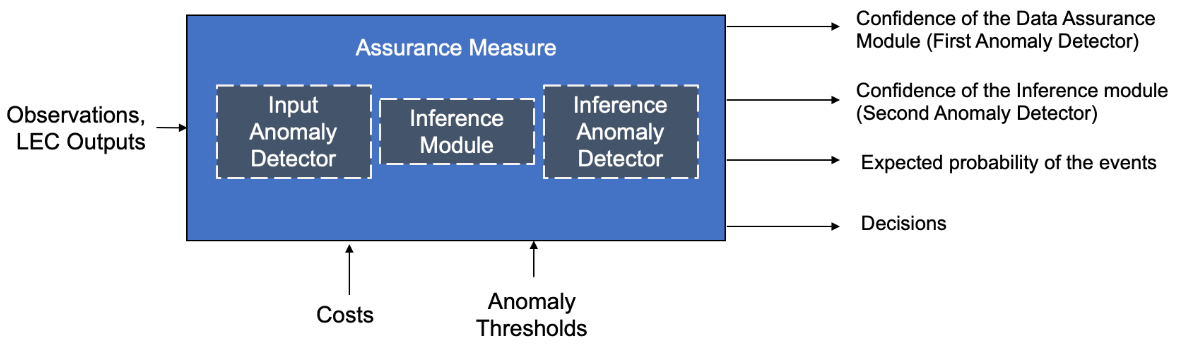

Assurance Measure
Input Anomaly Detector
Inference Module
Inference Anomaly Detector
Observations, LEC Outputs
Confidence of the Data Assurance Module (First Anomaly Detector)
Confidence of the Inference module (Second Anomaly Detector)
Expected probability of the events
Decisions
Costs
Anomaly Thresholds

Figure 1. Low-level Assurance Measure

3.1.2 System Architecture

Figure 2 shows a proposed architecture for trustworthy autonomy: a collection of run-time monitors that assess system properties (which may include assurance properties), taking inputs from the environment and system state [ADP2020]. The assurance measure quantifies the confidence/uncertainty in the assurance properties using both the inputs and outputs of the monitors. This, in turn, is one of the inputs for a decision mechanism that determines whether to proceed with the nominal system operation or to invoke contingency management actions. For example, when there is insufficient confidence, any expected system output that is otherwise assured may be masked.

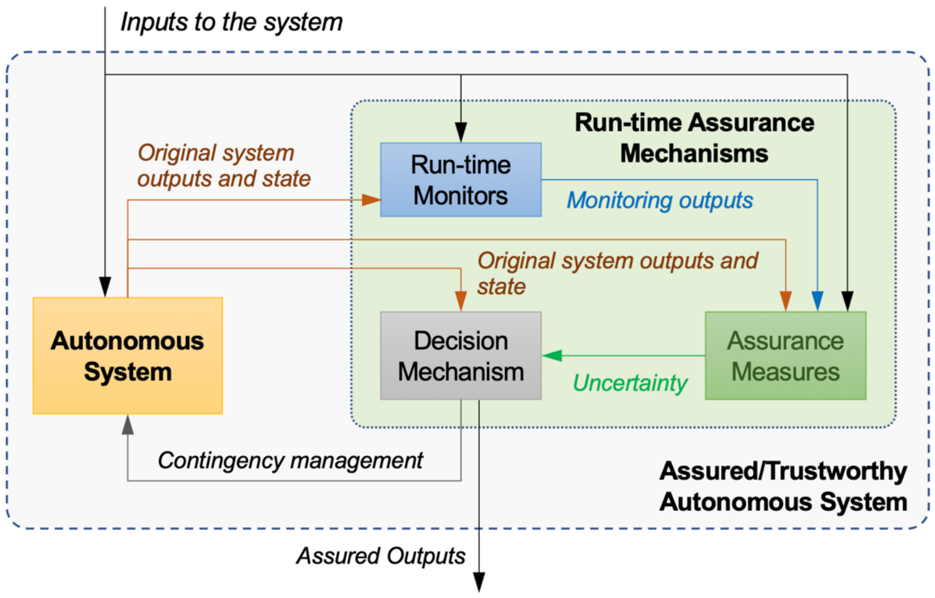


Figure 2. A Proposed Architecture for Trusted Autonomy

The aim of run-time assurance, also known as run-time verification, is to provide updates as to whether a system satisfies specified properties as it executes [ASTM2021]. Typically, this uses run-time monitors, which evaluate the properties using values extracted from the system and environment state. In a sense, therefore, the notion of assurance measure we have described here is a kind of monitor. However, it is worth making the following distinctions:

- monitors typically relate directly to properties of the system, whereas an assurance measure characterizes confidence in our knowledge of such properties; and
- an assurance measure seeks to aggregate a range of sources of information, including monitors. Thus it can be seen as a form of data fusion.

The architecture in Figure 2 is also closely related to the simplex architecture and its variants [ASTM2021], though there are some differences: since the assurance measure models the AS, which may itself be implemented as a simplex architecture, assurance measure outputs can be viewed as providing an additional level of analytic redundancy that is wider in scope than the safety controllers that simplex traditionally employs. We believe this can be advantageous in a run-time tradeoff between performance and safety. Also, the simplex decision making takes environment state as one of the inputs, while here they are reflected in the uncertainty forecast from assurance measures.

3.2 Methodology

We now clarify the relationship of assurance measures to the DAC concept and its core components (described in detail in the next section). Figure 3 shows a high-level methodology of the lifecycle of developing a DAC for an AS (broadly considered as the physical and logical system descriptions and its concept of operations). First we establish a baseline level for sufficient assurance, largely, by developing a static AC focused on the system requiring assurance (see Figure 3). This comprises various artifacts as shown (in the box labeled "system focus"), e.g., hazards, mitigation requirements, risk scenarios, and risk reduction justification in the form of structured arguments linked to evidence items such as simulation results, formal verification, etc. At this stage, pre-deployment assurance quantification is also a key component.

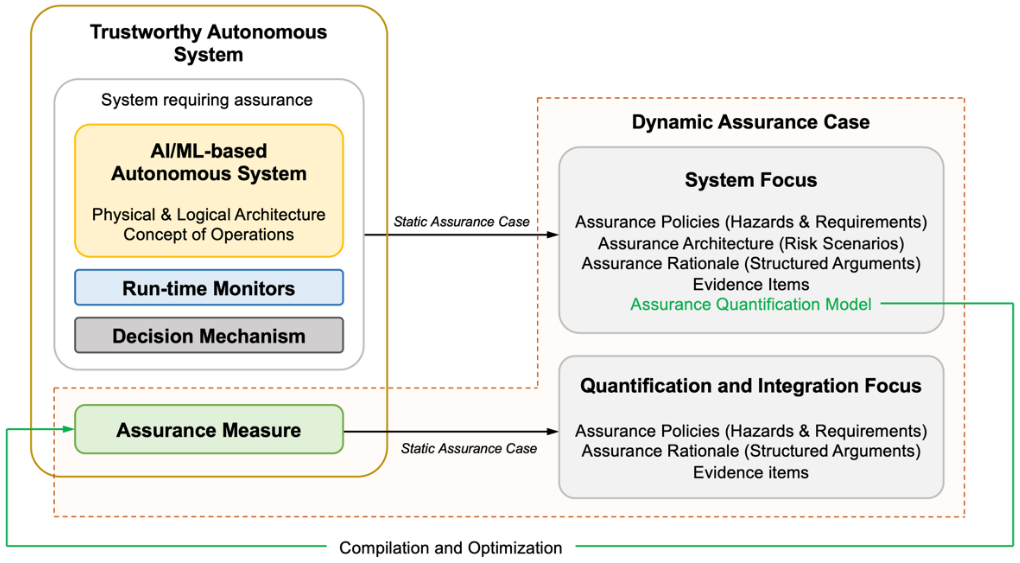

Trustworthy Autonomous System
System requiring assurance
AI/ML-based Autonomous System
Physical & Logical Architecture
Concept of Operations
Run-time Monitors
Decision Mechanism
Assurance Measure
Static Assurance Case
Static Assurance Case
Dynamic Assurance Case
System Focus
Assurance Policies (Hazards & Requirements)
Assurance Architecture (Risk Scenarios)
Assurance Rationale (Structured Arguments)
Evidence Items
Assurance Quantification Model
Quantification and Integration Focus
Assurance Policies (Hazards & Requirements)
Assurance Rationale (Structured Arguments)
Evidence items
Compilation and Optimization

Figure 3. Dynamic Assurance Methodology

Besides quantifying the assurance baseline, together with the hazard analysis it enables us to identify and discriminate between properties for which we construct assurance measures, and those that require monitoring. Trivially, these include properties whose violation is expected to impede continued safe operation or prevent mission completion. However, for instance, monitoring may suffice for certain component-level properties verified in design under assumptions of system and environment states, while assurance measures may be better suited to system- level properties potentially affected by emergent behavior. In general, we assume that there exist run-time monitors, some of which are part of the system requiring assurance, while others are tied to the validity of the evidence items used.

We compile the corresponding quantification models into optimized executables—assurance measures—that we then integrate into the system architecture as described earlier (Figure 2). Via a dashboard, the assurance measure can also passively provide a real-time assessment of the confidence in assurance properties to end users. In either case there is a need to provide additional justified confidence in the efficacy of the assurance measure itself, and in the integration, i.e., that hazardous interactions of the assurance measure, decision mechanism, and control system have been managed.

For this we develop additional static assurance artifacts (see Figure 3, in the box labeled "quantification and integration focus") for objectives such as timeliness of the assurance measure in the context of recovery actions; assurance measure performance in terms of the sensitivity and specificity of its forecasts; and mitigating of hazardous interactions from integrating the assurance measure, e.g., propagation of uncertainty computation errors.

3.3 Dynamic Assurance Cases

Our concept of DAC is the combination of the system, quantification and integration focused static assurance artifacts, and the assurance measure, which provides dynamic assurance. Practically, through-life assurance has a broad scope, and a comprehensive DAC must address a plurality of core and supplementary assurance concerns [CDP2017] through one or more of its main, interrelated components (Figure 4). We consider these components to be part of an assurance toolkit, where the particular assurance concern being addressed informs which components are required, and impacts their size and complexity.

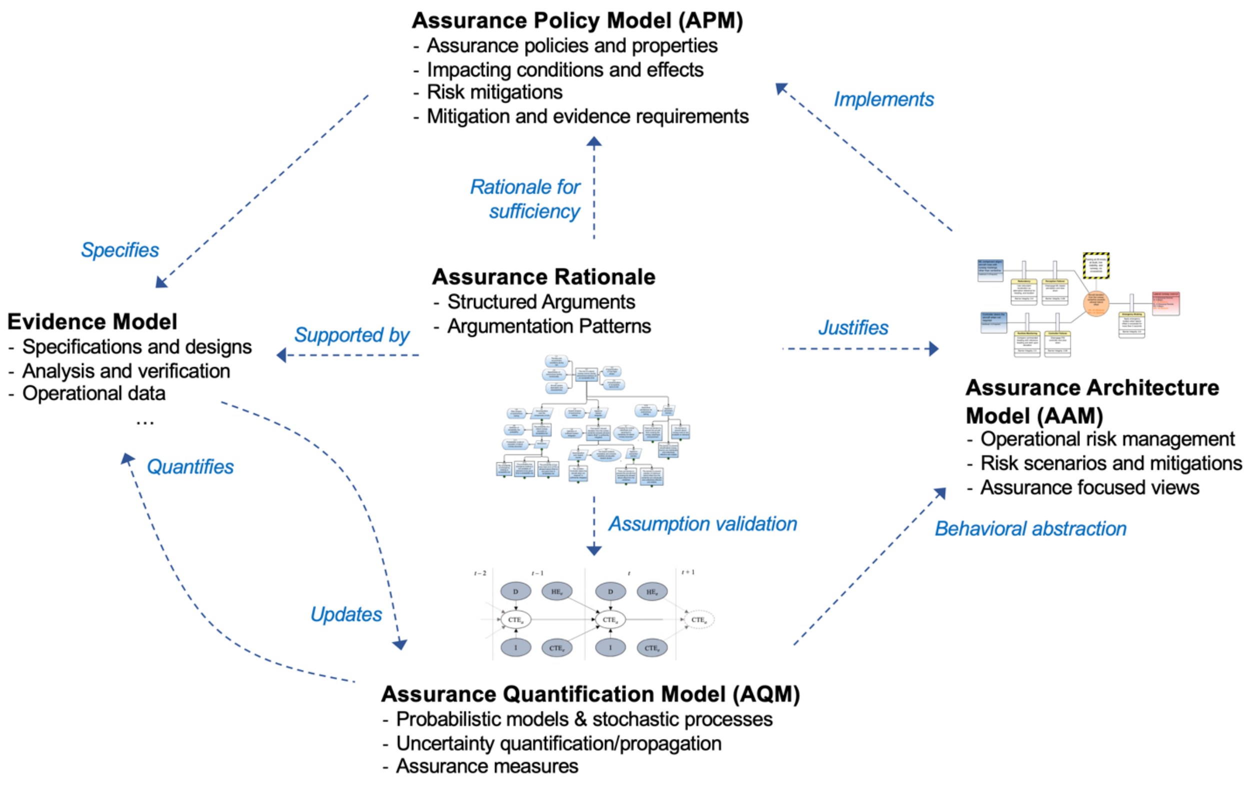
For instance, LE-CPS safety is a core concern—itself covering a broad gamut of assurance objectives including but not limited to design safety, and operational safety—that requires all DAC components. In contrast, reliable compilation of an ML model into a platform-specific executable has a narrower, tool-qualification focus. It represents a supplementary assurance concern that requires fewer DAC components We now describe each DAC component, their role in (dynamic) assurance, and their interrelations.

3.3.1 Assurance Policy Model

An assurance policy is a specification of the conditions that impact assurance properties, the resulting effects, and the corresponding mitigations for (the risk associated with) those effects. The assurance policy model (APM) is a model-based representation of assurance policies providing a basis against which sufficiency of assurance can be established. As such, it is both related to and kept consistent with other core DAC components. as we will clarify when describing the latter.

The APM concretely expresses what LE-CPS assurance means in terms of: i) the conditions under which assurance is impacted, in particular where there is a higher risk of undesired effects, and ii) the requirements for mitigating the risk associated with those impacts. As previously mentioned, these requirements are, in fact assurance claims that are yet to be substantiated by evidence and therefore record assurance properties and their allocation to the relevant items. More generally, the APM captures the (functional and non-functional) guarantees to be provided together with the assumptions made, mappings to the LE-CPS physical and logical components, bounds on acceptable behaviors, system states, etc. This, in turn, facilitates reasoning about assurance gaps due to the various assumptions made, and due to scoping.

To formulate assurance policies in the APM, we can leverage traditional hazard analysis techniques, such as a Functional Hazard Analysis (FHA), or newer ones such as System Theoretic Process Analysis (STPA), along with requirements decomposition and refinement techniques.



Assurance Policy Model (APM)
- Assurance policies and properties
- Impacting conditions and effects
- Risk mitigations
- Mitigation and evidence requirements
Implements
Rationale for sufficiency
Specifies
Assurance Rationale
- Structured Arguments
- Argumentation Patterns
Evidence Model
- Specifications and designs
- Analysis and verification
- Operational data
...
Supported by
Justifies
Assurance Architecture Model (AAM)
- Operational risk management
- Risk scenarios and mitigations
- Assurance focused views
Quantifies
Assumption validation
Behavioral abstraction
Updates
Assurance Quantification Model (AQM)
- Probabilistic models & stochastic processes
- Uncertainty quantification/propagation
- Assurance measures

Figure 4. Dynamic Assurance Case Concept

3.3.2 Assurance Architecture Model

An assurance architecture models a system from an assurance viewpoint as a collection of scenarios that show how risk is modified. The assurance architecture model (AAM) is a model-based representation of the assurance architecture. We have adopted barrier models, extending an earlier notion of safety architecture [DPW2019], to represent the AAM. This choice has been motivated by the observation that a collection of scenarios can conveniently describe an assurance concern and, in turn, the related assurance properties. This tightly couples the AAM to the physical and functional items constituting the architecture, whilst highlighting the roles that the items and their capabilities play in risk modification, e.g., prevention, recovery, tolerance, masking, etc.

Conceptually, the AAM composes distinct but related operational scenarios, each of which models the impact on assurance in terms of: i) the progression of events that migrate the system to higher risk states; and ii) the mechanisms of the system architecture employed to manage risk. We use Bow Tie Diagrams (BTDs) to specify these scenarios in a graphical way [DPW2019].

Shared BTD elements capture relations between scenarios at appropriate abstraction levels. Thus, for system-level assurance concerns, we model system-level operational risk situations, each of which we can further refine into lower-level risk scenarios that themselves require design-time or operational mitigations. The AAM can be seen as an implementation of the APM, although each model is closely related to, and synchronized with the other, recording different information. For example, we can model the assurance architecture of the risk reduction mechanisms themselves as additional BTDs reflecting, for example, scenarios showing failure modes and their local effects. Simultaneously, the APM captures the corresponding assurance requirements though those are not reflected in the AAM.

3.3.3 Assurance Quantification Model

The main purpose of assurance quantification is to provide an assessment of the confidence that can be justifiably placed in an LE-CPS item based on data associated with measurable assurance properties. Consequently, with a suitable assurance quantification model (AQM), we can establish both a baseline level of assurance (to support the decision to release a system into service), and evaluate whether that level of assurance continues to be maintained at run-time (supporting a run-time risk assessment and mitigation).

Many LE-CPSs used in safety-critical applications are stochastic dynamical systems. As such, at the system-level, the AQM is a probabilistic, model-based representation of a stochastic process whose underlying random variables (RVs) describe the AS state space at a suitable level of abstraction. Specific realizations of these RVs correspond to the assurance properties of interest (in particular, those that can be reasonably quantified), while the associated probability distributions reflect the corresponding uncertainty in those properties. That is, we express confidence in assurance properties in terms of the uncertainty in the related assurance measures, with lower uncertainty corresponding to higher confidence that the related assurance property holds. Effectively, this is a probabilistic query on the AQM, leveraging a range of techniques for uncertainty quantification (UQ) and propagation that account for various types of uncertainty, such as model and parameter uncertainty. Component-focused assurance quantification of ML components is also feasible [ADP2019] and the corresponding AQM takes into account component-level usage details.

3.3.4 Evidence Model

Evidence underpins assurance and is crucial for trusting LE-CPSs. The evidence model as a core DAC component relates heterogenous evidence items, records their provenance, captures the assertions that can be made, along with their usage context, whilst facilitating their tracing to other core DAC components. In particular, we link evidence items to assurance rationale (described next) for both justifying specific AS assurance claims, and to justify why the evidence items should themselves be trusted. In the latter case, note that the assurance claims are about the evidence items, whereas in the former case, they are about the LE-CPS.

Abstractly, we can think of verification tools as being mappings from collections of evidence items (i.e., tool inputs such as a model, specification, or property) to other evidence items (i.e., tool outputs such as a proof, results of static analysis, or model checking), together with i) the relations that specify those mappings, and ii) any assumptions that must hold on the inputs, including dependencies between the inputs. In this case, the role of the evidence model is to record these various mappings, relations, and assumptions, and how they are invoked and referenced in a DAC for a specific LE-CPS. More generally, the evidence model is an interface to reference concrete external evidence items in the application-specific DAC components. For instance, when a structured argument references, say, a piece of formal verification, it refers to the corresponding entry in the evidence model.

3.3.5 Assurance Rationale

We use structured arguments to capture various kinds of assurance rationale, in much the same way as a traditional, static AC. They primarily serve to capture the reasoning why specific assurance claims should be accepted based on the evidence supplied. We also use structured arguments for the assurance of the remaining DAC components themselves, addressing such (meta) assurance as: sufficiency of the stated assurance policies; appropriateness of the assumptions made in the AAM and the AQM, e.g., independence of risk mitigations in an assurance architecture; the relevance and completeness of the scenarios specified in the AAM; and, as mentioned earlier, the suitability and relevance of the evidence used.

We can communicate this kind of rationale in a number of ways, e.g., as a narrative, in a tabular or a graphical form, or as a combination of the three. Here, we use the Goal Structuring Notation (GSN) [GSN2021]: a standardized graphical language to describe key components of an assurance argument. For methodological details on developing assurance arguments, see [DP2018].

4.0 RESULTS AND DISCUSSION

Here we summarize the platform-specific assurance cases created for the air domain and undersea domain challenge problems. We also describe the results of formal methods integration and the enhancements made to our DAC tool, AdvoCATE.

4.1 Platform-specific Dynamic Assurance Cases

4.1.1 Air Domain Challenge Problem – Autonomous Visual Landing

4.1.1.1 Challenge Problem Summary

This challenge problem (CP 3.1) is focused on assurance of LECs used for perception and decision making within the context of an air domain application, in particular autonomous visual landing (AVL). Landing procedures require that the aircraft enter a so-called *traffic pattern* (Figure 5) which comprises specific phases at which specific decisions must be made (by the pilot in command) to land safely. For this challenge problem, the LECs are invoked when the aircraft is on the *final* phase.

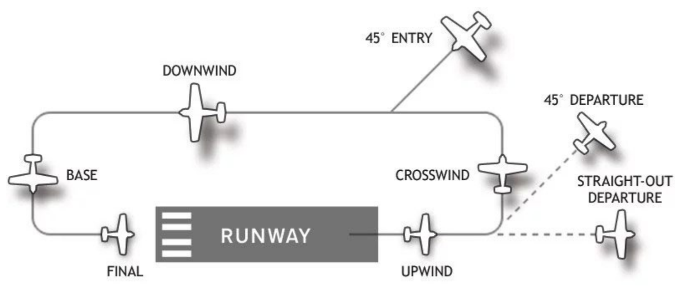


Figure 5. Typical Aircraft Landing Traffic Pattern

In this phase, the aircraft must be aligned with the runway (i.e., parallel to the horizon and heading in the same direction as the runway centerline) and descend at a steady rate whilst staying aligned. Typically, the descent follows a trajectory (so-called *glide path*) that has a fixed angle (between 3 – 5 degrees) to the horizontal plane (of the runway). The decision either to continue to land or to *go-around*, i.e., abort the landing, can be made at any point in the aircraft descent trajectory, depending on the (perceived) state of the runway, i.e., whether or not there are any collision hazards, and the stability of the aircraft as it descends. In instrument-based approach procedures using the prevailing Autoland and Instrument Landing Systems (ILS), such as those found in transport-category aircraft, a so-called *decision height* (DH) is the distance above the runway at which the pilot in command (first) decides whether or not to continue landing or to execute a missed approach (or a go-around).

The Concept of Operations (CONOPS) makes the following assumptions:

- Single runway operations under visual flight rules (VFR) and visual meteorological conditions (VMC)
- No crosswind operations
- No terrain obstructions on the glidepath (e.g., treetops, built-up structures, etc.)

- Non-towered airport environment (i.e., uncontrolled airspace, class G, without air traffic control services) with non-cooperative air traffic (i.e., aircraft operating without transponders)

The landing function is aided by two LECs:

- Runway Clear (RwyClr), the function responsible for perception, detection and tracking of potential collision hazards, and estimation of the future position of the detected hazards relative to the aircraft after it lands;
- Pose Estimation (PosEstm), the function responsible for determining the pose of the aircraft relative to the runway, i.e., its 3-D orientation in space, and translational position relative to the touchdown location on the runway. Effectively this LEC localizes the aircraft.

Functional Architecture

Figure 6 shows the high-level functional architecture for this challenge problem (italicized text in the figure indicates the data inputs and outputs).

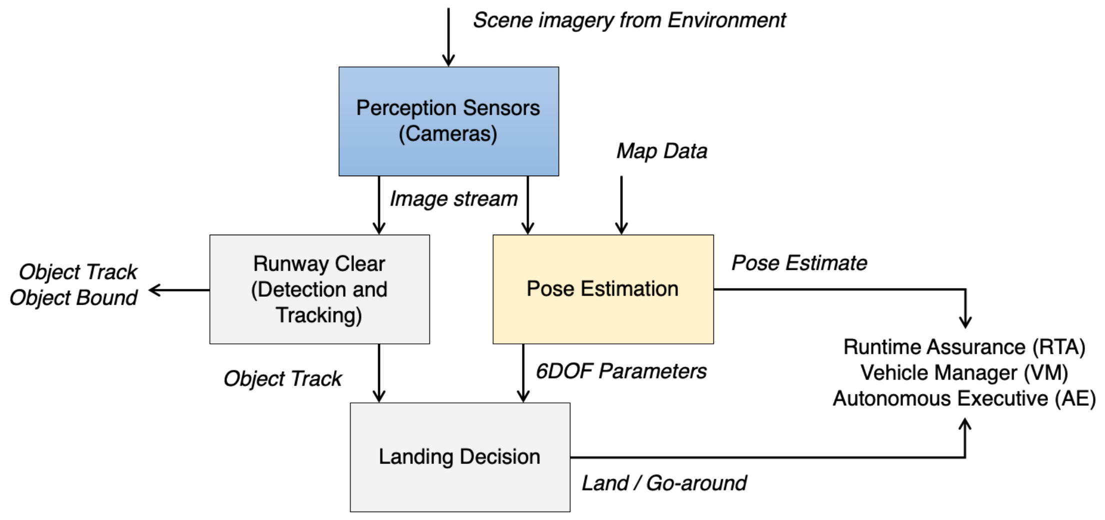


Figure 6. CP3.1 High-level Functional Architecture

Both LECs receive as input, a stream of full high definition (1920×1080 pixel) color images at a 10Hz rate (from wing-mounted cameras in the flying platform, and alternatively from a simulated 50mm and 12mm optic, in the hardware-in-the-loop, iron-bird, test platform). Both image streams are synchronized. PosEstm additionally receives external map data as input. In response to these inputs: (1) RwyClr produces as its output, the location of the centroid of a detected object in x, y coordinates relative to the landing location, together with an estimate of the velocity vector, and the diameter of the detected object; and (2) PosEstm produces as its output, the six degree of freedom (6-DOF) aircraft pose estimate aircraft, along with the pixel locations of the *keypoints* of the runways[1], i.e., specific points of interest in an image or scene.

Both sets of LEC outputs are used by a *Landing Decision* (LNDDESC) function whose response is a Boolean decision: (continue to) land, or go-around. The pose estimate and the response of landing decision are used by other aircraft functions, including those involved in controlling the aircraft during its descent.

Physical Architecture

We give a list of the subsystems (and constituent items) to contextualize the allocation of the functions and requirements described subsequently in Section 4.1.1.4.

- Perception Subsystem
  - Perception sensors
- Ownship State Estimation Subsystem
  - Global Positioning System (GPS) and Inertial Reference System (IRS)
- Vehicle Manager Subsystem
- Autonomous Executive Subsystem
- Contingency Manager Subsystem
- Navigation and Aircraft Database
- Runway Clear Subsystem
- Pose Estimation Subsystem
- Land Decision Subsystem
- Actuation Subsystem

4.1.1.2 Assurance Objectives

For this report, the focus is mainly on the assurance of the POSESTM LEC, in particular, its contribution to overall landing safety.

As such, the main safety assurance objectives are to provide sufficient confidence that under all specified[2] operating conditions: (i) neither the intended behavior of the POSESTM function nor its failure conditions lead to an unacceptable outcome, and (ii) the POSESTM function does not exhibit any unintended behavior that could lead to an unacceptable outcome, more frequently than the rate corresponding to an acceptable risk level—or equivalently, a target level of safety, (TLOS)—for those outcomes.

Here, the unacceptable outcomes to be avoided are:

- Collision with the terrain (e.g., a tail strike)

[1] For this challenge problem, 16 runway keypoints are relevant: 4 end-points of the runway, 4 end-points of the *runway threshold*, and 4 end-points of each of the left and right *aiming pads* on either side of the runway centerline.

[2] We assume the CONOPS is validated to have specified all foreseeable operating conditions.

- Controlled Flight Into Terrain (CFIT)
- Landing in an area other than the intended runway
- Runway excursion, i.e., rolling off the runway tarmac.

These outcomes are causally preceded by *landing safety hazards*, i.e., a combination of uncontrolled system states (SSs) and specific environmental conditions (ECs) that emerge during landing. 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. An assumption here is that RwyClr has established that there is no collision hazard in the landing trajectory. Thus, from a system safety standpoint, an additional unacceptable outcome to be avoided during landing (to which RwyClr contributes) is:

- Collision with a ground vehicle on the runway or taxiway, or with another aircraft.

Referring to the functional architecture in Figure 6, the LndDesc function uses the pose estimate and the track of the detected object to decide whether or not to land. Effectively, this involves predicting whether the track of the detected object is such that a *runway incursion*[3] is likely, or whether the approach is unstable at and after DH. Either of these conditions should result in a go-around decision.

4.1.1.3 Assurance Case Architecture

The assurance case architecture gives a high-level overview of (the structure of) the rationale used to substantiate that the specified assurance objectives have been met. Figure 7 shows a graphical schematic of the AVL assurance case architecture (Note: this concept has not been formalized in this project; however it is introduced here since it useful to present the "big-picture" of the DAC).

The graph on the right of Figure 7 (whose root node is labeled "*AVL System Description, functional and physical organization*") represents the structure of the overall system-level assurance case. Each node itself abstracts a fragment of the underlying assurance argument. The enclosing rounded rectangles (e.g., the box labeled "*AVL safety claims*") indicates the assurance aspect being addressed by the corresponding argument. For instance, the root node and its immediate child nodes abstract a fragment of the assurance case that concerns safety claims about the AVL system, given in terms of the system hazards and functional failures. Likewise, lower levels of architecture—e.g., those enclosed by the box labeled "*Safety Requirements Decomposition and Allocation*", concern arguments that invoke a requirements decomposition and allocation inference strategy for the system elements listed (*Navigation, Sensing, Nav. State Estimation,* etc.).

The interpretation of this graph is as follows: assurance of AVL system safety involves system-level safety claims, which are then decomposed into claims about the various system functions, shown here as claims about the *Navigation* function and *Other* functions. Since the focus of this DAC is on PosEstm, a function that in essence supports aircraft navigation, only this branch of the assurance case architecture has been shown, while other parts of the assurance case architecture,

[3] An occurrence at an aerodrome involving the incorrect presence of an aircraft, vehicle, or person on the protected area of a surface designated for the landing and takeoff of aircraft.

e.g., for Sensing, are not in scope. In particular, the *Navigation* function includes lower-level functions for *Sensing*, *Navigation* (*Nav.*) *State Estimation*, and for *Contingency Management*. This fragment of the assurance case architecture thus indicates that assurance of higher-level system safety claims relies, in part, upon assurance of those lower-level functions.

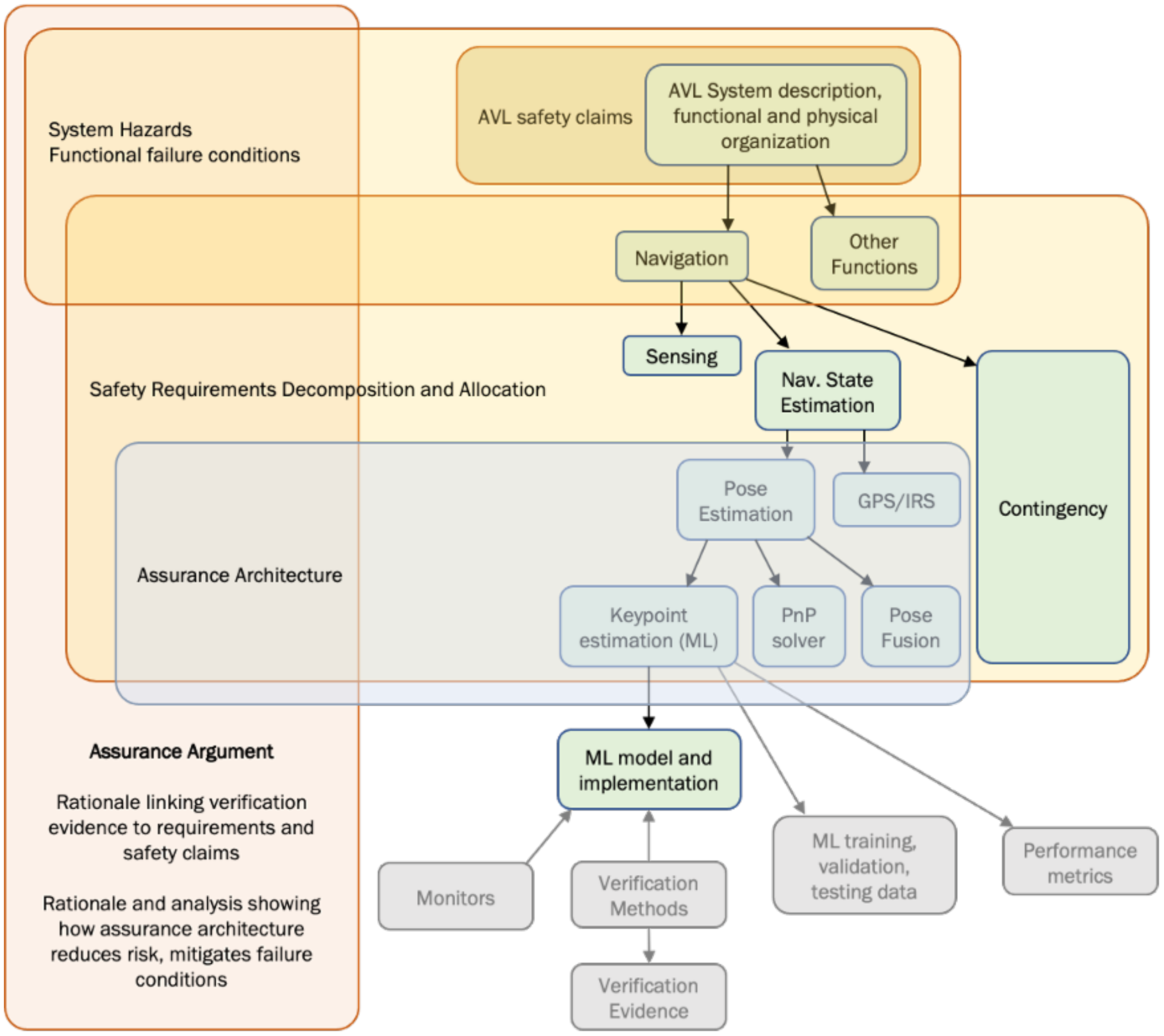

System Hazards
Functional failure conditions
AVL safety claims
AVL System description, functional and physical organization
Navigation
Other Functions
Sensing
Nav. State Estimation
Contingency
Safety Requirements Decomposition and Allocation
Pose Estimation
GPS/IRS
Assurance Architecture
Keypoint estimation (ML)
PnP solver
Pose Fusion
ML model and implementation
Assurance Argument
Rationale linking verification evidence to requirements and safety claims
Rationale and analysis showing how assurance architecture reduces risk, mitigates failure conditions
Monitors
Verification Methods
Verification Evidence
ML training, validation, testing data
Performance metrics

Figure 7. Schematic of AVL Assurance Case Architecture

Nav. State Estimation, in turn, leverages the pose estimates produced by PosEstm, which itself relies upon the LEC-based keypoint estimates—shown here as *Keypoint Estimation* (*ML*)—a so-called *Perspective n-Point* (PnP) solver, and *Pose Fusion*. The PnP solver implements an algorithm that estimates the 6-DOF pose based on three-dimensional points in space and their corresponding two-dimensional image projections (i.e., the keypoints).

Assurance of *Keypoint Estimation* involves providing confidence that the implemented ML model used for estimating keypoints fulfils the allocated assurance objectives. This assurance may also rely upon evidence of appropriate ML model training, validation and testing, and can be further supported through performance metrics that characterize its behavior on known as well as unseen data.

4.1.1.4 Assurance Case Elements

Functional Decomposition and Allocation

The AVL function is decomposed into the following sub-functions that are allocated to the physical system architecture as below:

- *Guidance* and *Control* sub-functions, each allocated to the Vehicle Manager, Autonomous Executive, and Land Decision subsystems;
- *Navigation* function allocated to the Pose Estimation, Ownship State Estimation, and Perception subsystems;
- *Ground traffic awareness* function, allocated to the Runway Clear and Perception subsystems.

Hazards and Requirements

As previously mentioned, the main landing hazard to which the `PosEstm` function can contribute is: *Unstable Approach*[4].

The (safety) requirement corresponding to avoiding this hazard is stated as follows: *The aircraft shall have a stable final approach lined up with the designated runway descending on a constant angle glidepath* (*between 3 and 5 degrees glideslope*) *towards the aiming point.*

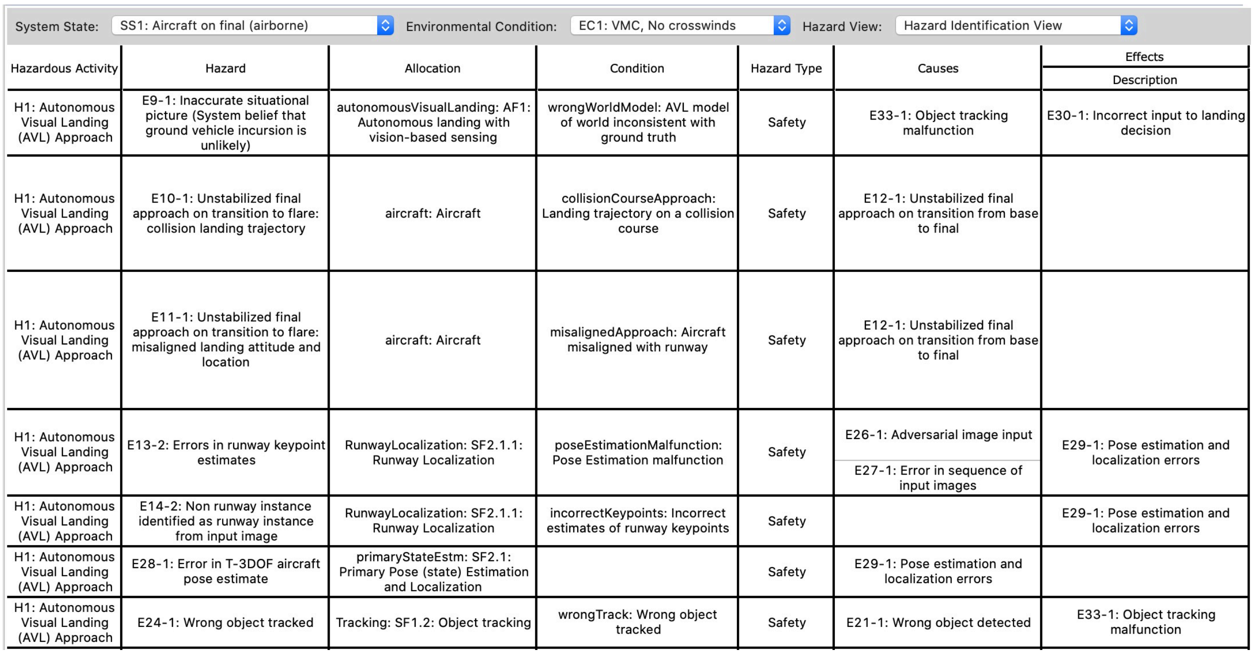
This requirement can be refined into lower-level requirements that specify what constitutes a stable final approach, in terms of the aircraft system state parameters, e.g., airspeed, attitude, position of the landing gear and control surfaces, power/thrust settings, descent or sink rate, altitude/height above touchdown, etc. The safety requirement that *mitigates* (more specifically, recovers from) the unstable approach landing hazard is stated as follows: *The aircraft shall reject the landing and execute a go-around if the approach is unstable at the decision height* (DH) *appropriate for the aircraft type and landing procedure.*

A precursor event to the unstable approach landing hazard is *navigation state error.* The pose estimate produced by the `PosEstm` function, which characterizes the 6-DOF orientation of the aircraft, is a component of the *navigation state* of the aircraft. As such, errors in pose estimation contribute to the navigation state error and thereby to the unstable approach landing hazard.

Figure 8 shows a screenshot of the AdvoCATE tool used to record the DAC for this air-domain CP. Specifically, what is shown is an excerpt of the hazard log produced from the formal hazard identification process, conducted as part of a functional hazard assessment (FHA), on the AVL function. As shown, the system state (SS) being considered for analysis is when the aircraft is on final approach (i.e., it is airborne), under the applicable environmental conditions (ECs) specified in the CONOPS (i.e., VMC with no crosswinds).

[4] Also called as an *Unstabilized Approach.* In this report, both terms are used interchangeably.

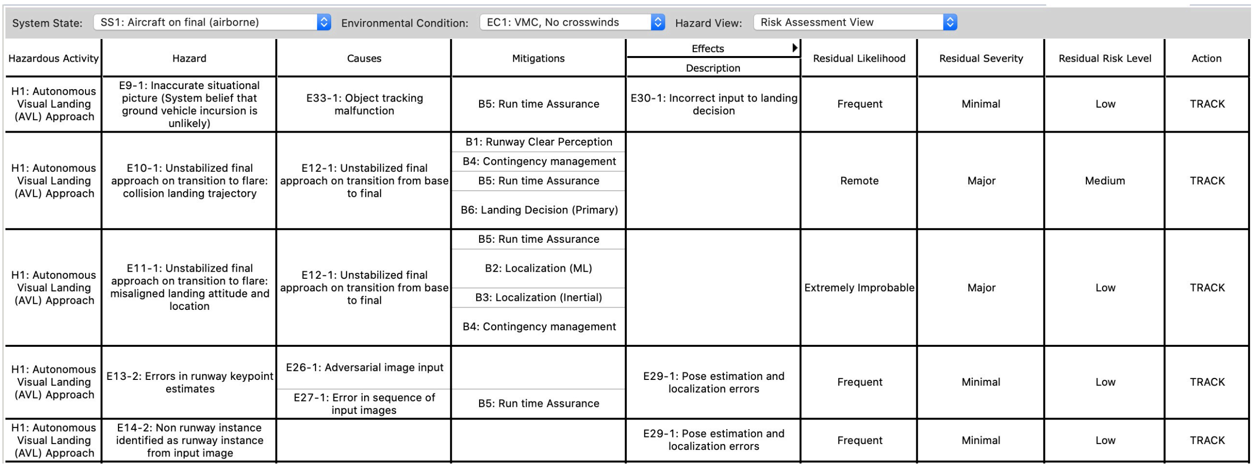
The *allocation* for the identified hazards (more specifically functional failure conditions) indicates the function (or the allocated subsystem) where the failure condition manifests. The *condition* specifies the particular loss of control situation characterizing the functional failure. The plausible causes and effects of the hazards also have been specified. For instance, the hazard *E13-2: Errors in runway keypoint estimates* corresponds to a functional failure of PosEstm, whose potential causes include *E26-1: Adversarial image inputs*, and *E27-1: Errors in the sequence of input images*, leads to the effect *E29-1: Pose estimation and localization errors.*



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

Figure 8. Hazard Identification (FHA) in AdvoCATE

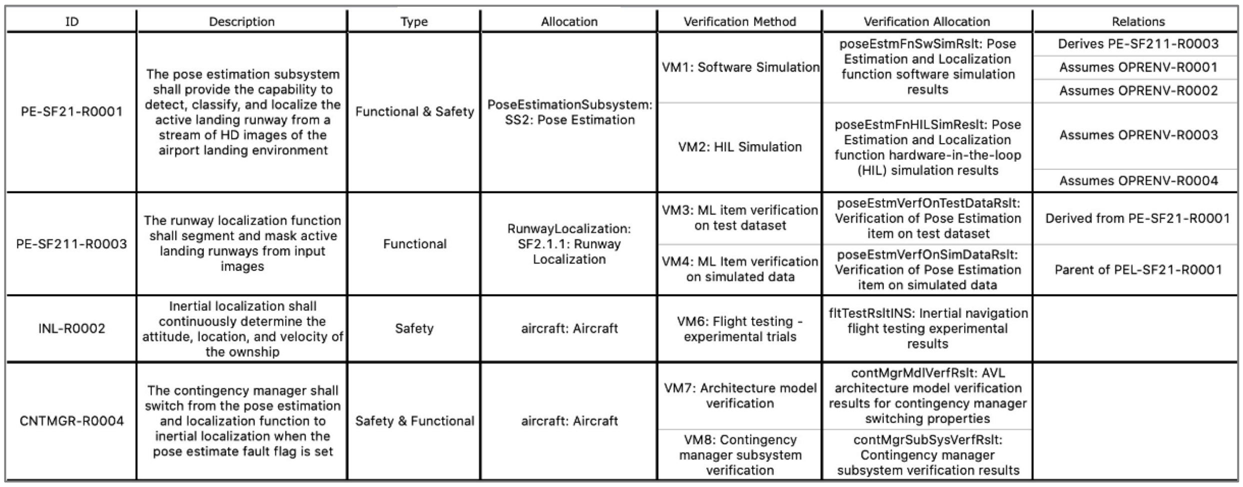


System State: SS1: Aircraft on final (airborne) Environmental Condition: EC1: VMC, No crosswinds Hazard View: Risk Assessment View

Hazardous Activity	Hazard	Causes	Mitigations	Effects	Residual Likelihood	Residual Severity	Residual Risk Level	Action
				Description				
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

Figure 9. Risk Assessment (FHA) in AdvoCATE

Figure 9 shows a subsequent step in the FHA, presenting a suite of candidate mitigations for each of the identified hazards, along with the expected level of risk reduction upon proper implementation and invocation of the mitigations. For example, mitigations for the aircraft-level hazard *E11-1: Unstabilized final approach on transition to flare: misaligned landing attitude and location* include *B5: Runtime assurance*, *B2: Primary localization based on machine learning*, *B3: secondary localization based on inertial sensors*, and *B4: contingency management* mechanisms.



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	

Figure 10. Excerpt of Requirements Log in AdvoCATE

Figure 10 shows an excerpt of the requirements log as recorded using AdvoCATE, containing the safety / mitigation requirements that emerge as a consequence of the FHA. In addition to the statement of the requirements, the requirements log records other relevant information, such as: the type of requirements, their allocation (to an element of the physical architecture, or a function in the functional architecture), proposed verification methods, the location of the verification results (verification allocation), as well as relations between requirements that help to support additional assurance activities such as ensuring internal consistency.

For example, the requirement *PE-SF211-R0003: the runway localization function shall segment and mask the active landing runways from input images* has been allocated to the subfunction *SF2.1.1: Runway Localization*, one of the lower-level functions of PosEstm along with *Keypoint Estimation* and *PnP Solver State Estimation*. Two verification methods have been identified: *VM3: ML item verification on test data set*, and *VM4: ML item verification on simulated data*, along with their respective verification allocations. This requirement is itself derived from the choice of using ML—in particular a Deep Neural Network (DNN), *U-Net* [RFB2015]—to detect and localize the runway using a segmentation-based approach.

In addition to the excerpt of requirements shown in Figure 10, the main requirement on PosEstm is stated as follows: *The pose estimate of aircraft attitude, location, and velocity shall be consistent with the true aircraft pose*. This is both a functional and a safety requirement, since an incorrect pose estimate can lead to an unstable approach due to the control system compensating when not required. This requirement includes aspects of *timing safety* (i.e., on the worst-case execution time and real-time deadlines that apply during pose estimation), and the *required navigation performance* (i.e., the accuracy and precision bounds on the pose estimates produced). Although these concerns are within the scope of the DAC, they require specific implementation choices that were not in the scope of this effort; hence we do not consider them further.

Subsystem Safety Analysis

Subsequent to the FHA, the safety analysis of the Pose Estimation subsystem (or equivalently the PosEstm function) additionally involves characterizing the impact of various kinds of inputs (including propagated failure conditions from upstream subsystems/items) in terms of the effects produced both within the boundary of the function/subsystem, and at its boundary to the wider, containing system.

For instance, inputs to the Pose Estimation Subsystem are images from the Perception subsystem, in particular an image stream from the perception sensors (cameras), and map data from the Navigation and Aircraft Database (see Figure 6 and the accompanying narrative). Sensor failure conditions can manifest as so-called out-of-distribution (OOD) inputs and/or adversarial inputs that can lead to failure conditions of the *Runway Localization* and *Keypoint Estimation* sub-functions. If those failure conditions are, in turn, not detected and corrected/mitigated they will propagate to the PnP solver, leading to navigation state estimation failure conditions.

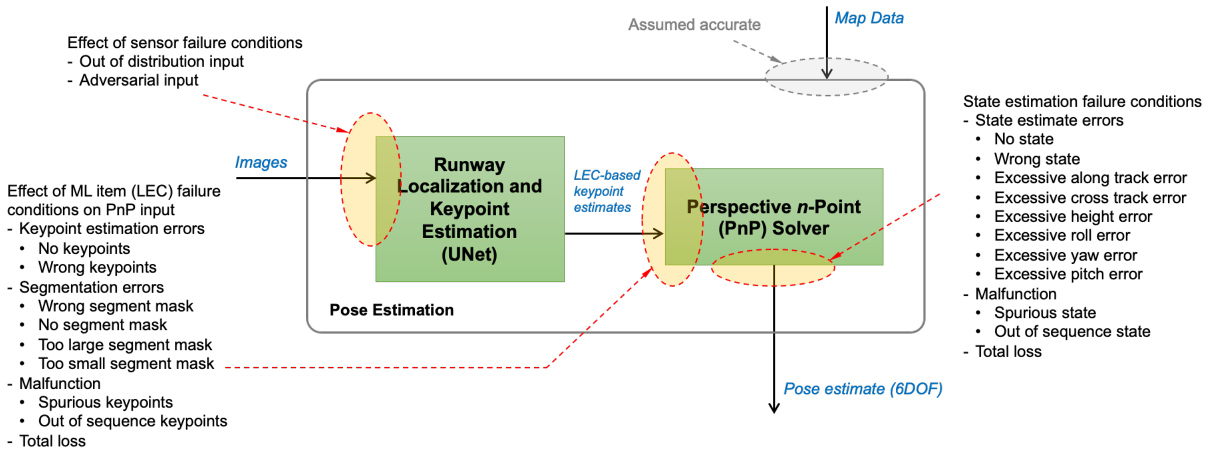


Figure 11. Pose Estimation Subsystem Safety Analysis

Figure 11 shows an internal schematic of the PosEstm function, showing how image inputs are used to produce *keypoint estimates* and runway segment masks (not indicated) using U-Net, following which the PnP solver produces 6-DOF pose estimates. A candidate list of failure conditions at the input to each function, and their effect is also given. For example, OOD or adversarial inputs can produce any one or more of the following erroneous responses from U-Net, such as no keypoints, wrong keypoints, spurious keypoints, keypoints produced out of sequence, no segment mask, wrong segment masks, or inaccurate segment masks. Those erroneous inputs, in turn, can lead to a variety of state estimate errors, or malfunctions in pose/state estimation.

Safety Architecture

The safety architecture represents a composition of mitigated risk scenarios.

We can model the system/functional hazards, their precursors, and the contributing failure conditions identified through the FHA and the subsystem safety analysis (see preceding discussion) and their inter-relations in terms of a risk scenario, i.e., a causal event chain showing how initiating events lead to loss of control events (hazards) that eventually manifest as the undesired effects that are to be avoided. Figure 12 gives an example of one such risk scenario modeled as a BTD (see Section 3.3.2), showing initiating events (e.g., *Adversarial image input*) leading to the hazard (also known as a *top event* in BTD terminology) *Errors in runway keypoint estimates*, that eventually leads to the effect *Pose estimation and localization errors*. In fact, this risk scenario (partially) represents the propagation of sensor errors via the U-Net ML model (that implements the *Runway Localization* and *Keypoint Estimation* sub-functions of `PosEstm`) across the function interface in the form of a function failure condition.

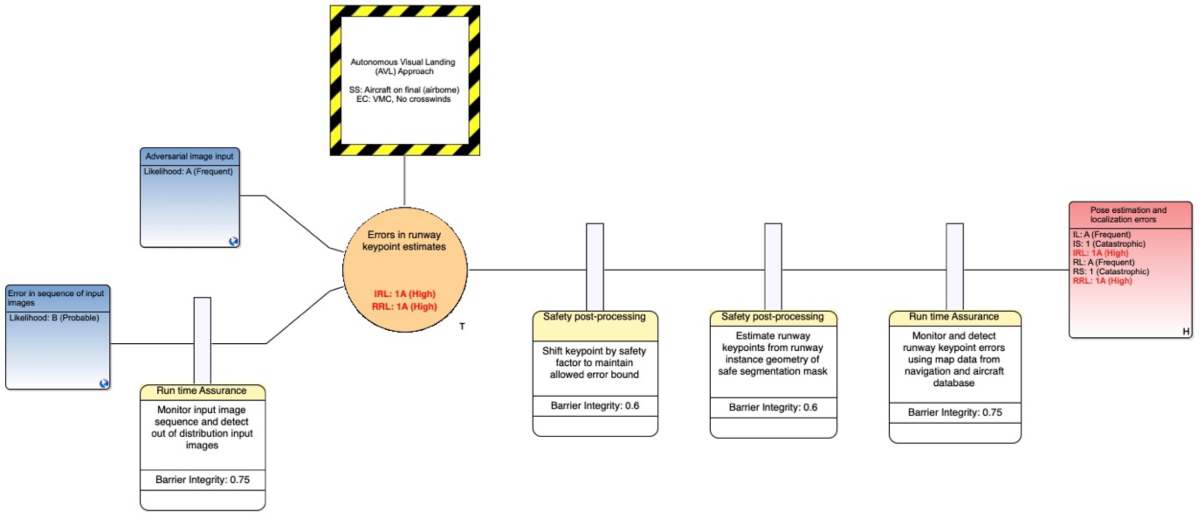

Autonomous Visual Landing (AVL) Approach
SS: Aircraft on final (airborne)
EC: VMC, No crosswinds
Adversarial image input
Likelihood: A (Frequent)
Error in sequence of input images
Likelihood: B (Probable)
Errors in runway keypoint estimates
IRL: 1A (High)
RRL: 1A (High)
T
Run time Assurance
Monitor input image sequence and detect out of distribution input images
Barrier Integrity: 0.75
Safety post-processing
Shift keypoint by safety factor to maintain allowed error bound
Barrier Integrity: 0.6
Safety post-processing
Estimate runway keypoints from runway instance geometry of safe segmentation mask
Barrier Integrity: 0.6
Run time Assurance
Monitor and detect runway keypoint errors using map data from navigation and aircraft database
Barrier Integrity: 0.75
Pose estimation and localization errors
IL: A (Frequent)
IS: 1 (Catastrophic)
IRL: 1A (High)
RL: A (Frequent)
RS: 1 (Catastrophic)
RRL: 1A (High)
H

Figure 12. Bow Tie Diagram for Errors in Keypoint Estimation

Also shown in BTD in Figure 12 are a suite of mitigations (*barriers* and *controls*, in BTD terminology), that are meant to reduce risk and either prevent, or recover from, respectively, 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 function invokes a monitoring capability to observe sequences of input images to detect errors, and OOD inputs—a *control*. Likewise, when errors in keypoint estimation inevitably occur, additional mitigations are to be invoked, including:

- *Safety post-processing*: This involves two controls, namely: (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*: This 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 (see Figure 11).

By constructing several such risk scenarios modeling the different causal chains of events and the associated mitigations, and composing them, a new model—the safety architecture—can be formed that specifies: (i) the mitigations used to manage hazards and their causes and effects; (ii) the circumstances (scenarios) under which the mitigations are invoked. For more details on the specifics of safety architecture development, refer to [DPW2019].

Figure 13 gives a fragment, highlighting some barriers, as shown by the shaded, dotted ovals. These represent the modifications to the original functional architecture (Figure 11) that have been introduced to mitigate the contribution of the failure conditions—identified in the PosEstm subsystem safety analysis—to the unstable approach landing hazard, thereby providing higher assurance that PosEstm will meet its requirement (of producing pose estimates that are consistent with the true aircraft pose). Figure 14 shows the modified functional architecture.

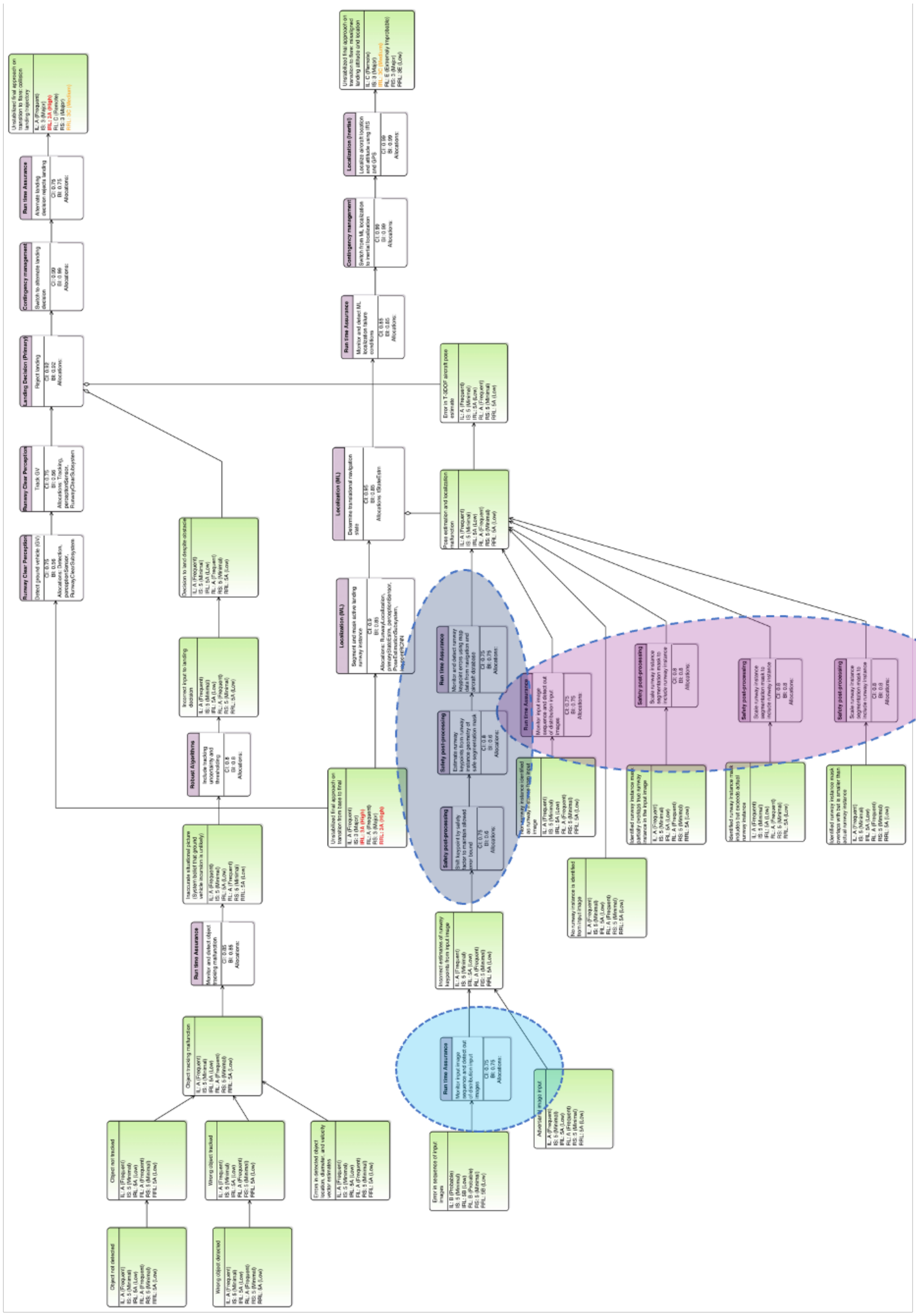


Figure 13. Excerpt of Safety Architecture for Pose Estimation

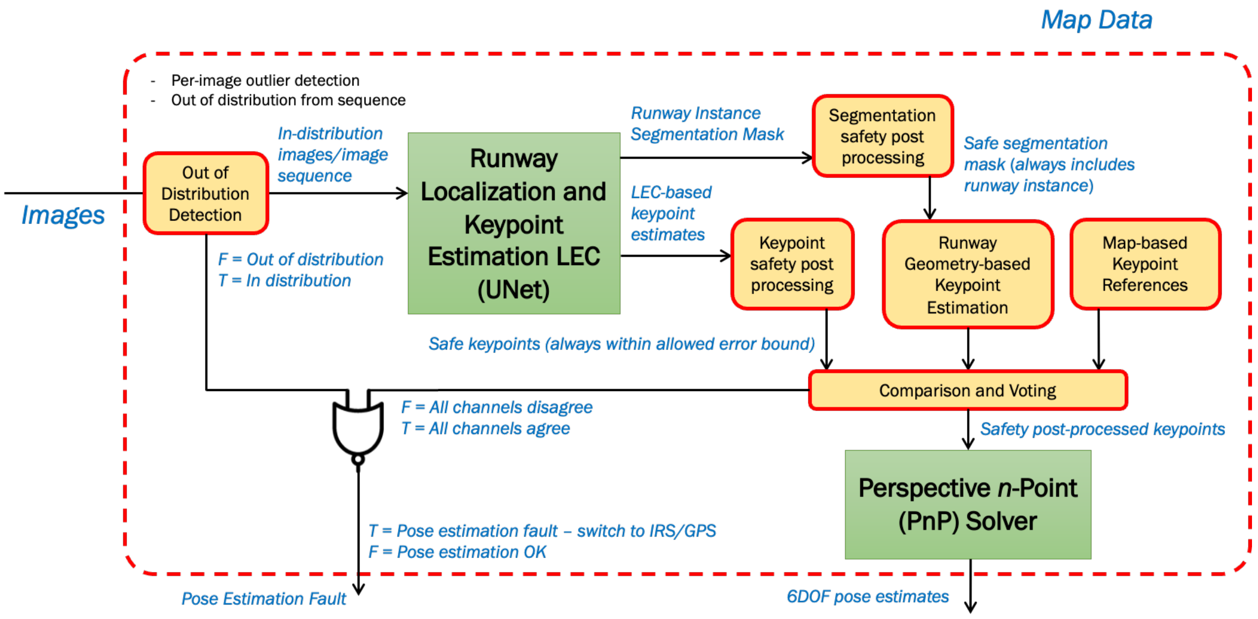

Map Data
- Per-image outlier detection
- Out of distribution from sequence
Images
Out of Distribution Detection
In-distribution images/image sequence
Runway Localization and Keypoint Estimation LEC (UNet)
Runway Instance Segmentation Mask
LEC-based keypoint estimates
Segmentation safety post processing
Safe segmentation mask (always includes runway instance)
Keypoint safety post processing
Runway Geometry-based Keypoint Estimation
Map-based Keypoint References
F = Out of distribution
T = In distribution
Safe keypoints (always within allowed error bound)
Comparison and Voting
F = All channels disagree
T = All channels agree
Safety post-processed keypoints
Perspective n-Point (PnP) Solver
T = Pose estimation fault – switch to IRS/GPS
F = Pose estimation OK
Pose Estimation Fault
6DOF pose estimates

Figure 14. Modified Pose Estimation Functional Architecture for High Assurance

The following correspondence between the mitigation functions of the modified functional architecture (Figure 14), and the risk scenario / BTD view of the safety architecture (Figure 12) can be observed:

- The block labeled *Out of Distribution Detection* (Figure 14) implements the risk mitigation control for errors in sequences of images input to PosEstm as specified in the *Run time Assurance* prevention barrier (i.e., the first barrier on the left in Figure 12).
- The *Safety post processing* barrier (Figure 12) contains two recovery controls, the first of which is implemented by the functions associated with the blocks labeled *Keypoint safety post processing* and *Segmentation safety post processing*, respectively (Figure 14). The second control is implemented by the block labeled *Runway Geometry-based Keypoint Estimation* (Figure 14).
- Lastly, the *Run time Assurance* recovery barrier/control (i.e., the fourth barrier in Figure 12) maps to the blocks labeled *Map-based Keypoint References*, and *Comparison and Voting* (Figure 14).
- Additionally, the responses of the *OOD Detection* function and the *Comparison and Voting* function can be used to produce a fault flag that indicate pose estimation faults. That, in turn, can be used to invoke a contingency mechanism implemented by the Contingency Manager subsystem.

Note that the modified functional architecture in Figure 14 serves to provide assurance of keypoint estimation fidelity. However, the approach to PosEstm assurance additionally requires assurance of the PnP solver, since pose estimates are in fact produced by applying the PnP algorithm to the keypoints it receives as input from the *Keypoint Estimation* function. Although we do not consider assurance of the PnP solver in this report, we indicate its role in the assurance rationale component of the DAC discussed subsequently.

Recall (Figure 5) that during aircraft landing, a traffic pattern is entered which involves multiple phases. We can abstract the safety architecture to reflect this phased view of risk scenarios. The idea is that in each phase, there are different risk scenarios, some of which can be common across phases, and others that are related across phases.

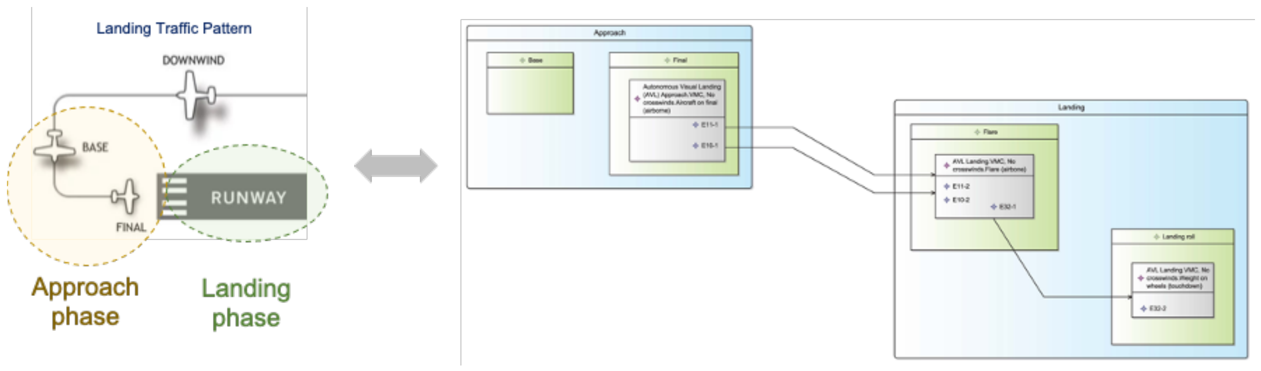

Landing Traffic Pattern
DOWNWIND
BASE
FINAL
RUNWAY
Approach phase
Landing phase
Approach
Base
Final
Autonomous Visual Landing (AVL) Approach.VMC, No crosswinds.Aircraft on final (airborne)
E11-1
E10-1
Landing
Flare
AVL Landing.VMC, No crosswinds.Flare (airborne)
E11-2
E10-2
E32-1
Landing roll
AVL Landing.VMC, No crosswinds.Weight on wheels (touchdown)
E32-2

Figure 15. Phases View of the Safety Architecture for Pose Estimation

Figure 15 shows a phases view of the safety architecture (also see Section 4.4.3.2) for PosEstm, showing the phases of descent: approach, and landing, each of which themselves comprise sub-phases: *Base*, *Final* (Approach), and *Flare*, and *Landing Roll* (Landing) respectively. The figure shows nodes that abstract the phase-specific safety architecture for each sub-phase, and the links indicate that events in one sub-phase impact events in the subsequent sub-phases.

Assurance Rationale

The assurance rationale for PosEstm embeds the following argument (described in a narrative form, interspersed with fragments of the graphical argument structures, as captured in AdvoCATE)

The main safety claim is formulated as follows: PosEstm *is acceptably safe for use.* Here, "*acceptably safe*" is defined in terms of the unacceptable outcomes to which PosEstm contributes (see Section 4.1.1.2) not occurring more frequently than the rate corresponding to the TLOS considered acceptable[5] for those outcomes. This safety claim can be decomposed based on the safety objectives stated earlier (Section 4.1.1.2): that is, under all specified operating conditions (i) neither the intended behavior of the PosEstm function nor its failure conditions lead to an unacceptable outcome, and (ii) the PosEstm function does not exhibit any unintended behavior that could lead to an unacceptable outcome.

These objectives can be reflected as the following claims: PosEstm *satisfies its allocated system (safety) requirements*; *all identified failure conditions* of PosEstm are sufficiently mitigated; all identified hazardous interactions of PosEstm are sufficiently mitigated. The first relates to intended

[5] As defined by a regulator, for example.

behavior not leading to an unacceptable outcome, i.e., satisfying its functional safety requirements. The third relates to PosEstm not exhibiting unintended behavior.

Figure 16 shows a high-level structure—an *argument architecture*—of the safety argument, reflecting the decomposition of the main safety into three sub-claims and associated supporting sub-arguments (to be elaborated subsequently in this section).

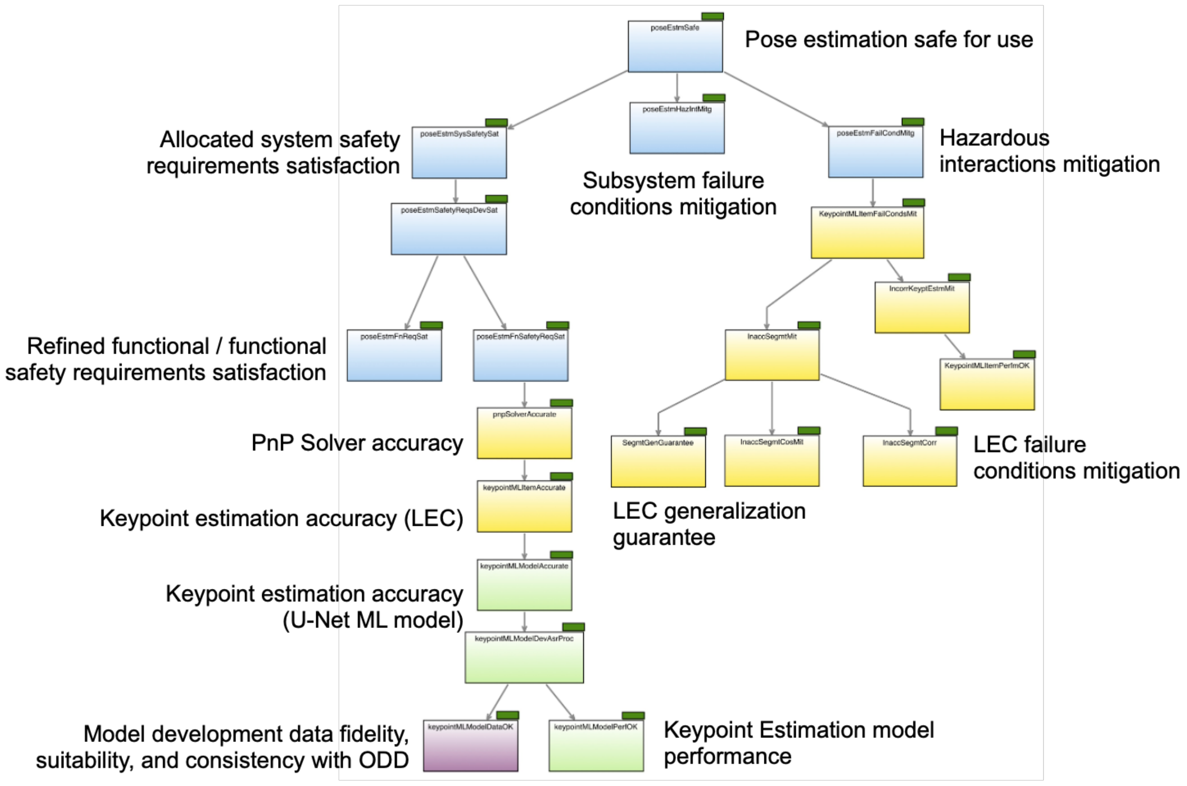

poseEstmSafe
Pose estimation safe for use
poseEstmSysSafetySat
Allocated system safety requirements satisfaction
poseEstmHazIntMitg
Subsystem failure conditions mitigation
poseEstmFailCondMitg
Hazardous interactions mitigation
poseEstmSafetyReqsDevSat
KeypointMLItemFailCondsMit
IncorrKeyptEstmMit
poseEstmFnReqSat
poseEstmFnSafetyReqSat
Refined functional / functional safety requirements satisfaction
InaccSegmtMit
KeypointMLItemPerfmOK
pnpSolverAccurate
PnP Solver accuracy
SegmtGenGuarantee
InaccSegmtCosMit
InaccSegmtCorr
LEC failure conditions mitigation
keypointMLItemAccurate
Keypoint estimation accuracy (LEC)
LEC generalization guarantee
keypointMLModelAccurate
Keypoint estimation accuracy (U-Net ML model)
keypointMLModelDevAsrProc
keypointMLModelDataOK
Model development data fidelity, suitability, and consistency with ODD
keypointMLModelPerfOK
Keypoint Estimation model performance

Figure 16. Architecture of the Safety Argument for Pose Estimation

Figure 17 shows the concrete structure of the assurance argument, depicted graphically using the Goal Structuring Notation (GSN), as well as its relation to the argument architecture. As shown in the figure, the top-level of the argument architecture highlighted by the dotted oval region abstracts the claim decomposition and clarifying contextual information.

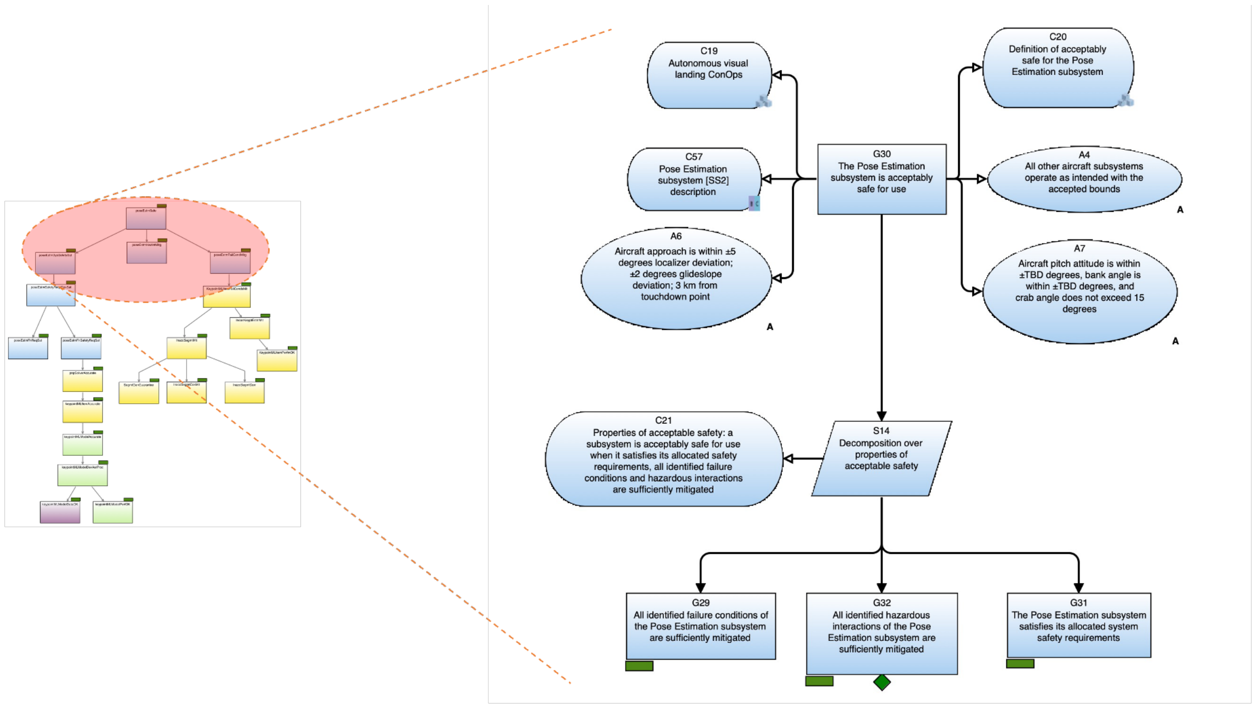
For this report, we focus on the claim associated with satisfying the allocated requirements. As indicated earlier, the main claim for which assurance is required is the functional safety requirement for PosEstm: *The pose estimate of aircraft attitude, location, and velocity shall be consistent with the true aircraft pose*. This can be shown to hold with high confidence[6], when the PosEstm

[6] We have not specified what constitutes "high confidence" in this report. One approach could be to consider "high-confidence" as the 95% or 99% binomial proportion confidence interval for the probability of producing an *accurate*

outputs, i.e., the estimates of the 6-DOF parameter values (i.e., the rotational parameters *roll*, *pitch*, *yaw*, and the translational parameters *surge*, *heave*, *sway*) are consistent with the true aircraft orientation and location in the appropriate reference frame.

In other words, it must be shown with high confidence that PosEstm produces accurate estimates of the 6-DOF parameters. The acceptable uncertainty bounds (or equivalently, the margin of error) in the parameter estimates considered accurate is a part of the contextual information that must be made explicit in the assurance argument (however, we have not defined those bounds in this report).

The outputs of PosEstm are in fact the outputs of the PnP solver. Thus, for the 6-DOF parameter estimates to be accurate, the PnP solver must produce the correct outputs for the given keypoint inputs. More specifically, the PnP solver must correctly transform the keypoints supplied to it, and the keypoints themselves must be both valid (within the set of admissible values) and accurate. Correct PnP transformation requires that no errors are introduced in processing keypoints and producing 6-DOF parameter estimates. That is, the specification of the PnP algorithm is valid, and its implementation is correct with respect to its specification.



C19
Autonomous visual landing ConOps
C20
Definition of acceptably safe for the Pose Estimation subsystem
C57
Pose Estimation subsystem [SS2] description
G30
The Pose Estimation subsystem is acceptably safe for use
A4
All other aircraft subsystems operate as intended with the accepted bounds
A
A6
Aircraft approach is within ±5 degrees localizer deviation; ±2 degrees glideslope deviation; 3 km from touchdown point
A
A7
Aircraft pitch attitude is within ±TBD degrees, bank angle is within ±TBD degrees, and crab angle does not exceed 15 degrees
A
C21
Properties of acceptable safety: a subsystem is acceptably safe for use when it satisfies its allocated safety requirements, all identified failure conditions and hazardous interactions are sufficiently mitigated
S14
Decomposition over properties of acceptable safety
G29
All identified failure conditions of the Pose Estimation subsystem are sufficiently mitigated
G32
All identified hazardous interactions of the Pose Estimation subsystem are sufficiently mitigated
G31
The Pose Estimation subsystem satisfies its allocated system safety requirements

Figure 17. Fragment of Top-level Safety Argument Structure

Accuracy of keypoint inputs, informally, implies a 1-1 correspondence between the points identified in a 2D projection on an image of a 3D scene, and the ground truth. Since the keypoint

(i.e., correct/consistent with true pose, and precise) pose, Pr(Accurate Pose Estimate), where the corresponding probability of failure (to produce an accurate pose estimate) —given as Pr(Pose Estimate Failure) = 1 – Pr(Accurate Pose Estimate)—is not greater than the acceptable TLOS for the *unstable approach* landing hazard.

inputs to the PnP solver are the outputs of an LEC (i.e., an implementation of the U-Net DNN model that realizes the *Runway Localization and Keypoint Estimation* function), keypoint inputs to the PnP solver are accurate when i) the LEC produces accurate keypoint estimates as output in response to a stream of image inputs from the perception sensors; and ii) any errors in LEC-based keypoint estimation are detected and corrected before they are propagated to the PnP solver (note that this corresponds to the safety post processing and runtime assurance recovery barriers captured in the safety architecture shown in Figure 12 and Figure 13).

To show that the U-Net LEC produces accurate keypoint estimates, it has to be shown that: First, the LEC produces accurate keypoint estimates on unseen *in-sample* data, i.e., on image streams collected for validating that LEC behavior is as required, prior to its deployment into operation. The evidence for this includes, for example, the results of verification that for all usage situations specified in the CONOPS, over the entire duration of the intended use, for all images in a sequence of a predetermined length, the LEC produces as output:

- keypoints that always lie within a region that includes the ground-truth keypoints, and the dimensions of that region are within the acceptable margin of error;
- runway instance segment masks that always contain the ground-truth runway instances.

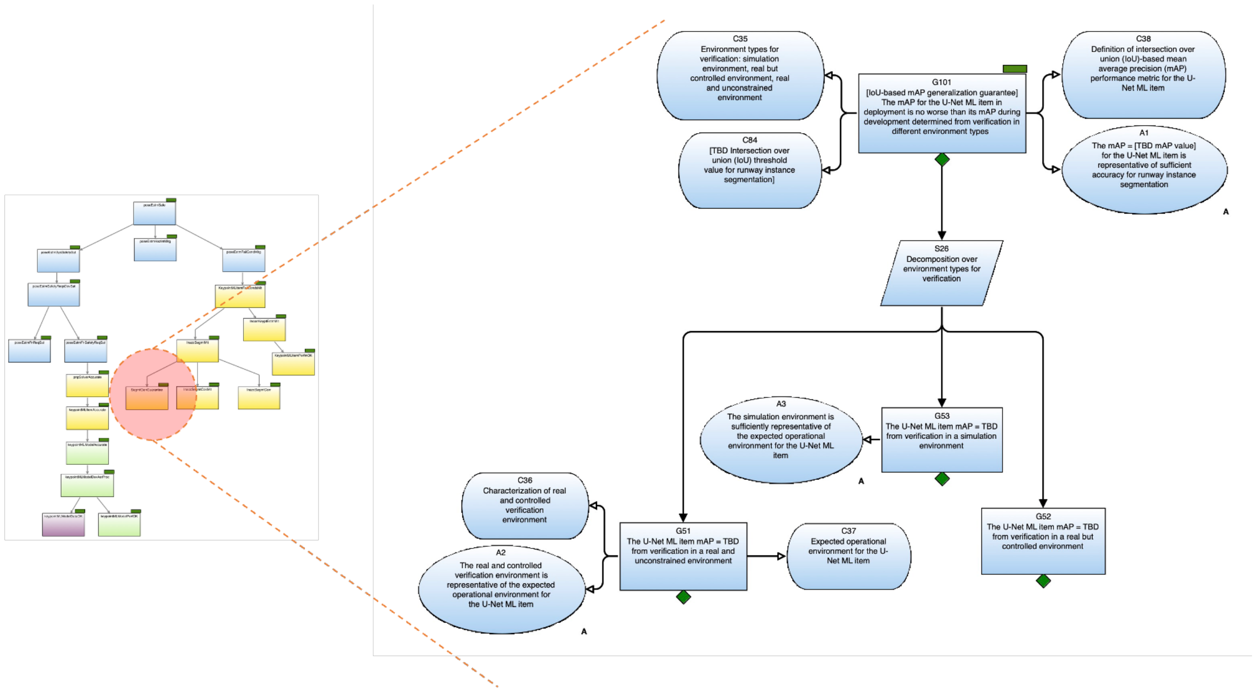

C35
Environment types for verification: simulation environment, real but controlled environment, real and unconstrained environment
C84
[TBD Intersection over union (IoU) threshold value for runway instance segmentation]
G101
[IoU-based mAP generalization guarantee] The mAP for the U-Net ML item in deployment is no worse than its mAP during development determined from verification in different environment types
C38
Definition of intersection over union (IoU)-based mean average precision (mAP) performance metric for the U-Net ML item
A1
The mAP = [TBD mAP value] for the U-Net ML item is representative of sufficient accuracy for runway instance segmentation
A
S26
Decomposition over environment types for verification
A3
The simulation environment is sufficiently representative of the expected operational environment for the U-Net ML item
A
G53
The U-Net ML item mAP = TBD from verification in a simulation environment
C36
Characterization of real and controlled verification environment
A2
The real and controlled verification environment is representative of the expected operational environment for the U-Net ML item
A
G51
The U-Net ML item mAP = TBD from verification in a real and unconstrained environment
C37
Expected operational environment for the U-Net ML item
G52
The U-Net ML item mAP = TBD from verification in a real but controlled environment

Figure 18. Fragment of Safety Argument Invoking Generalization Guarantees

Additional evidence can include quantitative metrics that characterize LEC accuracy performance: for instance, keypoint estimation error rates (which are shown to be no worse than the acceptable error rate) and object keypoint similarity (OKS) based mean average precision (MAP).

Secondly, the LEC behavior exhibited on unseen in-sample data will generalize to unseen, *out-of-sample* data. This constitutes a so-called *generalization guarantee*, for which supporting evidence can include verification of generalization behavior in different verification environments, e.g., in simulation, or a real and constrained environment, or incrementally in a real and unconstrained environment. For this report, we have not defined what precisely constitutes a generalization guarantee, although one possible formulation involves claiming and showing that the OKS-based MAP is no worse in deployment than the values obtained during LEC development. Figure 18 shows a graphical depiction of this argument structure using GSN. This argument also supports, in part, the objective of showing that `PosEstm` does not exhibit unintended behavior.

Lastly, the LEC receives inputs consistent with its operational design domain (ODD)—the specification of the full space of inputs that the LEC is expected to encounter in use in which it must properly function—and any inputs not consistent with its ODD are detected and filtered (note that this corresponds to the runtime assurance prevention barrier captured in the safety architecture shown in Figure 12 and Figure 13). Due to the first and the third items above, we must additionally show that the data sampled for training and validation (during LEC development) is *appropriate*. Figure 19 shows a fragment of the corresponding argument.

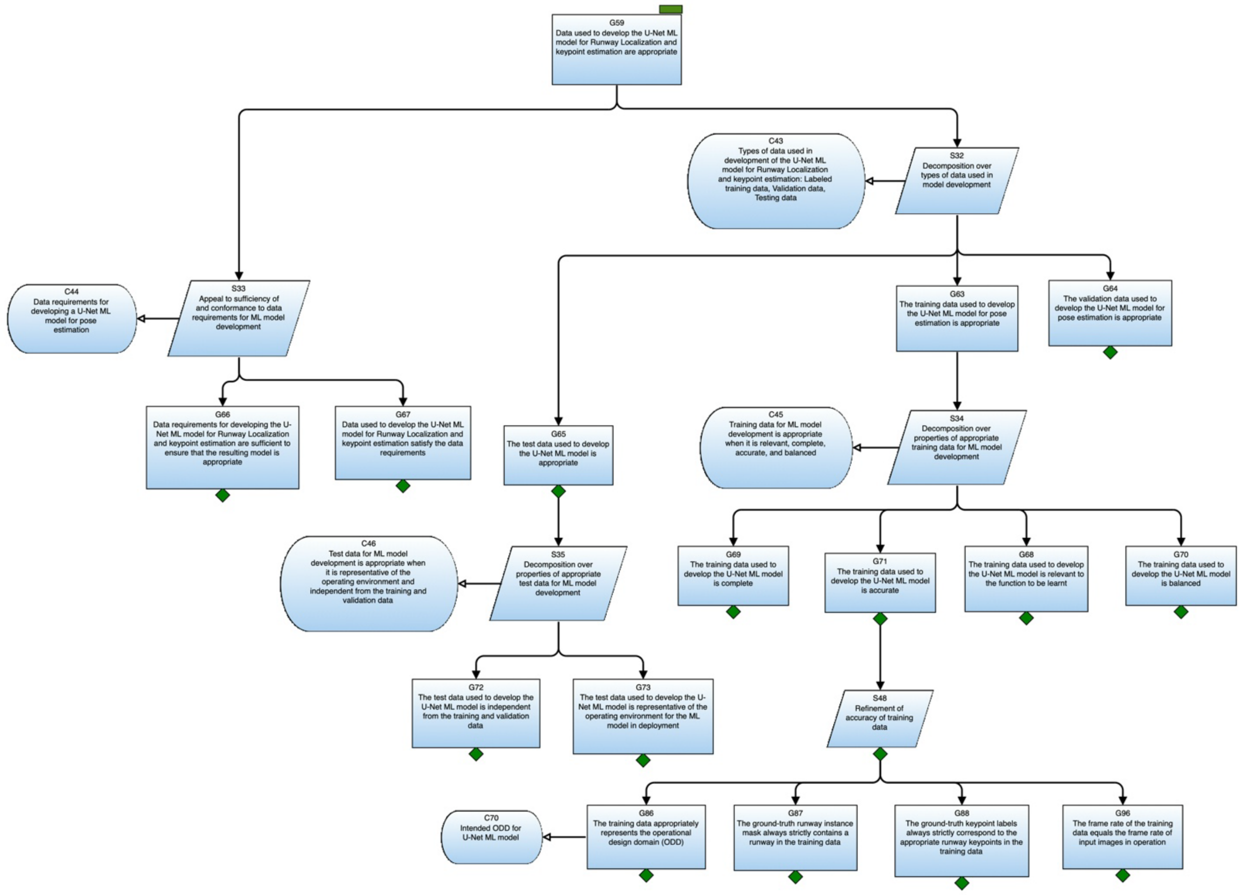

G59
Data used to develop the U-Net ML model for Runway Localization and keypoint estimation are appropriate
C43
Types of data used in development of the U-Net ML model for Runway Localization and keypoint estimation: Labeled training data, Validation data, Testing data
S32
Decomposition over types of data used in model development
C44
Data requirements for developing a U-Net ML model for pose estimation
S33
Appeal to sufficiency of and conformance to data requirements for ML model development
G63
The training data used to develop the U-Net ML model for pose estimation is appropriate
G64
The validation data used to develop the U-Net ML model for pose estimation is appropriate
G66
Data requirements for developing the U-Net ML model for Runway Localization and keypoint estimation are sufficient to ensure that the resulting model is appropriate
G67
Data used to develop the U-Net ML model for Runway Localization and keypoint estimation satisfy the data requirements
G65
The test data used to develop the U-Net ML model is appropriate
C45
Training data for ML model development is appropriate when it is relevant, complete, accurate, and balanced
S34
Decomposition over properties of appropriate training data for ML model development
C46
Test data for ML model development is appropriate when it is representative of the operating environment and independent from the training and validation data
S35
Decomposition over properties of appropriate test data for ML model development
G69
The training data used to develop the U-Net ML model is complete
G71
The training data used to develop the U-Net ML model is accurate
G68
The training data used to develop the U-Net ML model is relevant to the function to be learnt
G70
The training data used to develop the U-Net ML model is balanced
G72
The test data used to develop the U-Net ML model is independent from the training and validation data
G73
The test data used to develop the U-Net ML model is representative of the operating environment for the ML model in deployment
S48
Refinement of accuracy of training data
C70
Intended ODD for U-Net ML model
G86
The training data appropriately represents the operational design domain (ODD)
G87
The ground-truth runway instance mask always strictly contains a runway in the training data
G88
The ground-truth keypoint labels always strictly correspond to the appropriate runway keypoints in the training data
G96
The frame rate of the training data equals the frame rate of input images in operation

Figure 19. Fragment of Safety Argument for Data used for LEC Development

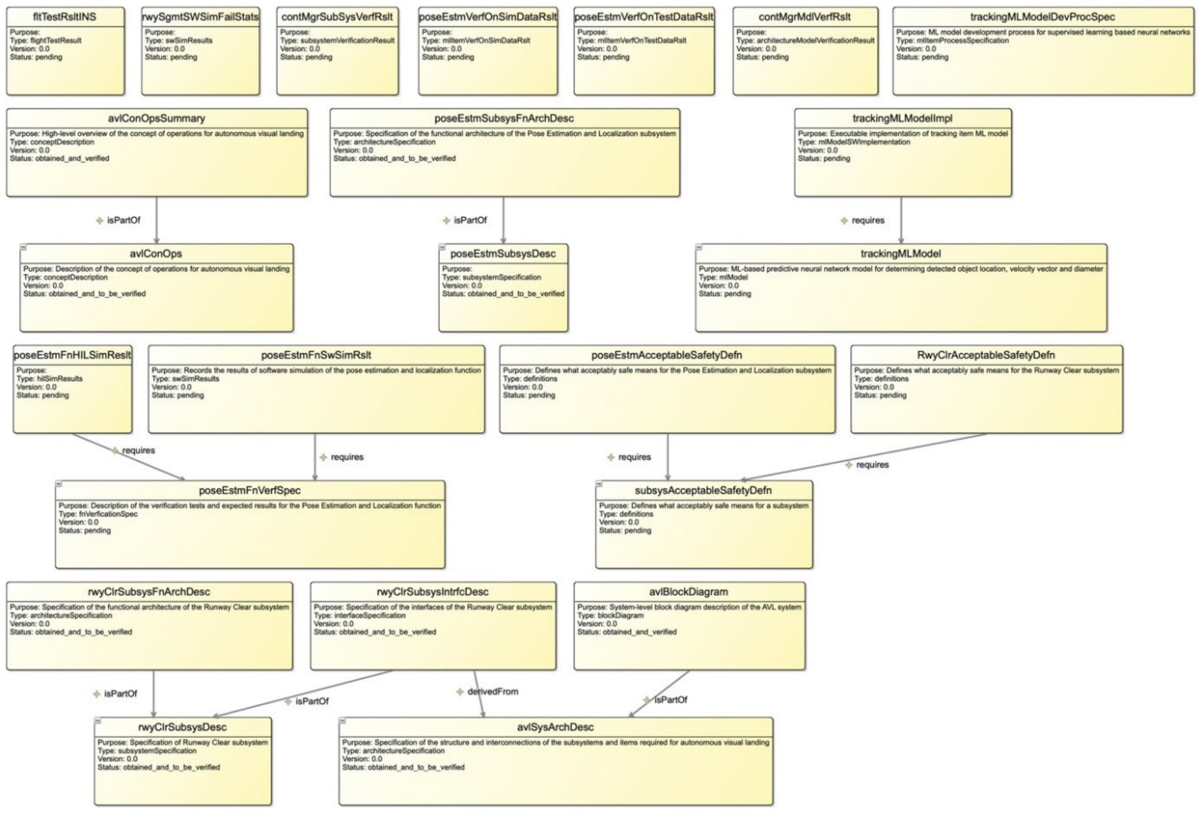
Here, the claim of appropriate data being used to develop the LEC is decomposed and refined by

- Appealing to the concrete data conforming to the specified data requirements for U-Net ML model development, and
- Reasoning over each type of data (i.e., training, validation, and testing) used. For this sub-argument, the following properties of the data are considered: the data are complete, balanced, relevant, and accurate. Accuracy of data in particular can be decomposed into at least the following claims:
  - the data is representative of the ODD for PosEstm;
  - the ground truth runway instance segment mask always strictly contains a runway;
  - the ground truth keypoint labels always strictly correspond to the true runway keypoints in the data; and
  - the frame rate of the data used in LEC development corresponds to the frame rate of the input images in operation.

Evidence

The following are examples of the type of evidence artifacts that can be used to support the assurance argument presented in the preceding discussion:

- Safety architecture design including architectural mechanisms such as runtime monitoring, and function output correction;



fltTestRsltINS
Purpose:
Type: flightTestResult
Version: 0.0
Status: pending
rwySgmtSWSimFailStats
Purpose:
Type: swSimResults
Version: 0.0
Status: pending
contMgrSubSysVerfRslt
Purpose:
Type: subsystemVerificationResult
Version: 0.0
Status: pending
poseEstmVerfOnSimDataRslt
Purpose:
Type: mlItemVerfOnSimDataRslt
Version: 0.0
Status: pending
poseEstmVerfOnTestDataRslt
Purpose:
Type: mlItemVerfOnTestDataRslt
Version: 0.0
Status: pending
contMgrMdlVerfRslt
Purpose:
Type: architectureModelVerificationResult
Version: 0.0
Status: pending
trackingMLModelDevProcSpec
Purpose: ML model development process for supervised learning based neural networks
Type: mlItemProcessSpecification
Version: 0.0
Status: pending
avlConOpsSummary
Purpose: High-level overview of the concept of operations for autonomous visual landing
Type: conceptDescription
Version: 0.0
Status: obtained_and_verified
poseEstmSubsysFnArchDesc
Purpose: Specification of the functional architecture of the Pose Estimation and Localization subsystem
Type: architectureSpecification
Version: 0.0
Status: obtained_and_to_be_verified
trackingMLModelImpl
Purpose: Executable implementation of tracking item ML model
Type: mlModelSWImplementation
Version: 0.0
Status: pending
isPartOf
isPartOf
requires
avlConOps
Purpose: Description of the concept of operations for autonomous visual landing
Type: conceptDescription
Version: 0.0
Status: obtained_and_to_be_verified
poseEstmSubsysDesc
Purpose:
Type: subsystemSpecification
Version: 0.0
Status: obtained_and_to_be_verified
trackingMLModel
Purpose: ML-based predictive neural network model for determining detected object location, velocity vector and diameter
Type: mlModel
Version: 0.0
Status: pending
poseEstmFnHILSimReslt
Purpose:
Type: hilSimResults
Version: 0.0
Status: pending
poseEstmFnSwSimRslt
Purpose: Records the results of software simulation of the pose estimation and localization function
Type: swSimResults
Version: 0.0
Status: pending
poseEstmAcceptableSafetyDefn
Purpose: Defines what acceptably safe means for the Pose Estimation and Localization subsystem
Type: definitions
Version: 0.0
Status: pending
RwyClrAcceptableSafetyDefn
Purpose: Defines what acceptably safe means for the Runway Clear subsystem
Type: definitions
Version: 0.0
Status: pending
requires
requires
requires
requires
poseEstmFnVerfSpec
Purpose: Description of the verification tests and expected results for the Pose Estimation and Localization function
Type: fnVerficationSpec
Version: 0.0
Status: pending
subsysAcceptableSafetyDefn
Purpose: Defines what acceptably safe means for a subsystem
Type: definitions
Version: 0.0
Status: pending
rwyClrSubsysFnArchDesc
Purpose: Specification of the functional architecture of the Runway Clear subsystem
Type: architectureSpecification
Version: 0.0
Status: obtained_and_to_be_verified
rwyClrSubsysIntrfcDesc
Purpose: Specification of the interfaces of the Runway Clear subsystem
Type: interfaceSpecification
Version: 0.0
Status: obtained_and_to_be_verified
avlBlockDiagram
Purpose: System-level block diagram description of the AVL system
Type: blockDiagram
Version: 0.0
Status: obtained_and_verified
isPartOf
isPartOf
derivedFrom
isPartOf
rwyClrSubsysDesc
Purpose: Specification of Runway Clear subsystem
Type: subsystemSpecification
Version: 0.0
Status: obtained_and_to_be_verified
avlSysArchDesc
Purpose: Specification of the structure and interconnections of the subsystems and items required for autonomous visual landing
Type: architectureSpecification
Version: 0.0
Status: obtained_and_to_be_verified

Figure 20. Evidence Dependency Graph for Pose Estimation Assurance

- Specifications of the ODD, operational envelope, and concept of operations;
- Results of verification of properties applicable to the ML model and its implementation (i.e., ML *item*) such as those involving pixel-level keypoint estimation accuracy, segmentation mask accuracy, keypoint to segmentation mask relations, and robustness;
- Subsystem architecture verification results including properties concerning the accuracy in correcting errors in keypoint estimation and segmentation masks, worst case execution time (WCET), and navigation state accuracy;
- Testing-based statistics on ML model and item performance metrics on properties such as inference accuracy;
- Evidence of data accuracy, representativeness, coverage, completeness, and relevance; and
- Verification results of ML model to implementation transformation fidelity.

Note that the above are not a comprehensive list. Figure 20 shows an *evidence dependency graph*, as constructed within AdvoCATE, that indicates some of the concrete evidence items and their interrelations. For more details on the specifics of the graph and how evidence is captured and represented in the tool, see Section 4.4.7.

4.1.2 Undersea Domain Challenge Problem – Obstacle Avoidance

NG CP 4

Challenge problem summary

Assurance Objectives

Assurance Case Architecture

Assurance Case Elements

4.2 Assurance Measures for Challenge Problems

For Phase 3, we developed models for assuring the collision avoidance and degradation detection LECs for the undersea domain platform and the object detection and pose estimation LECs for the air domain platform.

4.2.1 Undersea Domain

For Phase 3, we have focused on two challenge problems (CPs): CP4 (obstacle avoidance), and CP6 (operating under degraded modes)

4.2.1.1 CP4 – Obstacle Avoidance

CP4 involves a search-and-rescue scenario in which an autonomous underwater vehicle (AUV) must perform a ladder search while avoiding obstacles. The framework for the high-level assurance measure for this problem is shown below along with the details of the assurance measure we developed for the collision avoidance LEC.

High-level Assurance Measure

The mission objectives for CP4 are as follows:

- In-distribution input signal from Forward Looking Sonar (FLS)
- Successful obstacle avoidance
- Grid search completion (>95% of waypoints reached)
- Loiter completion (>1/3 of search time for loiter)
- Return to surface safely.

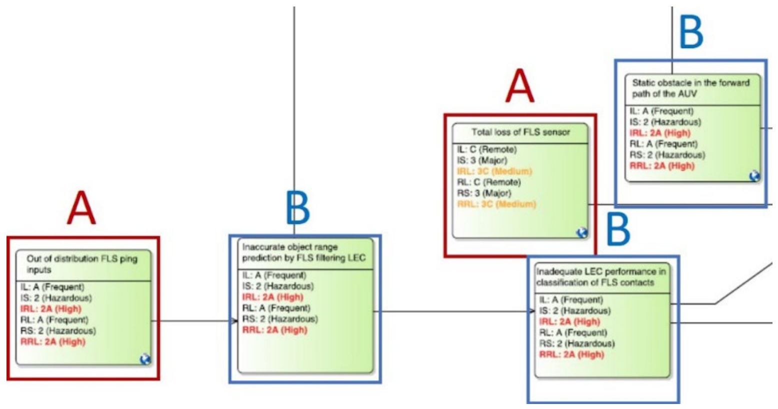


Figure 21. Partial Safety Architecture with Related Mission Objectives

Each of these mission objectives was linked to various events in the safety architecture. A small portion of the safety architecture with associated mission objectives is shown in Figure 21. For example, mission objective A (obstacle avoidance) would be negatively impacted in the event that an inaccurate object range was computed by the FLS filtering LEC. In this instance, the AUV may not have time to be rerouted around the obstacle. The links between events in the safety architecture yield a graphical representation of the dependencies of the mission objectives as shown in Figure 22. The overall assurance measure is the joint probability of all mission objectives and may be computed from the graph.

$$P(A, B, C, D) = P(A)P(A)P(A, B)P(D|A, C) \tag{1}$$

$P(A)$ is computed via the low-level assurance measure described in the next section. The conditional probabilities are computed from data generated from the simulator.

Low-level Assurance Measure

The low-level assurance measure was designed based on an *assurance property*, a quantifiable measure for success or failure, for collision avoidance. The assurance property for this problem is

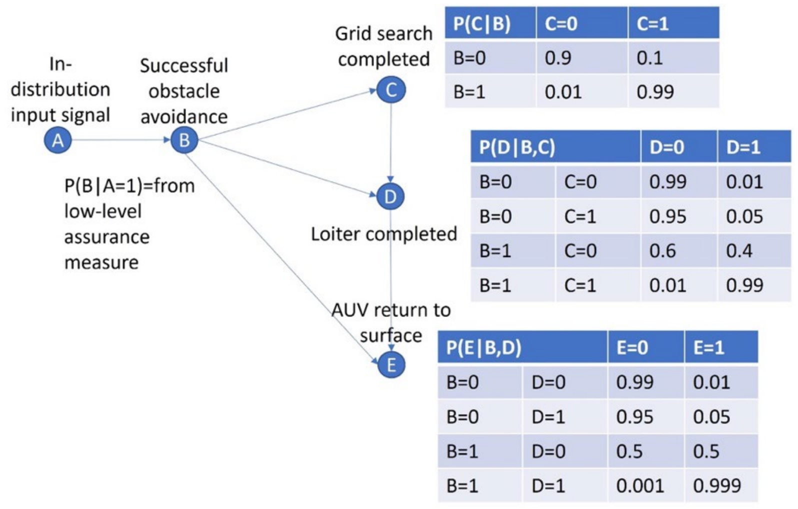
$$P((R_{obs}(t = T{:}T + n) \leq \text{CPA} \mid I(t = 0{:}T)) \geq C \tag{2}$$

where $R_{obs}(t)$ is the range to obstacle at time t, CPA is the closest point of approach (10m), C is the confidence threshold (0.05), and $I(t = 0{:}T)$ is the sequence of observations until time T of inputs to the control LEC: closest point of approach, obstacle range from the FLS, estimated object size.

The collision avoidance LEC is a reinforcement learning model that determines the appropriate heading-speed-depth (HSD) command based on the current state data, as follows:

- Current range to obstacle
- Current closest point of approach to obstacle given the current heading and depth
- Estimated object size.

For assurance measure development, we analyze the inputs to and outputs of the LEC for 5 time steps. Analyzing the inputs allows us to determine whether the AUV is in a familiar environment. In previous phases, we have accomplished this task using autoencoders and inductive conformal prediction. For this phase, we have utilized evidential regression models, which allow for quantification of aleatoric (data or input) uncertainty as well as epistemic (inference) uncertainty simultaneously. Additionally, the evidential model replaced the ensemble of Bayesian neural networks used in the previous phase for inference.


In-distribution input signal
A
Successful obstacle avoidance
B
P(B|A=1)=from low-level assurance measure
Grid search completed
C
Loiter completed
D
AUV return to surface
E

P(C|B)	C=0	C=1
B=0	0.9	0.1
B=1	0.01	0.99

P(D|B,C)		D=0	D=1
B=0	C=0	0.99	0.01
B=0	C=1	0.95	0.05
B=1	C=0	0.6	0.4
B=1	C=1	0.01	0.99

P(E|B,D)		E=0	E=1
B=0	D=0	0.99	0.01
B=0	D=1	0.95	0.05
B=1	D=0	0.5	0.5
B=1	D=1	0.001	0.999

Figure 22. Graphical Model for Overall Assurance Measure for CP4

Evidential models place priors over the likelihood function instead of network weights as shown in Figure 4. The model is trained to predict the parameters of an evidential distribution, such that the total loss includes model fit in terms of negative log likelihood plus the regularization of evidence scaled by the error of the prediction [ASSR2020]. This allows for quantification of both aleatoric (input distribution) and epistemic (LEC performance) uncertainty. The architecture for the model for the assurance measure is as follows:

- Dense layer, size 128, rectified linear unit (ReLU) activation function
- Dense layer, size 64, ReLU activation function
- Dense layer, size 32, ReLU activation function
- Dense layer, size 16, ReLU activation function
- Dense normal gamma layer (evidential layer)

The model was trained on a data set of 72,835 cases at a learning rate of 10^{-4} for 100 epochs and was tested on a data set of 87,898 cases.

As the LEC provides a heading, speed, and depth (HSD) command in lieu of a prediction of collision, the assurance measure was designed to predict a future collision and to provide an alert in the event that a collision was deemed imminent. Therefore, the output of the assurance measure was the probability of a collision occurring at $t + 5$, where t is the current time step. To obtain data for training and testing, we ran the AUV simulator provided by NG in a mission configuration that led the AUV in a ladder formation while providing randomly generated obstacles to avoid.

Table 1 shows the performance of the assurance measure with the model type (ensemble of Bayesian neural networks) used in Phase 2 along with an evidential neural network and an evidential *long short-term memory* (LSTM) network. The results show that the evidential neural network achieved higher sensitivity and specificity while performing inference faster than the ensemble of BNNs.

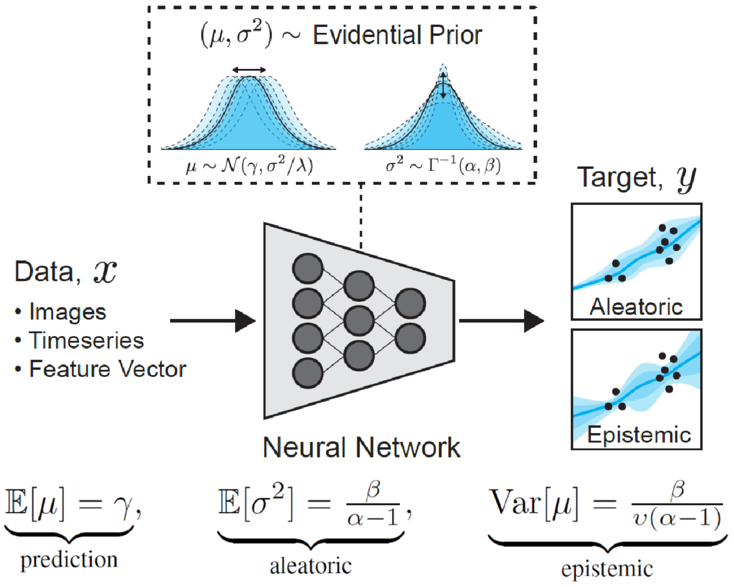

$(\mu, \sigma^2) \sim$ Evidential Prior
$\mu \sim \mathcal{N}(\gamma, \sigma^2/\lambda)$
$\sigma^2 \sim \Gamma^{-1}(\alpha, \beta)$
Target, y
Data, x
• Images
• Timeseries
• Feature Vector
Aleatoric
Epistemic
Neural Network
$\underbrace{\mathbb{E}[\mu] = \gamma,}_{\text{prediction}}$
$\underbrace{\mathbb{E}[\sigma^2] = \frac{\beta}{\alpha - 1},}_{\text{aleatoric}}$
$\underbrace{\mathrm{Var}[\mu] = \frac{\beta}{v(\alpha - 1)}}_{\text{epistemic}}$

Figure 23. Deep Evidential Regression

Two specific scenarios were provided for which the assurance technologies were to be tested. The first scenario, seen in Figure 24, involved an obstacle being too close to a way point. For this scenario, we allow the collision avoidance LEC a certain amount of time to reach the waypoint while trying to avoid the obstacle, but if too much time passes without achieving this goal, we signal the system to move on to the next waypoint. This scenario has been tested on the *Iver* AUV in water, and the assurance technology was successful in directing the Iver AUV to avoid the obstacle.

Table 1. Assurance Measure Results for CP4 (Obstacle Avoidance)

Assurance Measure Model	AUC	Sensitivity	Specificity	False Negative Rate	Inference Time
Ensemble of BNNs	0.9787	0.9898	0.9692	0.0102	6.677ms
Evidential NN	0.9980	0.9968	0.9977	0.0032	1.572ms
Evidential LSTM	0.9889	0.9973	0.9889	0.0027	4.816ms

Figure 25 shows the second scenario with a wall of obstacles between the AUV and a waypoint. The ideal outcome here is that the assurance technology would tell the AUV to return home in lieu of potentially getting lost in the debris field. The decision component of the assurance technology for this scenario instructed the AUV to move on to the next waypoint as before, and if the AUV is still too close to obstacles after a certain amount of time, the assurance technology instructs the AUV to return home. This scenario was tested in simulation only during which the AUV attempted a ladder search in the presence of obstacles in nominal and degraded modes.

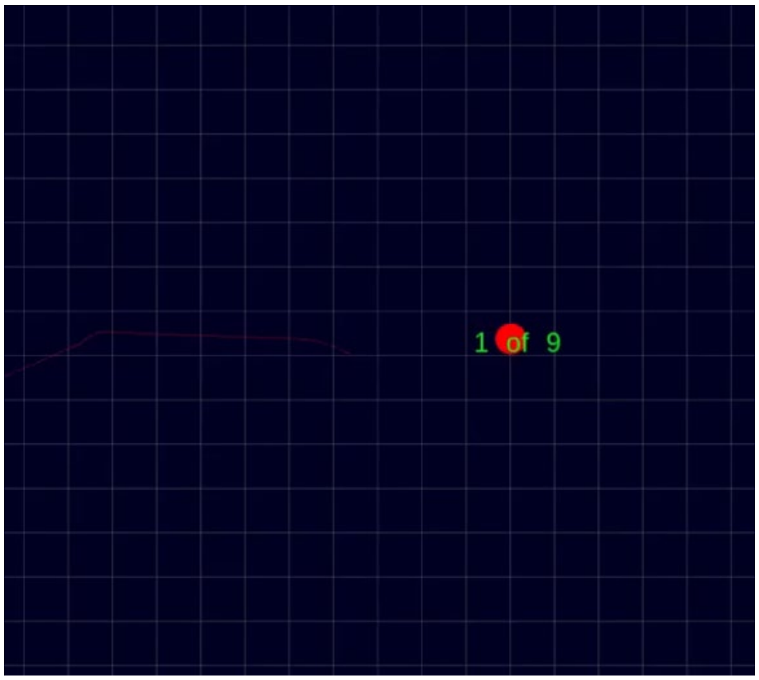


Figure 24. CP4 Scenario with Obstacle on Waypoint 1

The scenario in Figure 24 induces a collision when no assurance technology is being used as the AUV turns back to the waypoint and obstacle after initially avoiding the obstacle and is too close to redirect.

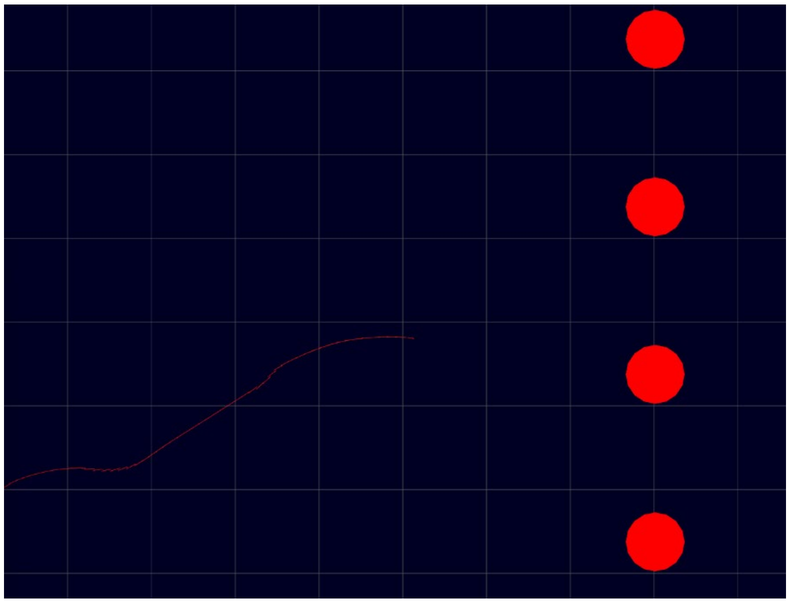


Figure 25. CP4 Scenario with Wall of Obstacles

The scenario in Figure 25 induces a collision as the AUV focuses on a single obstacle at a time and in avoiding one obstacle becomes too close to the others.

4.2.1.2 CP6 – Operating Under Degraded Modes

CP6 introduced fin degradation to the AUV. In this problem, a new LEC that determined the amount of disturbance was provided.

High-level Assurance Measure

For CP6, we add an additional node to the graph for the high-level assurance measure for CP4, including the mission objective of successful disturbance detection. Figure 7 shows this addition as mission objective E. The LEC behind mission objective E determines whether the collision avoidance LEC is utilizing the appropriate model weights for the given environment and, thus, directly affects mission objective A as shown. The updated overall assurance measure is then computed.

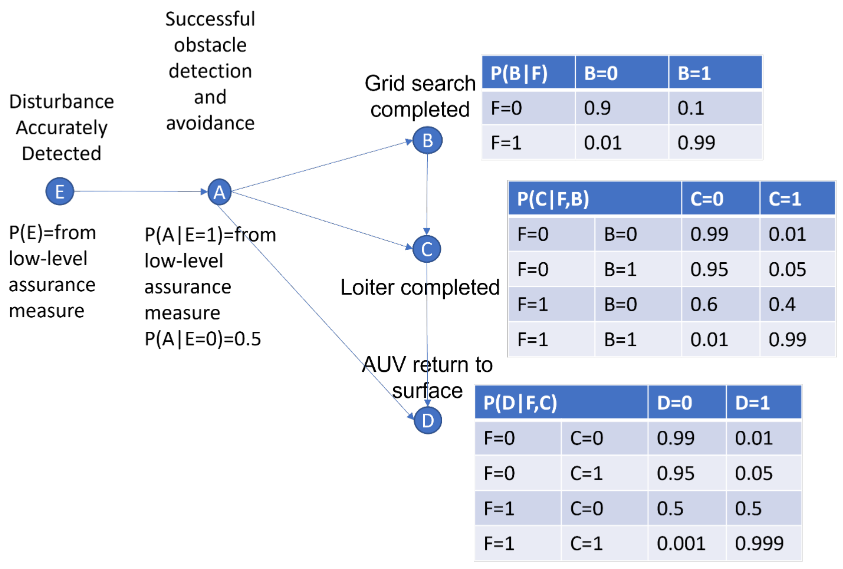
$$P(A,B,C,D,E) = P(E)P(E)P(A)P(A,B)P(D|A,C) \tag{3}$$

Given a successful mission objective E, $P(A)$ continues to be computed via the assurance measure for CP4. Otherwise, $P(A)$ is the expectation of a uniform distribution, 0.5.

Low-level Assurance Measure

The disturbance detection LEC analyzed fin outputs and state data to determine the amount of degradation and to instruct the collision avoidance LEC which weights to use. A set of weights were trained with the AUV in a degraded mode to compensate for the disturbance.

The assurance measure for this problem was developed in two steps. First, a model for determining degraded versus nominal mode was trained using state data. Then, a model for determining whether the appropriate weights were being applied was trained using fin inputs as well. Data for training (128,790 cases) and testing (77,373 cases) of both models was obtained using the provided simulator. The models used were evidential regression models, and performance metrics are shown in Table 2.


Disturbance Accurately Detected
E
P(E)=from low-level assurance measure
Successful obstacle detection and avoidance
A
P(A|E=1)=from low-level assurance measure P(A|E=0)=0.5
Grid search completed
B
C
Loiter completed
AUV return to surface
D

P(B|F)	B=0	B=1
F=0	0.9	0.1
F=1	0.01	0.99

P(C|F,B)		C=0	C=1
F=0	B=0	0.99	0.01
F=0	B=1	0.95	0.05
F=1	B=0	0.6	0.4
F=1	B=1	0.01	0.99

P(D|F,C)		D=0	D=1
F=0	C=0	0.99	0.01
F=0	C=1	0.95	0.05
F=1	C=0	0.5	0.5
F=1	C=1	0.001	0.999

Figure 26. Graphical Representation of Overall Assurance Measure for CP6

The architecture for the model determining nominal versus degrade mode was as follows:

- Long-term short-term memory (LSTM) layer of size 128
- Dense layer of size 64, ReLU activation function
- Dense normal gamma layer

The model was trained at a learning rate of 10^{-4} for 300 epochs with a batch size of 100. The architecture for the model determining correct weights was as follows:

- Dense layer of size 256, ReLU activation function
- Dense layer of size 128, ReLU activation function
- Dense layer of size 32, ReLU activation function
- Dense normal gamma layer

The model was trained at a learning rate of 7×10^{-5} for 500 epochs with a batch size of 1,000.

Table 2. Performance of Assurance Measure Models for CP6

Assurance Measure Model	AUC	Sensitivity	Specificity
Model for Mode	0.96	0.98	0.92
Model for Weights	0.99	1.0	0.99

4.2.2 Air Domain

For the air domain, CP 3.1 involved safe landing using two LECs for perception (see Section 4.1.1 for more details). The first LEC (CP 3.1.1) detects vehicles on the runway for tracking and obstacle avoidance. The second LEC (CP 3.1.2) performs pose estimation by identifying key points on the runway.

4.2.2.1 High-level Assurance Measure

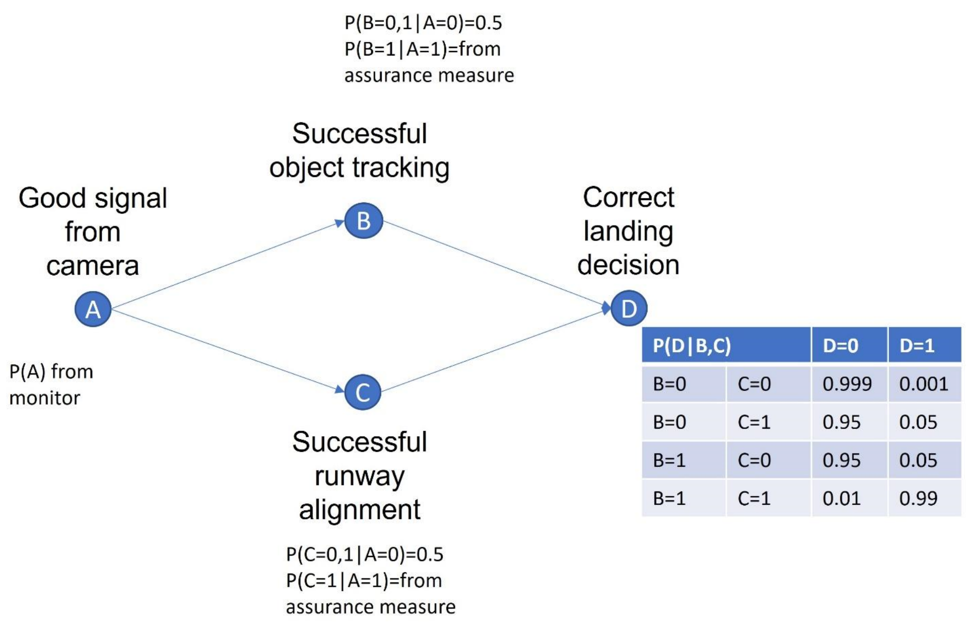
Mission objectives for this challenge problem include:

- Good input signal from camera
- Successful object tracking
- Successful pose estimation
- Correct landing decision

As for the previous CPs, we linked each mission objective to events in the safety architecture and generated a probabilistic graphical model as shown in Figure 27 and resulting in the following formula for computing the high-level assurance measure:

$$P(A,B,C,D) = P(A)P(A)P(A)P(D|B,C) \tag{4}$$

If mission objective A is successful, $P(B)$ and $P(C)$ are computed via the assurance measures for B and C, respectively. Otherwise, they are the expectation of a uniform distribution, 0.5.


P(B=0,1|A=0)=0.5
P(B=1|A=1)=from assurance measure
Successful object tracking
Good signal from camera
Correct landing decision
A
B
C
D
P(A) from monitor
Successful runway alignment
P(C=0,1|A=0)=0.5
P(C=1|A=1)=from assurance measure

P(D|B,C)		D=0	D=1
B=0	C=0	0.999	0.001
B=0	C=1	0.95	0.05
B=1	C=0	0.95	0.05
B=1	C=1	0.01	0.99

Figure 27. Graphical Model for Overall Assurance Measure for CP3.1

4.2.2.2 Low-level Assurance Measure for CP 3.1.1

The assurance measure for CP3.1.1 was designed to monitor the performance of the object detection LEC. The initial model design was for a semantic segmentation model that would output uncertainty per pixel. This model, however, was not providing desired results. Therefore, we altered the model and used a more general approach. The assurance measure model was designed to determine the presence or absence of a vehicle in a given sub-image.

Data for this model was taken using the bounding boxes of ground truth annotations +30 pixels on each side. The cropped images were then resized to $128\times128\times3$. Negative examples were taken at random outside of the bounding box areas. The model architecture is as follows:

- Dense layer of size 256, ReLU activation function
- Dense layer of size 128, ReLU activation function
- Dense layer of size 64, ReLU activation function
- Dense layer of size 32, ReLU activation function
- Dense normal gamma layer

The model was trained on 2,000 cropped images and was tested on 1,000. The model was trained for 10,000 epochs at a learning rate of 5×10^{-6}. The model achieved a sensitivity of 0.95, a specificity of 0.93, and an AUC of 0.98.

4.2.2.3 Low-level Assurance Measure for CP 3.1.2

The assurance measure for CP3.1.2 was designed to monitor the performance of the key point detection LEC for pose estimation. The planned model architecture was a deep evidential model incorporating convolutional layers, pooling and dense layers. Due to time constraints, this model was not trained or tested.

4.3 Formal Methods Integration

Tools for verifying neural networks are often complex and their usage subject to many constraints, contextual dependencies, and assumptions. This can make both performing the verification and the subsequent integration of the results into an assurance case challenging, especially for non-expert users. To ensure this, we need to demonstrate that the verification tool is itself trustworthy and that it has been correctly used for the specific application.

We describe an approach to incorporating results from verification tools in an assurance case by capturing the verification method information in a formally defined assurance case. In particular, we specify the verification steps for using the tool and capture their usage constraints. We capture the input evidence type needed by the tool and output evidence type provided by the tool. Then we specify the usage constraints over these evidence types in the form of parametrized assumptions and guarantees. These tool specifications are application independent. Whenever a tool is used, we capture each individual tool usage by instantiating the parameters in its tool specification regarding input and output evidence types. This allows us to also instantiate and possibly evaluate the assumption and guarantee statements. To incorporate the usage results in an assurance argument, we defined a common tool assurance pattern, which can be customized for each tool. Each tool pattern can be instantiated for specific tool usages. The tool input and output evidence types act as parameters of the argument pattern, and they get instantiated by the values from specific tool usages. This allows for semi-automated creation of an assurance argument fragment for each usage. In the case of a chain of tools, the instantiation can be recursive so that it automatically instantiates and connects patterns of all the tools in the chain, allowing for generation of a more comprehensive argument. The generated argument fragment presents the obtained results and the usage constraints underpinning its validity.

In the remainder of the section, we present the integrations of verification performed with VerifAI in Challenge Problems 1.1 and 2.1, and Venus in Challenge Problem 2.2 For each tool, we first present a tool overview, then we present the corresponding tool specification with the usage constraints capturing the assurance challenges for that tool. Finally, we present a fragment of the challenge problem specific assurance case addressing the application of the tool to that challenge problem.

4.3.1 VerifAI

4.3.1.1 Tool Overview

The VerifAI toolkit implements a methodology for reliable design of systems that include ML components. In particular, the toolkit supports modelling of the system, its requirements and environment, as well as analysis, debugging and improving the system design that includes the ML component.

The top-level architecture of VerifAI is shown in Figure 28. The VerifAI approach is composed of four main modules and an external simulator:

1. *Abstract feature space and Scenic modelling language:* The environment of the ML component is modelled in the Scenic probabilistic environment modelling language. This environment of an ML component consists of the system environment and the system configurations of interest. The parameter ranges can be either explicitly stated or using probabilities in Scenic.

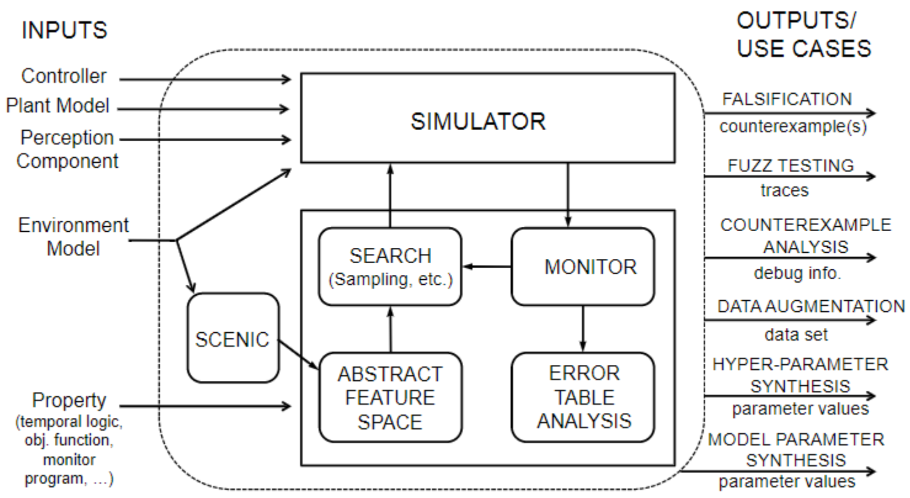


Figure 28: Structure and Operation of VerifAI [VAI2019]

2. *Searching/sampling the feature/environment space:* VerifAI uses the environment model specified in Scenic to generate test vectors, i.e., concrete scenes where each parameter has an assigned value based on the corresponding distributions assigned in Scenic program. VerifAI supports different ways of searching/sampling the environment space from the Scenic program.
3. *External simulator:* VerifAI can be coupled with different simulators. The selected simulator is used to simulate the system for each test vector. The simulator should include a model of the system, with the controller and the ML component, as well as the environmental model as specified in the corresponding Scenic program.
4. *Property monitor:* A monitor is used to check if each simulation satisfies or violates the system-level specification.
5. *Error table analysis:* The violations of the system-level specification are recorded in an error table for automated analysis.

Scenic Modeling Language

A scenic program can be used to describe environments of autonomous systems in terms of features, where each feature can have different values. The probabilistic nature of Scenic allows the assignment of distributions to the features describing an environment so that it is possible to say that a particular feature is more likely to have one value over another. Furthermore, on top of the feature distributions, a Scenic program can also contain constraints over features. An environment obtained by selecting a value for each feature is called a scene. Generating scenes from a Scenic program requires sampling from the distributions defined in the program, while respecting the stated constraints. If a feature doesn't have an explicit distribution assigned to it, then uniform distribution is assumed. Scenes are also referred to as test vectors or feature vectors. Feature can be weather, time, car model, car color, car heading, or car position, where their values can be, for example, neutral, clear, smog, clouds for weather, or 00:00 to 24:00 for time.

When used for verification of a perception ML component, the scenes from the Scenic program are used to generate realistic images by a simulator. Those images are used to test the perception component. The test result returns a correct or incorrect label based on the performance of the object detector component on the generated image. The correct/incorrect result for an image is then associated with the corresponding scene. The data set of scenes and their evaluations can be used in different ways to ultimately improve the performance of the perception component.

Sampling the Environment Space

The environment space can be sampled using static or active sampling techniques. The static sampling techniques rely on defining fixed distributions in the Scenic program. In contrast, active sampling techniques change how scenes are generated over time in response to feedback from earlier tests. To enable active sampling techniques to be applied on a Scenic program, Scenic is extended with parameters that are assigned by an external sampler. This allows Scenic to be used together with external sampling techniques, be it optimization or other algorithms to search the corresponding parameter space.

Monitoring

VerifAI supports monitoring MTL properties using the py-metric-temporal-logic python package. The user can also specify a custom monitor as a python function. The monitor evaluates the simulation with respect to the MTL property. The monitor takes a trace from the simulator and evaluates the MTL property over the trace. The MTL monitor does not just establish if the trace satisfies the formula or not, but it evaluates the robustness of the formula.

The robustness level is taken to be the greatest deviation from the formula achieved over the whole simulation. That is, if the robustness level is ≥ 0, the formula is satisfied, and if it is negative, the formula is not satisfied for the given trace. A greater robustness level means that a small disturbance in the simulation is not likely to invalidate the formula. If the robustness level is close to 0, it means that a small disturbance to the execution that resulted in the given trace may cause the property not to be satisfied. Negative robustness means that the formula is not satisfied, and lower number represents greater deviation from satisfying the formula.

The results of the monitor can be used to output the falsifying traces where the property is violated. The results could also be used to direct the search/sampling of the environment/feature space towards identifying scenes that will most likely result in a satisfying or violating the monitored property.

Refining Scenic Programs

Here we discuss refining Scenic programs to support training, debugging, and testing perception components based on monitoring results. The aim of the approach in [VAI2020] is to find rules for refining the Scenic program in order to maximize the probability of generating scenes that will have the same label. For example, one rule may maximize the chances of generating scenes where the perception component performs well, and another rule where the perception component performs poorly. These rules are used to refine the original Scenic program by adding the rules as program constraints. Such refined Scenic programs can that be used to generate scenes to further train, debug and test the perception component.

Error table analysis

The detected violations of the monitored properties (the counterexamples) are stored in a data structure called the error table. The rows of the table are counterexamples, and the columns are the Scenic abstract features. Each row presents a test vector that led to a violation of the monitored property. Such error table can be used offline for debugging or online to drive the search/sampling algorithms towards specific areas of the environment/feature space. VerifAI supports different techniques for the analysis of the error tables, depending on what is the end use (e.g., counterexample analysis or data set augmentation).

4.3.1.2 Tool application in Challenge Problems

***VerifAI Re-training for Centerline Tracking using TaxiNet* (*CP 1.1*)**

TaxiNet is an experimental autonomous aircraft taxiing system developed by Boeing. The system uses a neural network to estimate the aircraft's position from a camera image. A controller then steers the plane to track the centerline. The main TaxiNet centerline tracking requirement is that "TaxiNet shall keep the aircraft within 1.5m of the runway centerline during taxiing". To establish the quantitative target for this requirement, we consider human pilot performance with the same aircraft. For a Cessna Caravan, the aircraft will remain within 6 feet of the centerline 95% of the time. Hence, the probability at which the system meets the requirement should not be less than 95%.

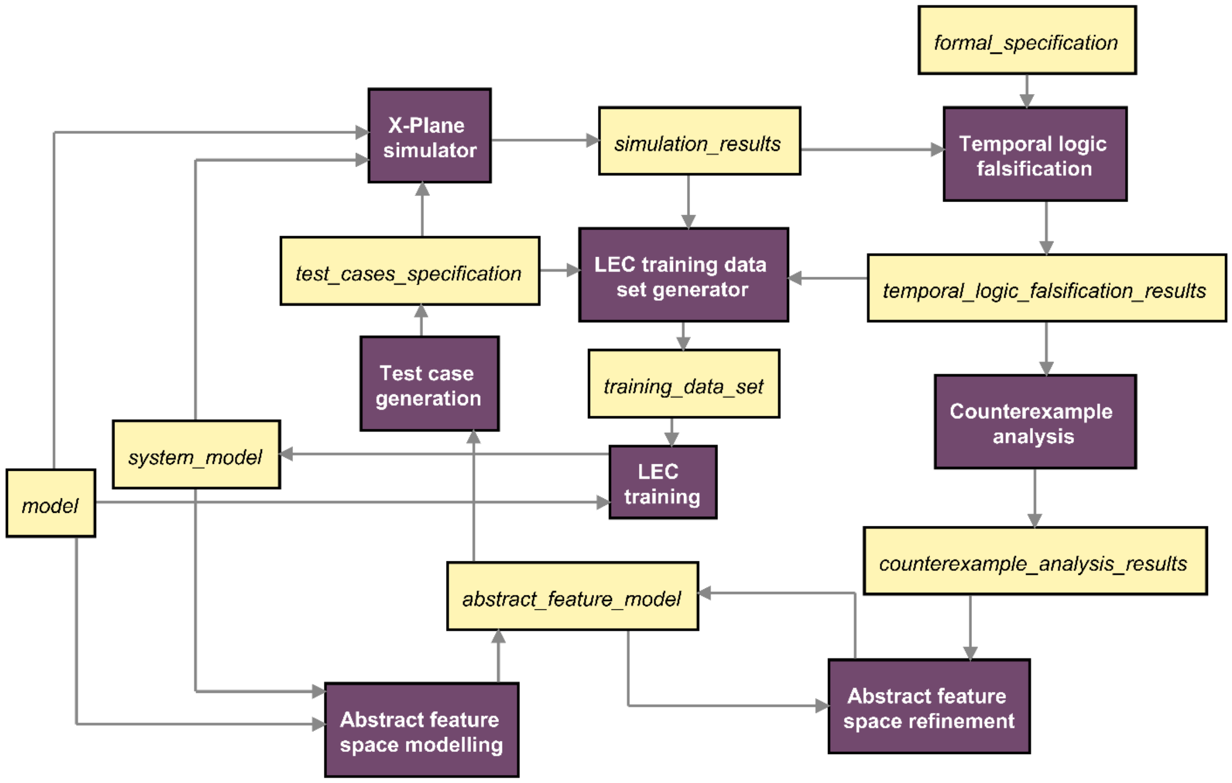

formal_specification
X-Plane simulator
simulation_results
Temporal logic falsification
test_cases_specification
LEC training data set generator
temporal_logic_falsification_results
Test case generation
training_data_set
Counterexample analysis
system_model
LEC training
model
counterexample_analysis_results
abstract_feature_model
Abstract feature space refinement
Abstract feature space modelling

Figure 29. Tools Integration Diagram for VerifAI Testing and Retraining

The following represents the VerifAI workflow for verifying the centerline property within CP 1.1 (captured in the Tools Integration Diagram in Figure 29)

- Create an abstract feature model (the initial Scenic program) based on the model of the environment and the system model (the simulated one).
- Use that Scenic program to generate test cases.
- For each test case, run a 30 second simulation using the X-Plane simulator, which is set up using the environmental model and the system model.
- From each simulation run, dump full trace of execution of the simulation and run the falsification tool to check if the specified MTL property is satisfied by the trace or not.
- Take all the traces that have not satisfied the property and perform counterexample analysis on them to identify reasons why each of those counterexamples happened. A few reasons, called failure scenarios, are identified and usually they account for most of the counterexamples.
- Consider the identified failure scenarios and try to refine the initial Scenic program to generate a new test case set that will focus more on those failure scenarios. This is a way to generate a training set for the LEC to better handle these failure scenarios.
- Run the simulations again for each new test case and perform falsification of each trace.
- Create a training data set that focuses more on the failure scenarios from the new test cases, simulation data, and falsification results. Retrain the LEC with this data set and then do another testing cycle of the latest LEC.

VerifAI Tool Specification (CP 1.1)

We captured the tool usage constraints in the form of assumptions and guarantees in the tool specification DSL. For creating the Scenic program using the VerifAI-Scenic tool (Figure 30), which in this case represents an activity, it's important that the environment model correctly represents the operational environment of the system considered in this application. The activity should guarantee that the resulting Scenic program is consistent with the system model and that the resulting Scenic program itself correctly represents the operational environment. Similar tool specification for other tools from Figure 29 are shown in Figure 31 – Figure 36.

```
tool "VerifAI-Scenic" {
	description "Abstract feature space modelling"
	inputs [ sysModel : uasDacEvidence.EvidenceType.system_model , envModel :
	uasDacEvidence.EvidenceType.model ]
	assumptions [
	{ sysModel } + " is correct" ,
	{ envModel } + " correctly represents the operational environment"]
	outputs [ scenicProgram : uasDacEvidence.EvidenceType.abstract_feature_model ]
	guarantees [
	{ scenicProgram } + " is consistent with the system model" ,
	{ scenicProgram } + " correctly represents the operational environment"
	]
	toolset TS1
}
```

Figure 30. VerifAI-Scenic Tool Specification

```
tool "VerifAI-TestGenerator" {
	description "Test case generation"
	inputs [ scenicProgram : uasDacEvidence.EvidenceType.abstract_feature_model ]
	assumptions[
	{scenicProgram}+ " is consistent with the system model",
	{ scenicProgram } + " correctly represents the operational environment"
	]
	outputs [ testSet : uasDacEvidence.EvidenceType.test_cases_specification ]
	guarantees [
	"Distribution of test cases in " + { testSet } + " is consistent with the
	feature distribution in "+{scenicProgram},
	{testSet}+" size conforms to the required test size for the given
	application",
	"Each test case from " +{testSet}+ " is consistent with the system model",
	{testSet}+ " correctly represents operational environment"]
	toolset TS1
}
```

Figure 31. VerifAI Test Generation Tool Specification

```
tool "VerifAI-X-PLANE" {
	description "X-Plane simulator"
	inputs [ testSet : uasDacEvidence.EvidenceType.test_cases_specification ,
	sysModel : uasDacEvidence.EvidenceType.system_model , envModel :
	uasDacEvidence.EvidenceType.model ]
	assumptions[
	"Each test case from " +{testSet}+ " is consistent with "+{sysModel},
	{envModel}+" correctly represents the operational environment",
	"Internal system model is consistent with "+{sysModel}+" and "+ {envModel}]
	outputs [ simResults : uasDacEvidence.EvidenceType.simulation_results ]
	guarantees[
	"A simulation is run for each test case from "+{testSet},
	"Each simulation run is correctly represented by a simulation trace in "
	+{simResults}]
	toolset TS1
}
```

Figure 32. VerifAI X-Plane Tool Specification

```
tool "VerifAI-MTLMonitor" {
      description "Temporal logic falsification"
      inputs [ simResults : uasDacEvidence.EvidenceType.simulation_results ,
      formalSpec : uasDacEvidence.EvidenceType.formal_specification ]
      assumptions [
      {formalSpec} +" variables are present in "+{simResults}]
      outputs [falsRes: uasDacEvidence.EvidenceType.
      temporal_logic_falsification_results ]
      guarantees [
      "Each trace from "+{simResults}+" is evaluated against "+{formalSpec}+" in "
      +{falsRes}]
      toolset TS1
}
```

Figure 33. VerifAI MTL Monitor Tool Specification

```
tool "VerifAI-CA" {
      description "Counterexample analysis"
      inputs [ falsRes : uasDacEvidence.EvidenceType.
      temporal_logic_falsification_results ]
      assumptions [{falsRes}+" contains only traces violating the formal property"]
      outputs [caRes : uasDacEvidence.EvidenceType.counterexample_analysis_results ]
      guarantees [
      {caRes}+" identified a failure scenario for each counterexample from "
      +{falsRes}]

      toolset TS1
}
```

Figure 34. VerifAI Counterexample Analysis Tool Specification

```
tool "VerifAI-ScenicRefinement" {
      description "Abstract feature space refinement"
      inputs [ inputScenic : uasDacEvidence.EvidenceType.abstract_feature_model ,
      caResults : uasDacEvidence.EvidenceType.counterexample_analysis_results ]
      assumptions[
      {inputScenic}+" correctly represents the environment model",
      {inputScenic}+" correctly represents system model"
      ]
      outputs [ outputScenic : uasDacEvidence.EvidenceType.abstract_feature_model ]
      guarantees[
      {outputScenic}+" correctly represents the environment model",
      {outputScenic}+" correctly represents system model"
      ]
      toolset TS1
}
```

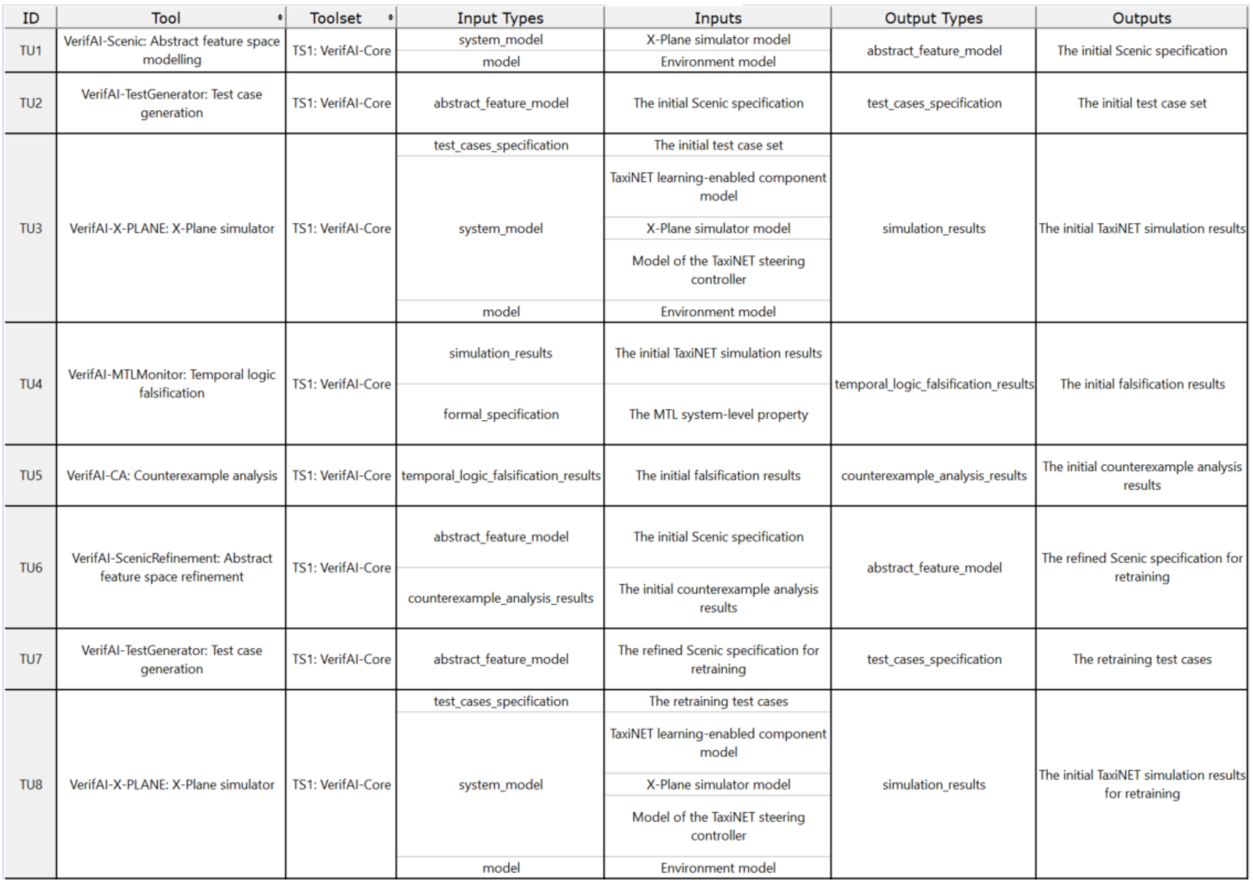
Figure 35. VerifAI Scenic Refinement Tool Specification

```
tool "VerifAI-TrainingDataGen" {
      description "LEC training data set generator"
      inputs [simResult : uasDacEvidence.EvidenceType.simulation_results , falsfRes:
      uasDacEvidence.EvidenceType.temporal_logic_falsification_results , testSet :
      uasDacEvidence.EvidenceType.test_cases_specification ]
      assumptions [
        "Each simulation trace in "+{simResult}+" corresponds to one test case in "
      +{testSet}+"and one simulation trace in "+{falsfRes},
        {testSet}+" correctly represents the operational environment",
      "The size of the input sets "+{simResult}+{testSet}+{falsfRes}+" meets the
      quantitative targets for the training set size"]
      outputs [ trainingSet : uasDacEvidence.EvidenceType.training_data_set ]
      guarantees [
      "Each image in "+{trainingSet}+" is representative of the simulation run from
      which it is extracted",
      {trainingSet}+" correctly represents the operational environment"]
      toolset TS1
}
```

Figure 36. VerifAI Training Data Generation Tool Specification

VerifAI Tool Use Specification (CP 1.1)

We capture three iterations of VerifAI falsification for the sake of LEC testing and retraining. The first iteration deals with testing the input LEC. It starts by making the Scenic program, generating the test cases, performing simulation and falsification of its results, then performing counterexample analysis of the falsification results. We then capture the refinement of the Scenic program based on the counterexample analysis results with the purpose to create specialized Scenic programs that will be used for generating new training data in order to improve the performance of the LEC in the areas where it does not perform well. The captured tool uses of the first iteration are shown in Figure 37. Similar tool uses are captured for the other iterations, where the training data is generated and LEC retrained, and finally where this retrained LEC is tested in another iteration.



ID	Tool	Toolset	Input Types	Inputs	Output Types	Outputs
TU1	VerifAI-Scenic: Abstract feature space modelling	TS1: VerifAI-Core	system_model	X-Plane simulator model	abstract_feature_model	The initial Scenic specification
			model	Environment model		
TU2	VerifAI-TestGenerator: Test case generation	TS1: VerifAI-Core	abstract_feature_model	The initial Scenic specification	test_cases_specification	The initial test case set
TU3	VerifAI-X-PLANE: X-Plane simulator	TS1: VerifAI-Core	test_cases_specification	The initial test case set	simulation_results	The initial TaxiNET simulation results
			system_model	TaxiNET learning-enabled component model		
				X-Plane simulator model		
				Model of the TaxiNET steering controller		
			model	Environment model		
TU4	VerifAI-MTLMonitor: Temporal logic falsification	TS1: VerifAI-Core	simulation_results	The initial TaxiNET simulation results	temporal_logic_falsification_results	The initial falsification results
			formal_specification	The MTL system-level property		
TU5	VerifAI-CA: Counterexample analysis	TS1: VerifAI-Core	temporal_logic_falsification_results	The initial falsification results	counterexample_analysis_results	The initial counterexample analysis results
TU6	VerifAI-ScenicRefinement: Abstract feature space refinement	TS1: VerifAI-Core	abstract_feature_model	The initial Scenic specification	abstract_feature_model	The refined Scenic specification for retraining
			counterexample_analysis_results	The initial counterexample analysis results		
TU7	VerifAI-TestGenerator: Test case generation	TS1: VerifAI-Core	abstract_feature_model	The refined Scenic specification for retraining	test_cases_specification	The retraining test cases
TU8	VerifAI-X-PLANE: X-Plane simulator	TS1: VerifAI-Core	test_cases_specification	The retraining test cases	simulation_results	The initial TaxiNET simulation results for retraining
			system_model	TaxiNET learning-enabled component model		
				X-Plane simulator model		
				Model of the TaxiNET steering controller		
			model	Environment model		

Figure 37. Excerpt of Tool Use Specification for VerifAI Retraining Application

VerifAI Evidence Log (CP1.1)

The input and output evidence captured in the Tool uses are recorded in the Evidence log in more detail. An excerpt from the Evidence log is shown in Figure 38. The Evidence definitions not only include information about the evidence item, itself, but can also include relations to other Evidence items. In that way we capture some of the required properties based on the usage constraints directly in the Evidence Log. For example, we can indicate that the SCENICFALSIF program is consistent with the XPlane-model, which means that we will be able to use scenes generated from that program in XPlane.

```
artifact scenicFalsif {
        description "The initial Scenic specification"
        purpose "The initial Scenic specification of the environment for TaxiNET falsification.
        Specifies generic scenarios of TaxiNET environment in terms of abstract features to
        guide TaxiNET falsification"
        type abstract_feature_model
        provenance ""
        status obtained_and_to_be_verified
        generatedBy VerifAI-Scenic from environmentModel and XPlane-model
        correctlyRepresents environmentModel
        isConsistentWith XPlane-model
}
artifact scenicFailure {
        description "The refined Scenic specification focused on failure scenarios"
        purpose "The refined Scenic specification focused on the failure scenarios used for
        runtime monitor training."
        type abstract_feature_model
        requires retrainedFailureConstraints
        generatedBy VerifAI-ScenicRefinement from retrainedCounterexampleAnalysisResults and
        scenicRetrainOR
        isConsistentWith XPlane-model
}
artifact TaxiNET-steeringControllerModel {
        description "Model of the TaxiNET steering controller"
        purpose "Steering controller that issues the steering command based on the input from
        TaxiNET-LEC"
        type system_model
        status obtained_and_to_be_verified
}
artifact TaxiNET-LEC-Model {
        description "TaxiNET learning-enabled component model"
        purpose "Model of the TaxiNET neural network that estimates the heading error and
        cross-track error of the plane based on the camera input"
        type system_model
        status obtained_and_to_be_verified
}
artifact TaxiNET-LEC-Model-Retrained {
        description "The retrained TaxiNET learning-enabled component model"
        purpose "The improved model of the TaxiNET neural network that estimates the heading
        error and cross-track error of the plane based on the camera input"
        type system_model
        isLearnedFrom trainingDataSet, trainingDataSetWOverrepresentation
}
artifact XPlane-model {
        description "X-Plane simulator model"
        purpose "Models the environment of the plane and its dynamics"
        type system_model
        correctlyRepresents environmentModel
}
artifact environmentModel {
        description "Environment model"
        purpose "Model of the TaxiNET environment"
        type model
        correctlyRepresents TaxiNET-OperationalEnvironment
}
```

Figure 38. Excerpt of Evidence Log from VerifAI Retraining

VerifAI CP1.1 Specific Argument

We connect the verification of the TaxiNet centerline tracking to the steering malfunctions hazard, as depicted in Figure 39. In particular, we focus on presenting the results of the verification of the centerline tracking requirement using VerifAI (Figure 40). We then decompose the VerifAI application based on the different VerifAI functions, focusing on Scenic modelling (Figure 41), test scene generation (Figure 42), temporal logic falsification (Figure 43), and failure detection techniques (Figure 44). We further populate each of these based on the tool specification argument pattern instances, which we further enrich and decompose manually.

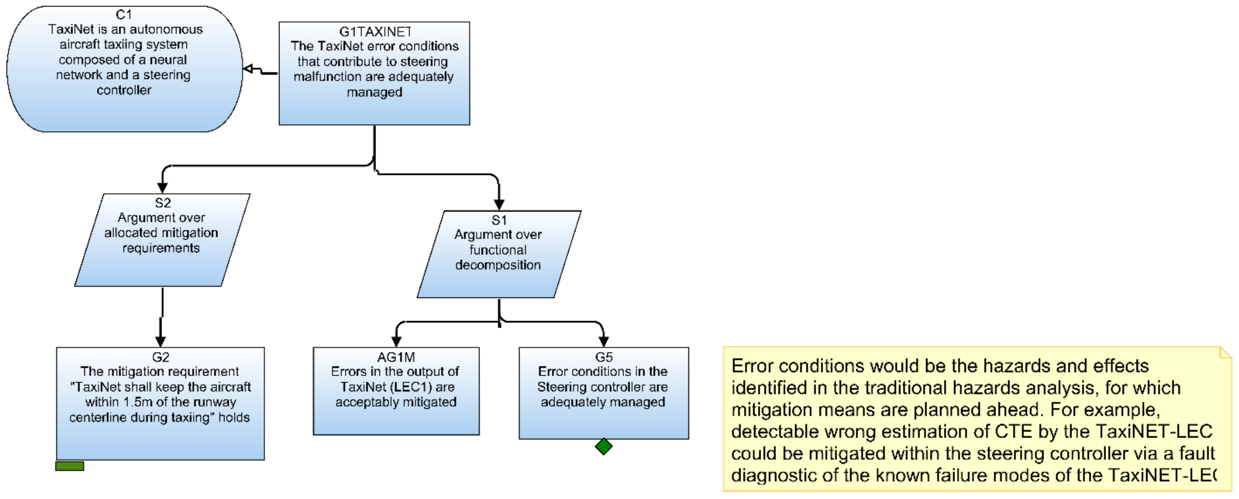

C1
TaxiNet is an autonomous aircraft taxiing system composed of a neural network and a steering controller
G1TAXINET
The TaxiNet error conditions that contribute to steering malfunction are adequately managed
S2
Argument over allocated mitigation requirements
S1
Argument over functional decomposition
G2
The mitigation requirement "TaxiNet shall keep the aircraft within 1.5m of the runway centerline during taxiing" holds
AG1M
Errors in the output of TaxiNet (LEC1) are acceptably mitigated
G5
Error conditions in the Steering controller are adequately managed
Error conditions would be the hazards and effects identified in the traditional hazards analysis, for which mitigation means are planned ahead. For example, detectable wrong estimation of CTE by the TaxiNET-LEC could be mitigated within the steering controller via a fault diagnostic of the known failure modes of the TaxiNET-LE(

Figure 39. Integration of Lower-Level Verification with Higher-Level Argument Claims

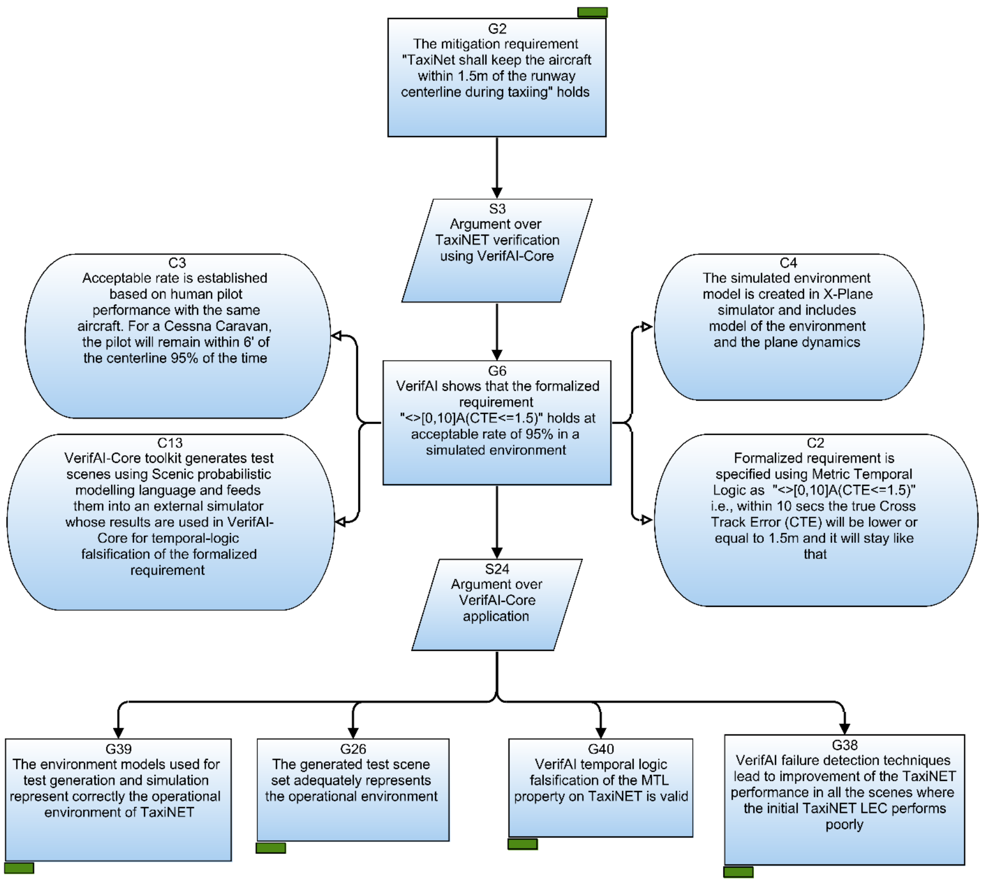

G2
The mitigation requirement "TaxiNet shall keep the aircraft within 1.5m of the runway centerline during taxiing" holds
S3
Argument over TaxiNET verification using VerifAI-Core
C3
Acceptable rate is established based on human pilot performance with the same aircraft. For a Cessna Caravan, the pilot will remain within 6' of the centerline 95% of the time
C4
The simulated environment model is created in X-Plane simulator and includes model of the environment and the plane dynamics
G6
VerifAI shows that the formalized requirement "<>[0,10]A(CTE<=1.5)" holds at acceptable rate of 95% in a simulated environment
C13
VerifAI-Core toolkit generates test scenes using Scenic probabilistic modelling language and feeds them into an external simulator whose results are used in VerifAI-Core for temporal-logic falsification of the formalized requirement
C2
Formalized requirement is specified using Metric Temporal Logic as "<>[0,10]A(CTE<=1.5)" i.e., within 10 secs the true Cross Track Error (CTE) will be lower or equal to 1.5m and it will stay like that
S24
Argument over VerifAI-Core application
G39
The environment models used for test generation and simulation represent correctly the operational environment of TaxiNET
G26
The generated test scene set adequately represents the operational environment
G40
VerifAI temporal logic falsification of the MTL property on TaxiNET is valid
G38
VerifAI failure detection techniques lead to improvement of the TaxiNET performance in all the scenes where the initial TaxiNET LEC performs poorly

Figure 40. Argument Fragment for Centerline Tracking Mitigation Requirement

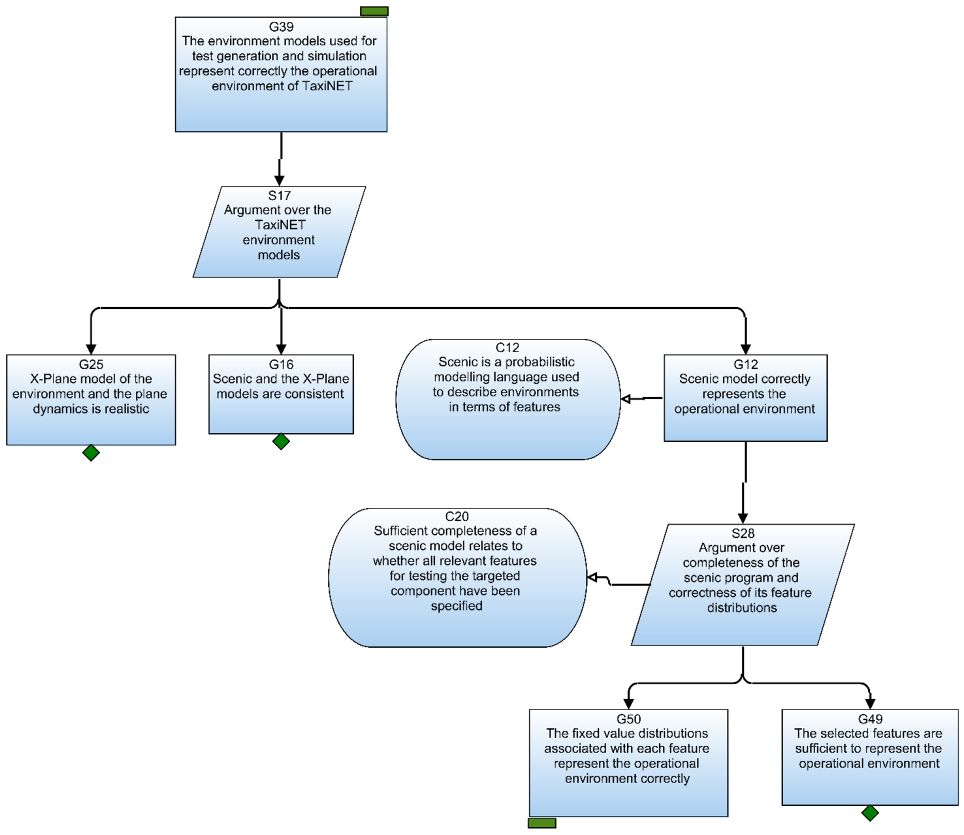

G39
The environment models used for test generation and simulation represent correctly the operational environment of TaxiNET
S17
Argument over the TaxiNET environment models
G25
X-Plane model of the environment and the plane dynamics is realistic
G16
Scenic and the X-Plane models are consistent
C12
Scenic is a probabilistic modelling language used to describe environments in terms of features
G12
Scenic model correctly represents the operational environment
C20
Sufficient completeness of a scenic model relates to whether all relevant features for testing the targeted component have been specified
S28
Argument over completeness of the scenic program and correctness of its feature distributions
G50
The fixed value distributions associated with each feature represent the operational environment correctly
G49
The selected features are sufficient to represent the operational environment

Figure 41. Argument Fragment for Constraints on Environment Models

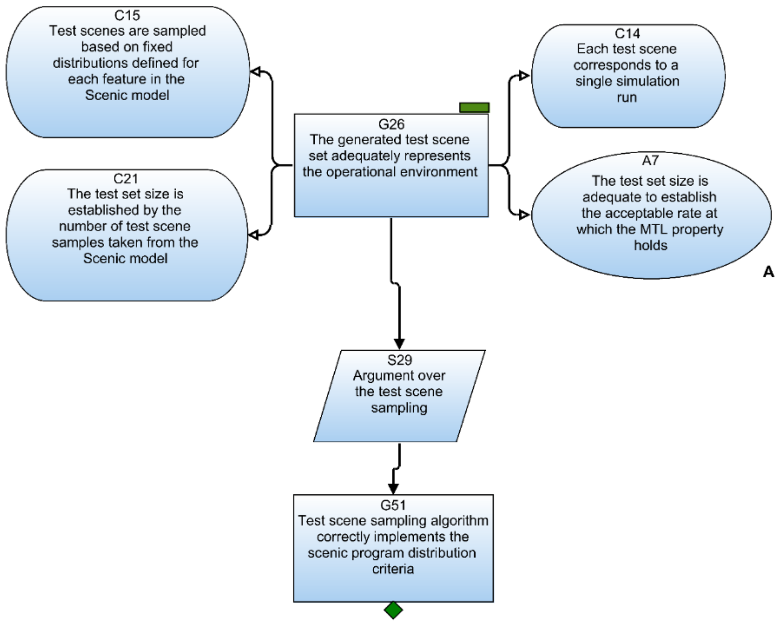

C15
Test scenes are sampled based on fixed distributions defined for each feature in the Scenic model
C21
The test set size is established by the number of test scene samples taken from the Scenic model
G26
The generated test scene set adequately represents the operational environment
C14
Each test scene corresponds to a single simulation run
A7
The test set size is adequate to establish the acceptable rate at which the MTL property holds
A
S29
Argument over the test scene sampling
G51
Test scene sampling algorithm correctly implements the scenic program distribution criteria

Figure 42. Argument Fragment for Test Scene Generation

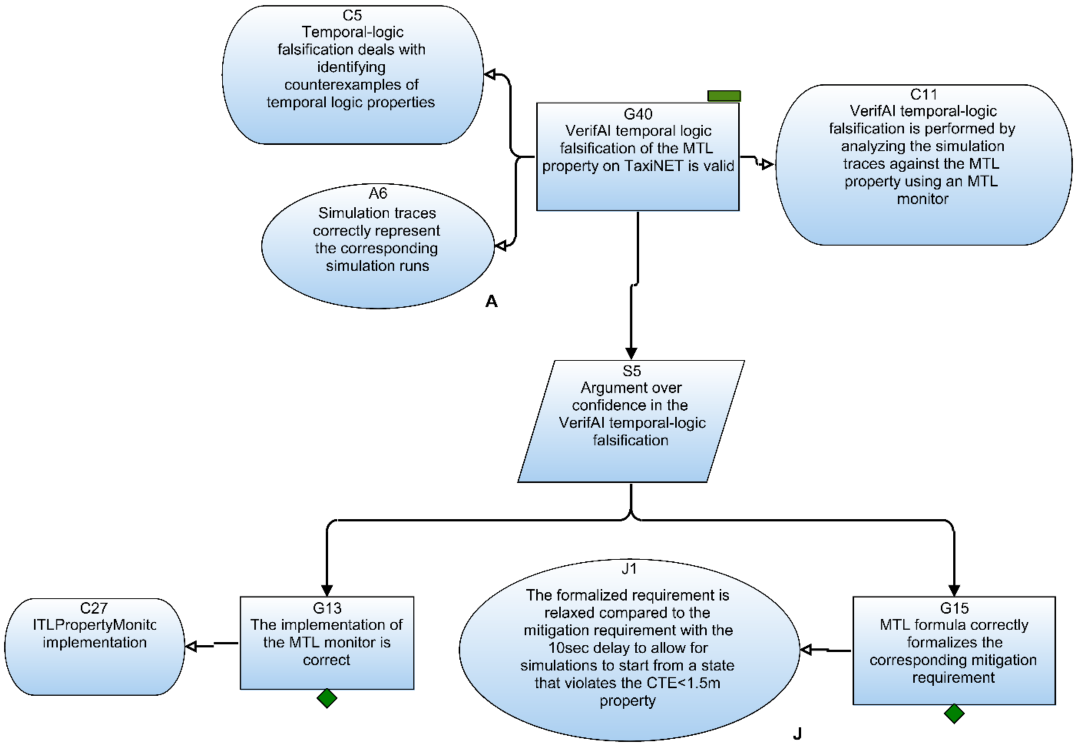

C5
Temporal-logic falsification deals with identifying counterexamples of temporal logic properties
G40
VerifAI temporal logic falsification of the MTL property on TaxiNET is valid
C11
VerifAI temporal-logic falsification is performed by analyzing the simulation traces against the MTL property using an MTL monitor
A6
Simulation traces correctly represent the corresponding simulation runs
A
S5
Argument over confidence in the VerifAI temporal-logic falsification
C27
ITLPropertyMonitc implementation
G13
The implementation of the MTL monitor is correct
J1
The formalized requirement is relaxed compared to the mitigation requirement with the 10sec delay to allow for simulations to start from a state that violates the CTE<1.5m property
J
G15
MTL formula correctly formalizes the corresponding mitigation requirement

Figure 43. Argument Fragment for Temporal Logic Falsification

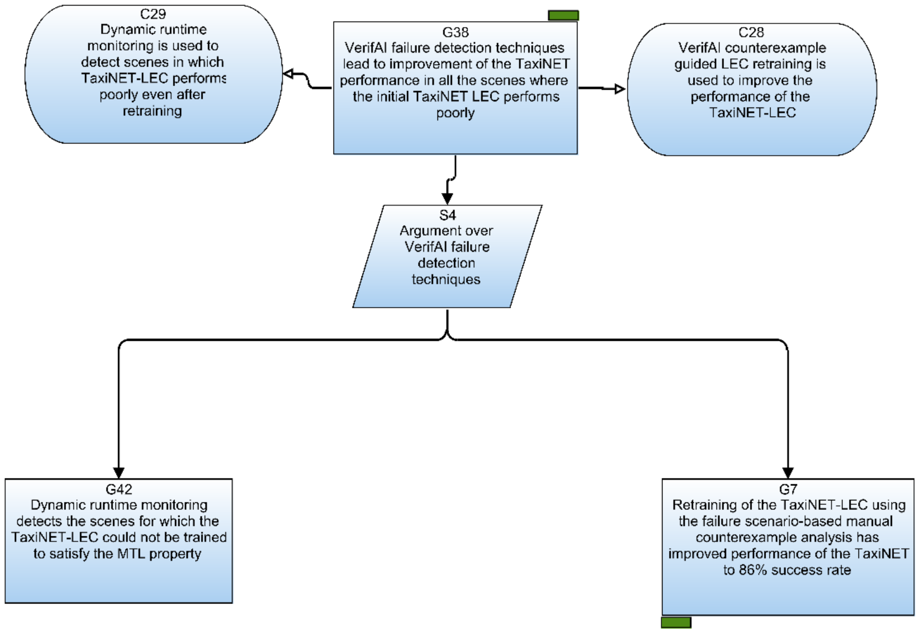

C29
Dynamic runtime monitoring is used to detect scenes in which TaxiNET-LEC performs poorly even after retraining
G38
VerifAI failure detection techniques lead to improvement of the TaxiNET performance in all the scenes where the initial TaxiNET LEC performs poorly
C28
VerifAI counterexample guided LEC retraining is used to improve the performance of the TaxiNET-LEC
S4
Argument over VerifAI failure detection techniques
G42
Dynamic runtime monitoring detects the scenes for which the TaxiNET-LEC could not be trained to satisfy the MTL property
G7
Retraining of the TaxiNET-LEC using the failure scenario-based manual counterexample analysis has improved performance of the TaxiNET to 86% success rate

Figure 44. Argument Fragment for VerifAI Failure Detection Techniques

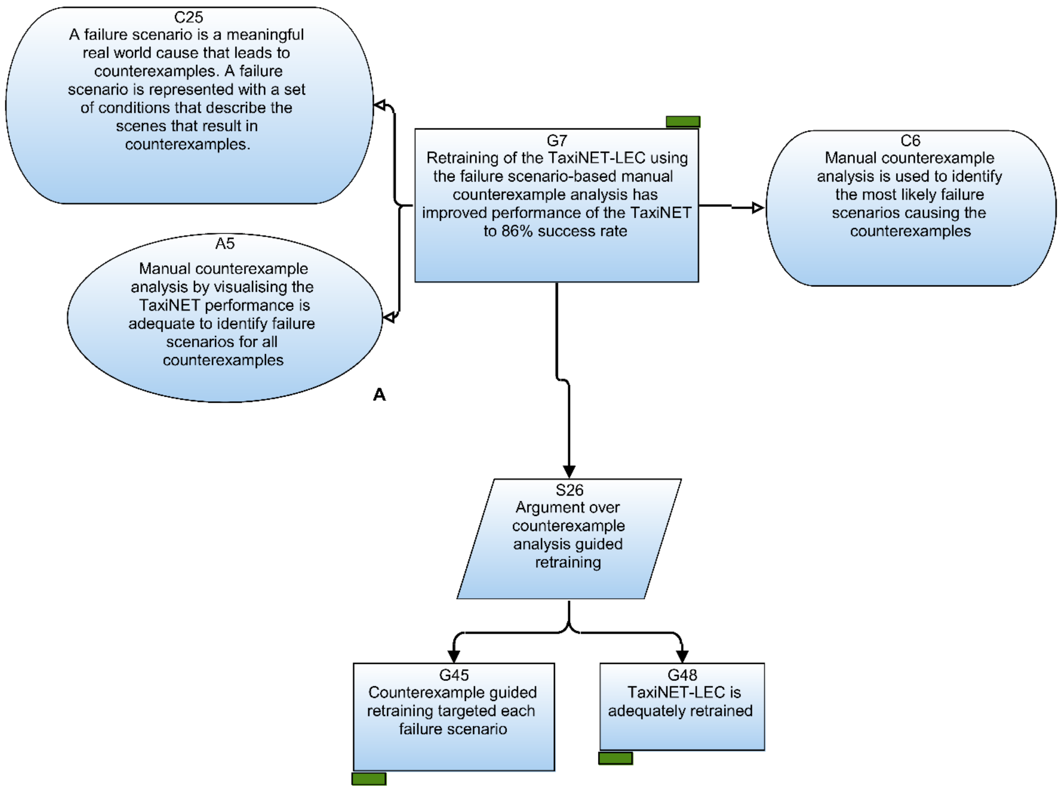

C25
A failure scenario is a meaningful real world cause that leads to counterexamples. A failure scenario is represented with a set of conditions that describe the scenes that result in counterexamples.
A5
Manual counterexample analysis by visualising the TaxiNET performance is adequate to identify failure scenarios for all counterexamples
A
G7
Retraining of the TaxiNET-LEC using the failure scenario-based manual counterexample analysis has improved performance of the TaxiNET to 86% success rate
C6
Manual counterexample analysis is used to identify the most likely failure scenarios causing the counterexamples
S26
Argument over counterexample analysis guided retraining
G45
Counterexample guided retraining targeted each failure scenario
G48
TaxiNET-LEC is adequately retrained

Figure 45. Argument Fragment on Results of VerifAI Retraining of LEC

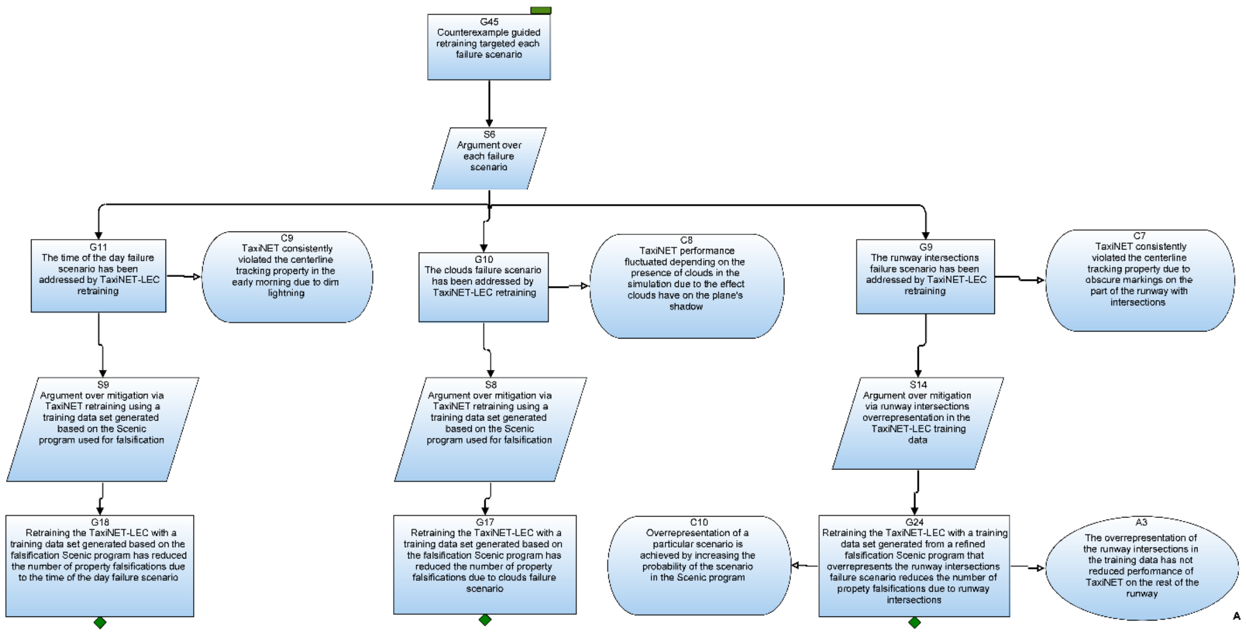

G45
Counterexample guided retraining targeted each failure scenario
S6
Argument over each failure scenario
G11
The time of the day failure scenario has been addressed by TaxiNET-LEC retraining
C9
TaxiNET consistently violated the centerline tracking property in the early morning due to dim lightning
G10
The clouds failure scenario has been addressed by TaxiNET-LEC retraining
C8
TaxiNET performance fluctuated depending on the presence of clouds in the simulation due to the effect clouds have on the plane's shadow
G9
The runway intersections failure scenario has been addressed by TaxiNET-LEC retraining
C7
TaxiNET consistently violated the centerline tracking property due to obscure markings on the part of the runway with intersections
S9
Argument over mitigation via TaxiNET retraining using a training data set generated based on the Scenic program used for falsification
S8
Argument over mitigation via TaxiNET retraining using a training data set generated based on the Scenic program used for falsification
S14
Argument over mitigation via runway intersections overrepresentation in the TaxiNET-LEC training data
G18
Retraining the TaxiNET-LEC with a training data set generated based on the falsification Scenic program has reduced the number of property falsifications due to the time of the day failure scenario
G17
Retraining the TaxiNET-LEC with a training data set generated based on the falsification Scenic program has reduced the number of property falsifications due to clouds failure scenario
C10
Overrepresentation of a particular scenario is achieved by increasing the probability of the scenario in the Scenic program
G24
Retraining the TaxiNET-LEC with a training data set generated from a refined falsification Scenic program that overrepresents the runway intersections failure scenario reduces the number of propety falsifications due to runway intersections
A3
The overrepresentation of the runway intersections in the training data has not reduced perfomance of TaxiNET on the rest of the runway
A

Figure 46. Argument Fragment for Addressing Identified Failure Scenarios

VerifAI Application for Runtime Assurance on TaxiNet (CP 2.1)

The main requirement to be addressed is "Dynamic runtime monitoring shall detect the scenes for which the TaxiNet LEC could not be trained". This relates to the training performed with VerifAI in CP1.1. The quantitative target in this case is the rate at which a monitoring component returns false negative results, i.e., when the component indicates there is no problem when there in fact is a problem. This should not be happening.

The VerifAI Runtime Monitoring workflow [VAI2021] in this case builds upon the workflow from CP1.1 and uses the information from there as inputs (Figure 47):

- The traces from falsification results are fragmented into 10 step windows such that there is an input window, the response time window, and the decision window. The technique works by taking the input window variable trace and looking for x steps into the future (the response time window length) to see if in that future window (the decision window) the property is met or not. One string of learning data is the variable trace from the input window and the yes/no property verification from the decision window.
- A decision tree is learned from such data to say whether you will be safe or unsafe in x steps in the future for a particular input window. However, for the same trace of variables in the input window, sometimes you can have a safe situation in the decision window and sometimes you can have unsafe situation. In such cases, the algorithm simply takes the majority. For example, for trace x there are two safe and one unsafe, the decision tree synthesizer will consider that this trace will yield a safe outcome. That is encoded in the decision tree. This assumption is captured in the argument.
- From such a learned decision tree, we generate a runtime monitor module that tells if you will be in a safe or unsafe situation given the current status of the variables.

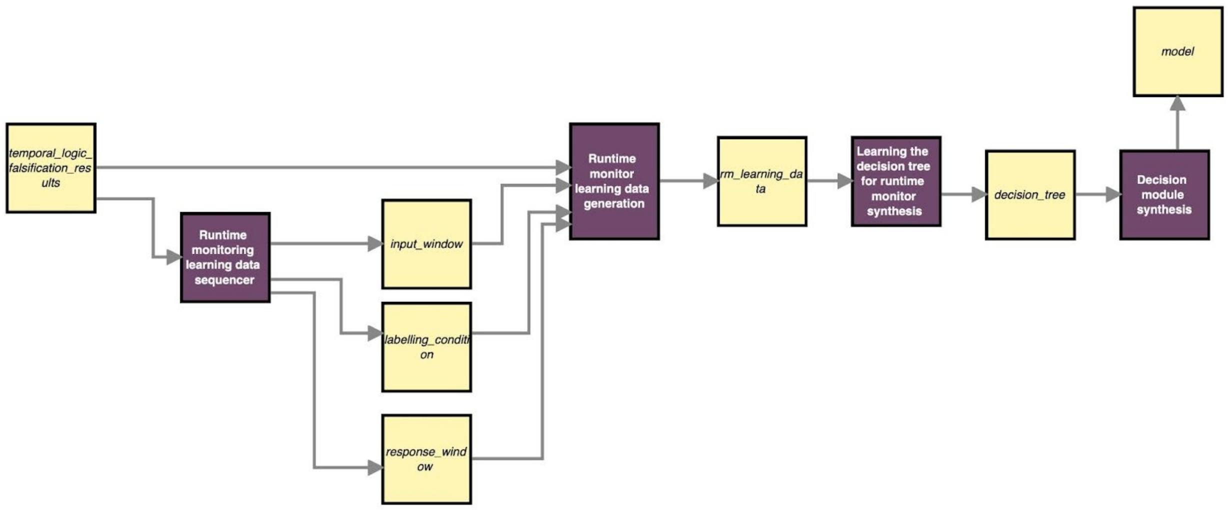

model
temporal_logic_falsification_results
Runtime monitoring learning data sequencer
input_window
labelling_condition
response_window
Runtime monitor learning data generation
rm_learning_data
Learning the decision tree for runtime monitor synthesis
decision_tree
Decision module synthesis

Figure 47. VerifAI Runtime Monitor Learning Workflow

VerifAI Tool Specification in CP2.1

The tool specification for each tool from Figure 47 is shown in Figure 48. Besides the tool inputs and outputs, we also defined the tool assumptions and guarantees. Given that the Runtime Monitor aims at catching all the failure scenarios for which the LEC does not perform adequately, we assume that the input data, which will be used for the runtime monitor learning, covers all the failure scenarios. The DATASEQUENCER activity needs to guarantee certain properties about the values that are used for sequencing the input data.

In particular, that the input window length is adequately determined, and that the response window accounts for the safe controller response time analysis. When generating the learning data from the raw input data, we need to make sure that the labelling condition correctly represents the CTE-violation property (the one for which the LEC was trained for). An important guarantee of the learning data generation activity is that the variables that are selected to be included in the learning data are monitorable in real time. Otherwise, the generated runtime monitor will not be feasible.

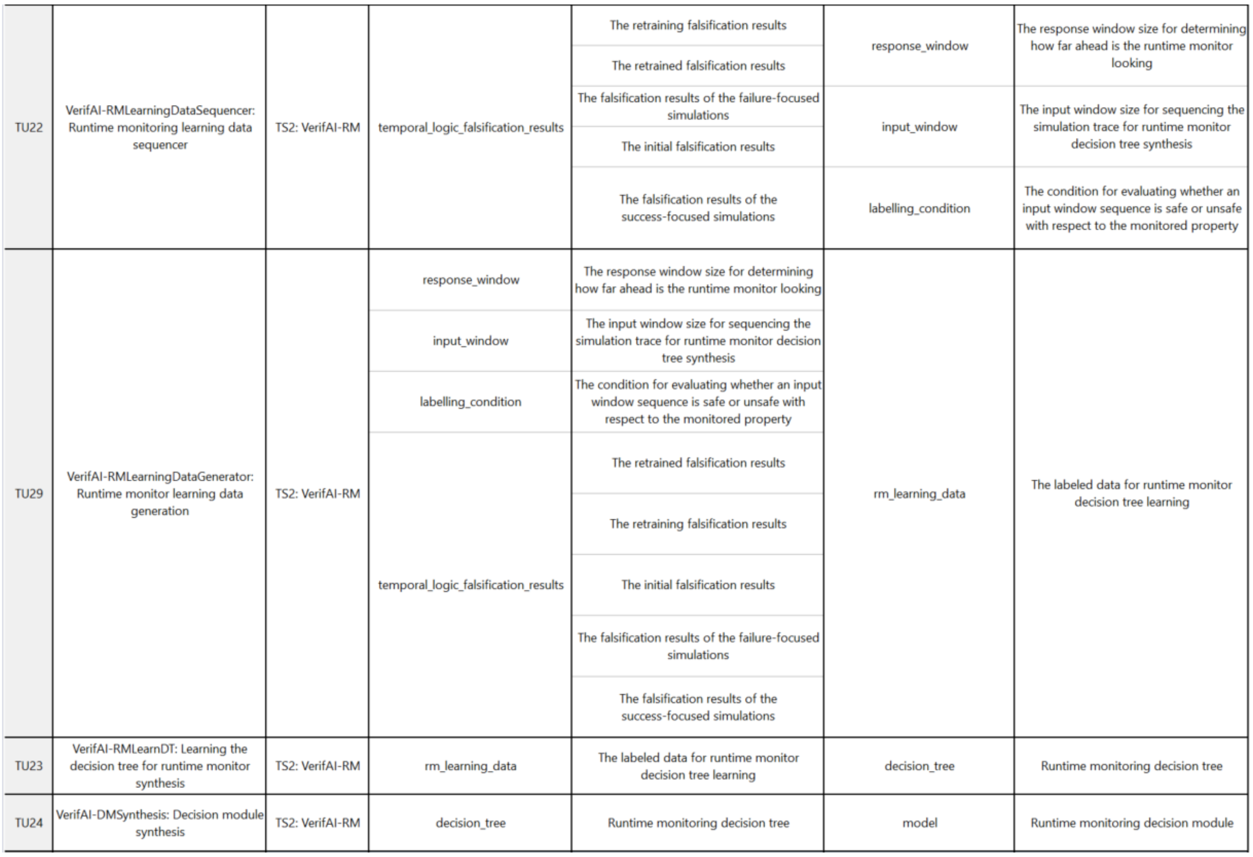
When learning the decision tree from the labelled data, we emphasize that special attention needs to be given to the decision tree learning condition and that it needs to be aligned with the risk targets associated with the potential consequences of wrong decision by the runtime monitor. For example, the learning data can contain multiple input windows with different labels. The same variable values that we are monitoring can sometimes lead to the property violations and sometimes not. Just having the decision condition that takes the majority may not always be acceptable from the risk perspective. Finally, we capture that the resulting monitor should guarantee that it complements the LEC and that it can detect all failure scenarios for which the LEC could not be trained.

VerifAI Tool Use Specification in CP2.1

An excerpt from the captured Tool Use of VerifAI Runtime Monitor learning is shown in Figure 49. The runtime monitoring was performed using the falsification results from different testing iterations of the LEC.

```
tool "VerifAI-RMLearningDataSequencer" {
        description "Runtime monitoring learning data sequencer"
        inputs [ rawData :
        uasDacEvidence.EvidenceType.temporal_logic_falsification_results ]
        assumptions [
        { rawData } + " covers all failure scenarios" ,
        {rawData}+" correctly represents the operational environment"]
        outputs [ resWin : uasDacEvidence.EvidenceType.response_window , inWin :
        uasDacEvidence.EvidenceType.input_window , lCond :
        uasDacEvidence.EvidenceType.labelling_condition ]
        guarantees [
        { resWin } + " accounts for the actual time the safe controller needs to
        adequately respond to a possible failure." ,
        "The input window length of "+{inWin}+" has been adequately determined" ]
        toolset TS2
}

tool "VerifAI-RMLearnDT" {
        description "Learning the decision tree for runtime monitor synthesis"
        inputs [ in1 : uasDacEvidence.EvidenceType.rm_learning_data ]
        outputs [ dTree : uasDacEvidence.EvidenceType.decision_tree ]
        guarantees [
        "The decision tree synthesizer condition used to build" +{dTree} +"meets the
        risk targets associated with the potential consequence of the resulting
        failure scenario"]
        toolset TS2
}

tool "VerifAI-DMSynthesis" {
        description "Decision module synthesis"
        inputs [ dTree : uasDacEvidence.EvidenceType.decision_tree ]
        assumptions [
        {dTree}+" covers all failure scenarios",
        {dTree}+" never marks an unsafe input window as safe"
    ]
        outputs [ monitor : uasDacEvidence.EvidenceType.model ]
        guarantees [
        "The synthesized runtime monitor"+{monitor}+" detects all scenarios for which
        the LEC could not be trained"]
        toolset TS2
}

tool "VerifAI-RMLearningDataGenerator" {
        description "Runtime monitor learning data generation"
        inputs [ resWin : uasDacEvidence.EvidenceType.response_window , inWin :
        uasDacEvidence.EvidenceType.input_window , lCond :
        uasDacEvidence.EvidenceType.labelling_condition , falsRes :
        uasDacEvidence.EvidenceType.temporal_logic_falsification_results ]
        assumptions [
        "The labeling condition "+{lCond}+" correctly represents the CTE-Violation
        property" ]
        outputs [ lData : uasDacEvidence.EvidenceType.rm_learning_data ]
        guarantees [
        "The selected input variables for decision tree learning in "+{lData} + "can
        be obtained with sufficient correctness in the operation time"]
```

Figure 48. VerifAI Runtime Monitoring Tool Specification



TU22	VerifAI-RMLearningDataSequencer: Runtime monitoring learning data sequencer	TS2: VerifAI-RM	temporal_logic_falsification_results	The retraining falsification results	response_window	The response window size for determining how far ahead is the runtime monitor looking
				The retrained falsification results		
				The falsification results of the failure-focused simulations	input_window	The input window size for sequencing the simulation trace for runtime monitor decision tree synthesis
				The initial falsification results		
				The falsification results of the success-focused simulations	labelling_condition	The condition for evaluating whether an input window sequence is safe or unsafe with respect to the monitored property
TU29	VerifAI-RMLearningDataGenerator: Runtime monitor learning data generation	TS2: VerifAI-RM	response_window	The response window size for determining how far ahead is the runtime monitor looking	rm_learning_data	The labeled data for runtime monitor decision tree learning
			input_window	The input window size for sequencing the simulation trace for runtime monitor decision tree synthesis		
			labelling_condition	The condition for evaluating whether an input window sequence is safe or unsafe with respect to the monitored property		
			temporal_logic_falsification_results	The retrained falsification results		
				The retraining falsification results		
				The initial falsification results		
				The falsification results of the failure-focused simulations		
				The falsification results of the success-focused simulations		
TU23	VerifAI-RMLearnDT: Learning the decision tree for runtime monitor synthesis	TS2: VerifAI-RM	rm_learning_data	The labeled data for runtime monitor decision tree learning	decision_tree	Runtime monitoring decision tree
TU24	VerifAI-DMSynthesis: Decision module synthesis	TS2: VerifAI-RM	decision_tree	Runtime monitoring decision tree	model	Runtime monitoring decision module

Figure 49. Excerpt of Tool Uses from VerifAI Runtime Monitoring

VerifAI Evidence Log in CP2.1

```
artifact decisionModule {
      description "Runtime monitoring decision module"
      purpose "Runtime monitoring decision module for determining whether a setting
      is trusted or not"
      type model
      generatedBy VerifAI-DMSynthesis from decisionTree
}

artifact decisionTree {
      description "Runtime monitoring decision tree"
      purpose "Decision tree for learning the runtime monitoring decision module"
      type decision_tree
      generatedBy VerifAI-RMLearnDT from EA7
}

artifact EA7 {
      description "The labeled data for runtime monitor decision tree learning"
      type rm_learning_data
      generatedBy VerifAI-RMLearningDataGenerator with RM-responseWindowSize and RM-
      inputWindowSize and RM-labellingCondition and retrainedFalsificationResults
      and retrainingFalsificationResults and initialFalsificationResults and EA5 and
      EA6
}

artifact RM-inputWindowSize {
      description "The input window size for sequencing the simulation trace for
      runtime monitor decision tree synthesis"
      type input_window
      generatedBy VerifAI-RMLearningDataSequencer from
      retrainingFalsificationResults and retrainedFalsificationResults and EA5 and
      initialFalsificationResults and EA6
}

artifact RM-responseWindowSize {
      description "The response window size for determining how far ahead is the
      runtime monitor looking"
      type response_window
      generatedBy VerifAI-RMLearningDataSequencer from
      retrainingFalsificationResults and retrainedFalsificationResults and EA5 and
      initialFalsificationResults and EA6
}

artifact RM-labellingCondition {
      description "The condition for evaluating whether an input window sequence is
      safe or unsafe with respect to the monitored property"
      type labelling_condition
      isConsistentWith MTLProperty
      generatedBy VerifAI-RMLearningDataSequencer from
      retrainingFalsificationResults and retrainedFalsificationResults and EA5 and
      initialFalsificationResults and EA6
}
```

Figure 50. Evidence Log Excerpt for Application of VerifAI Runtime Monitoring

VerifAI CP2.1 Specific Argument

To assure the requirement that the runtime monitor returns no false negative results, we decompose the claim over the VerifAI-RM application (Figure 51). Besides presenting the achieved quantitative targets, we also address the trustworthiness in the VerifAI-RM application. The argument in

Figure 52 indicates that the quantitative target set forth by the requirement is not met by the final results, since there were still 10% false negatives. Considering that this is an interim safety case, we present in the argument the latest result. It indicates that either further learning of the runtime monitor is needed, or that an alternative needs to be found to justify the final false negative rate.

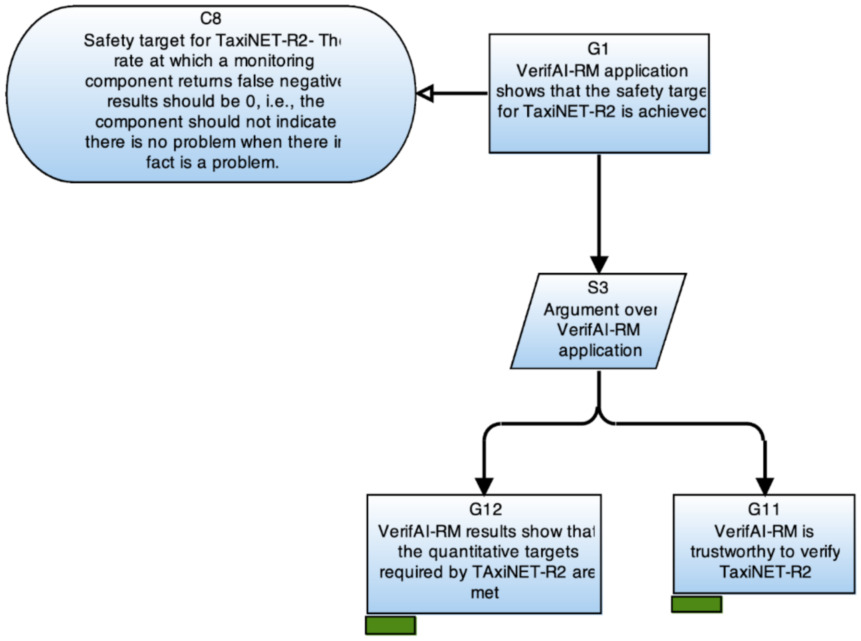

C8
Safety target for TaxiNET-R2- Th
rate at which a monitoring
component returns false negativ
results should be 0, i.e., the
component should not indicate
there is no problem when there i
fact is a problem.
G1
VerifAI-RM application
shows that the safety targe
for TaxiNET-R2 is achieve
S3
Argument ove
VerifAI-RM
application
G12
VerifAI-RM results show tha
the quantitative targets
required by TAxiNET-R2 are
met
G11
VerifAI-RM is
trustworthy to verify
TaxiNET-R2

Figure 51. Top-level Argument of VerifAI-RM

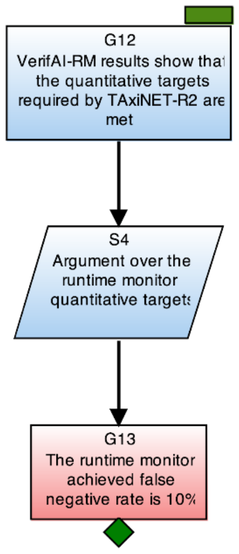


Figure 52. VerifAI Runtime Monitoring Quantitative Targets

We decompose the goal on VerifAI-RM trustworthiness by looking at whether all the usage constraints of the different parts of the workflow are met (Figure 53). Each of the VerifAI-RM tool-specific argument fragments (Figure 55, Figure 56, and Figure 54) are generated by instantiating a tool-specific pattern from the tool use information. Then, we have manually extended the result, and left the goals undeveloped to indicate that each of the usage constraints needs to be separately assured.

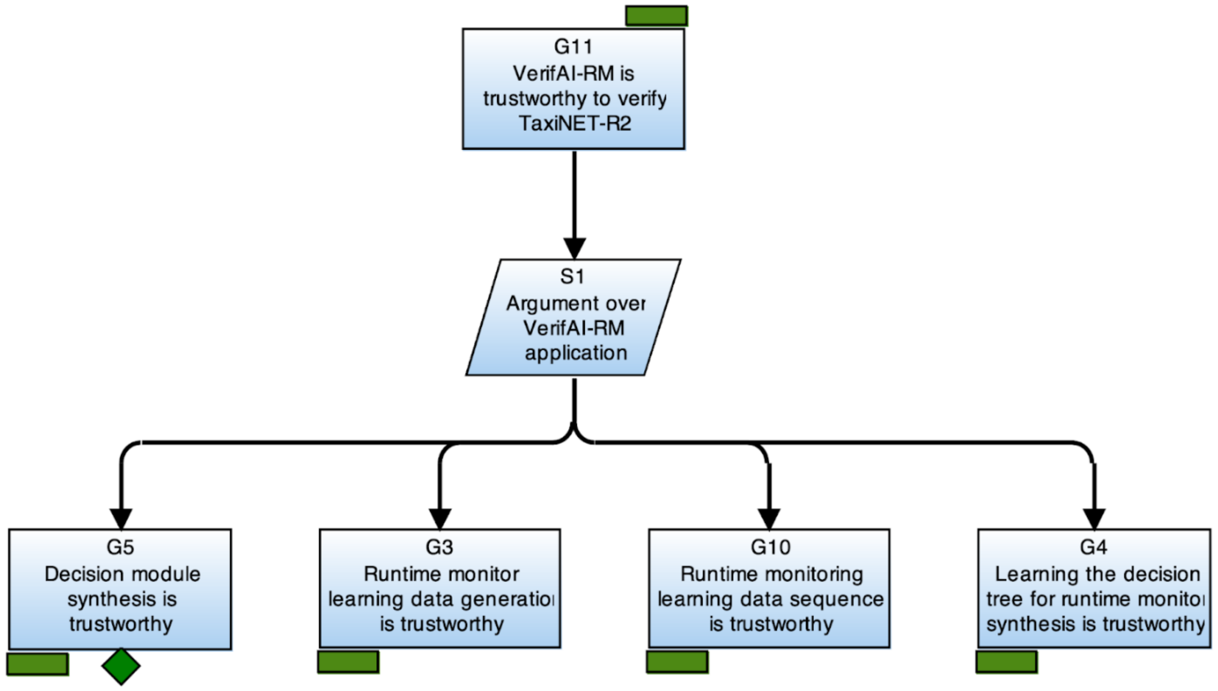

G11
VerifAI-RM is trustworthy to verify TaxiNET-R2
S1
Argument over VerifAI-RM application
G5
Decision module synthesis is trustworthy
G3
Runtime monitor learning data generation is trustworthy
G10
Runtime monitoring learning data sequence is trustworthy
G4
Learning the decision tree for runtime monitor synthesis is trustworthy

Figure 53. VerifAI-RM Trustworthiness

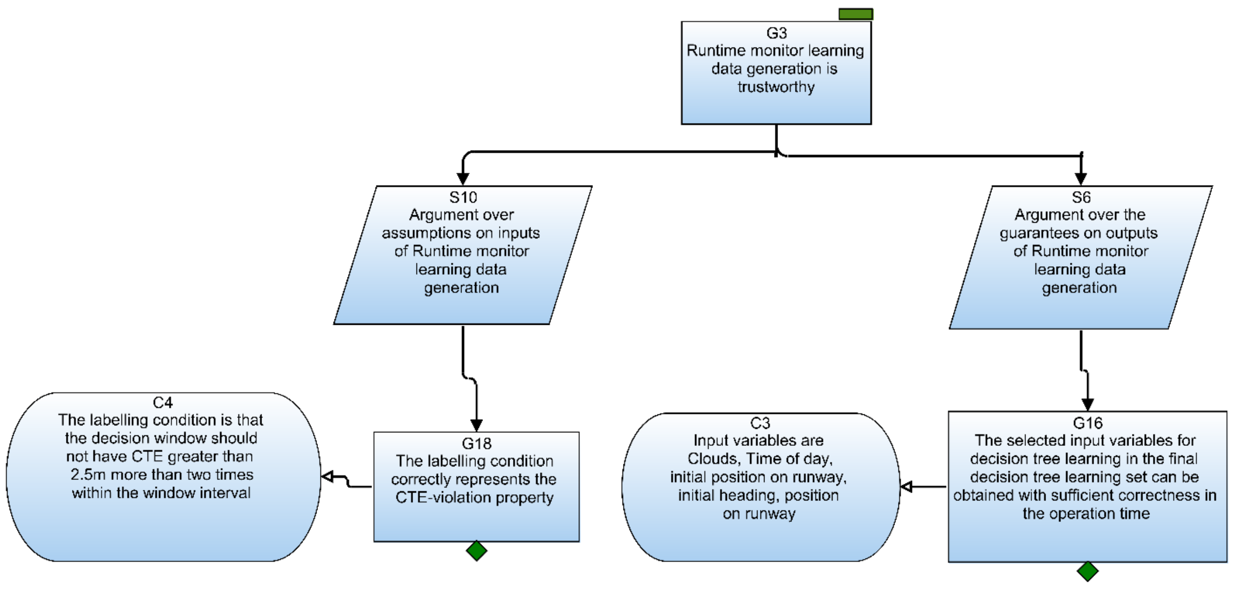

G3
Runtime monitor learning data generation is trustworthy
S10
Argument over assumptions on inputs of Runtime monitor learning data generation
S6
Argument over the guarantees on outputs of Runtime monitor learning data generation
C4
The labelling condition is that the decision window should not have CTE greater than 2.5m more than two times within the window interval
G18
The labelling condition correctly represents the CTE-violation property
C3
Input variables are Clouds, Time of day, initial position on runway, initial heading, position on runway
G16
The selected input variables for decision tree learning in the final decision tree learning set can be obtained with sufficient correctness in the operation time

Figure 54. VerifAI-RM Learning Data Generation Trustworthiness

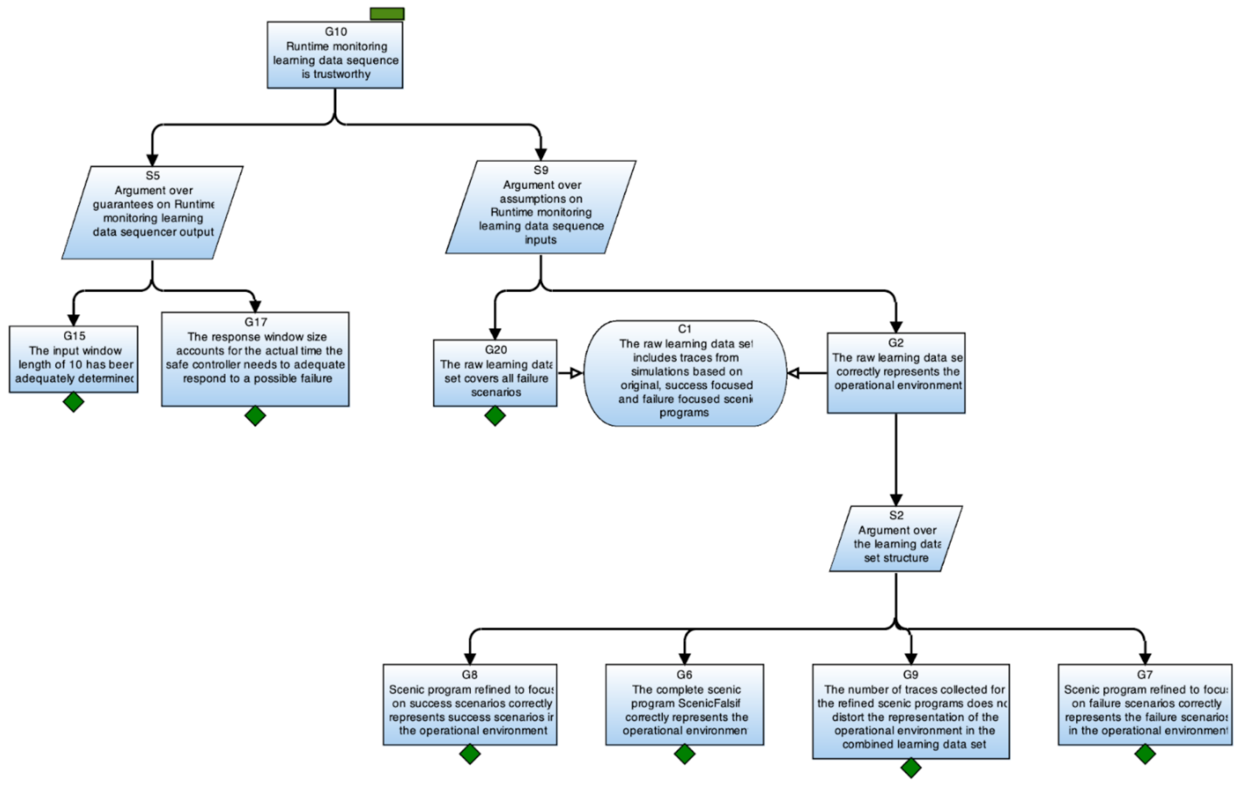

G10
Runtime monitoring learning data sequence is trustworthy
S5
Argument over guarantees on Runtime monitoring learning data sequencer output
S9
Argument over assumptions on Runtime monitoring learning data sequence inputs
G15
The input window length of 10 has been adequately determined
G17
The response window size accounts for the actual time the safe controller needs to adequate respond to a possible failure
G20
The raw learning data set covers all failure scenarios
C1
The raw learning data set includes traces from simulations based on original, success focused and failure focused scenic programs
G2
The raw learning data set correctly represents the operational environment
S2
Argument over the learning data set structure
G8
Scenic program refined to focus on success scenarios correctly represents success scenarios in the operational environment
G6
The complete scenic program ScenicFalsif correctly represents the operational environment
G9
The number of traces collected for the refined scenic programs does not distort the representation of the operational environment in the combined learning data set
G7
Scenic program refined to focus on failure scenarios correctly represents the failure scenarios in the operational environment

Figure 55. VerifAI-RM Learning Data Sequencer Trustworthiness

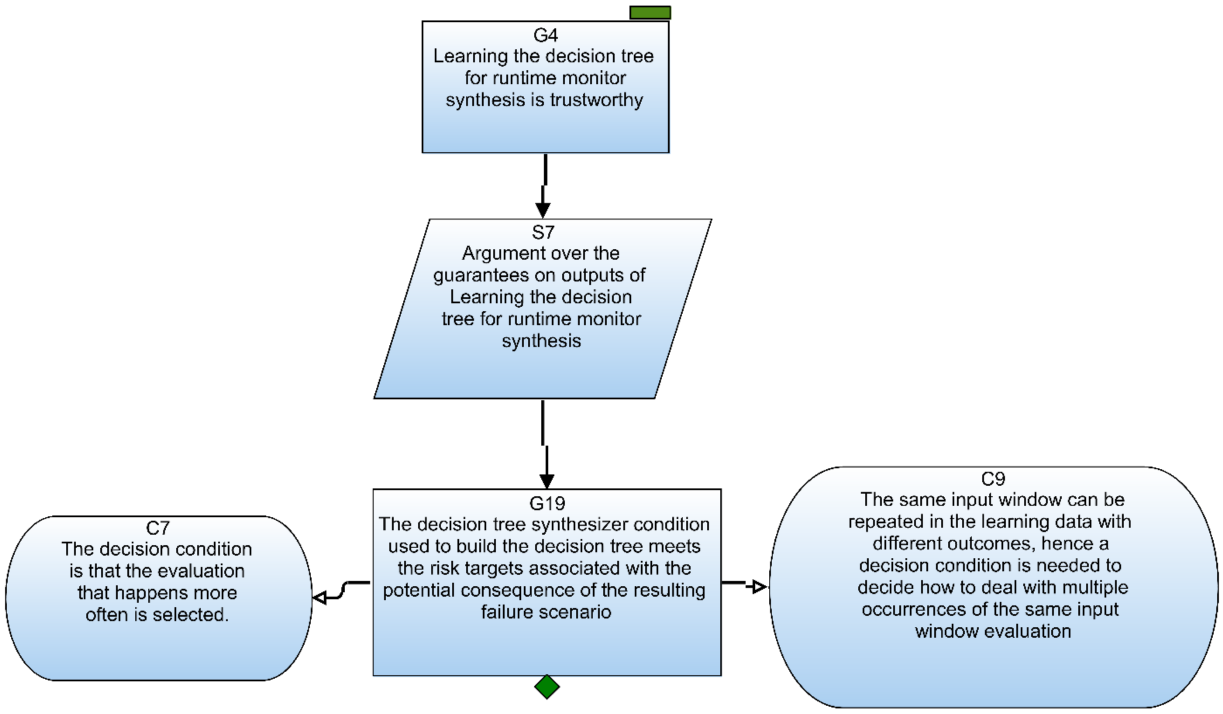

G4
Learning the decision tree for runtime monitor synthesis is trustworthy
S7
Argument over the guarantees on outputs of Learning the decision tree for runtime monitor synthesis
G19
The decision tree synthesizer condition used to build the decision tree meets the risk targets associated with the potential consequence of the resulting failure scenario
C7
The decision condition is that the evaluation that happens more often is selected.
C9
The same input window can be repeated in the learning data with different outcomes, hence a decision condition is needed to decide how to deal with multiple occurrences of the same input window evaluation

Figure 56. VerifAI-RM Decision Tree Learning Trustworthiness

4.3.2 Venus

4.3.2.1 Tool Overview

Venus [VEN2020] is a verification toolkit for ReLU-based feed-forward neural networks. Given a feed-forward NN, Venus answers a verification problem with YES or NO as to whether the NN for every input within a linearly definable set of inputs is contained within a linearly definable set of outputs. To optimize the verification, the Venus *Splitter* (Figure 57) decomposes the verification problem into smaller, more manageable chunks so that they can be executed in parallel. Each sub-problem is encoded as a Mixed Integer Linear Program (MILP) by the *MILP encoder* component.

An external *MILP solver* is called to solve each sub-problem independently. Due to the high dimensionality of inputs that the MILP solver would have to handle, Venus implements a *Dependency analyzer* as a callback function supplied to the *MILP solver*, to reduce the dimensionality of inputs that need to be verified. The *Dependency Analyzer* component excludes neurons that have no effect on the result of verification of the particular sub-problem, and in that way speeds up the MILP Solver. The MILP Solver returns a YES/NO answer as to whether the NN returns the same answer for all the inputs. Finally, the *Results integrator* takes the results from all the sub-problems and comes up with the final result for the initial verification problem. If there were any results with NO, the final result will be a NO, and it would show the input for which the result was NO. And if all are YES, the result is YES.

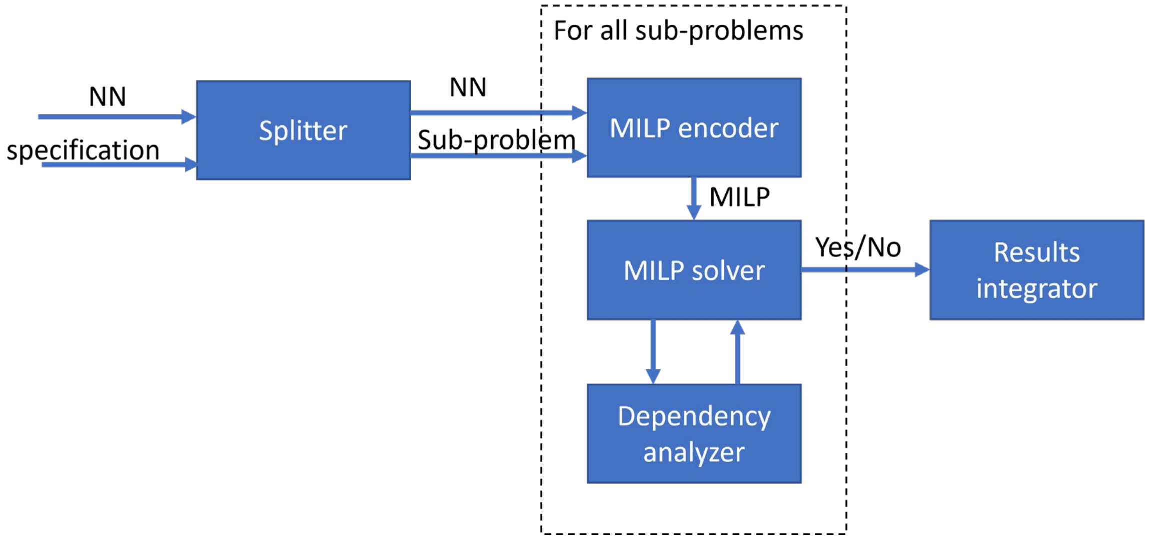


Figure 57. High Level Venus Overview

4.3.2.2 Venus Trustworthiness Assurance Case

To assure the trustworthiness of Venus, we developed a simple assurance case to capture Venus internal structure, risks, and how these risks are mitigated. We define the internal structure in terms of functional decomposition, then we perform hazard analysis and allocate the different hazards to the defined functions. We then give a collection of tool specifications and define assume/guarantee contracts such that each low-level hazard is addressed by at least one tool guarantee. We then use the tool specification to semi-automatically create an argument assuring the tool trustworthiness based on the hazard analysis and the tool contracts. We first automatically generate the argument based on the Hazard Log and Tool Specification information, and then customize it with additional information.

Venus Functional Decomposition

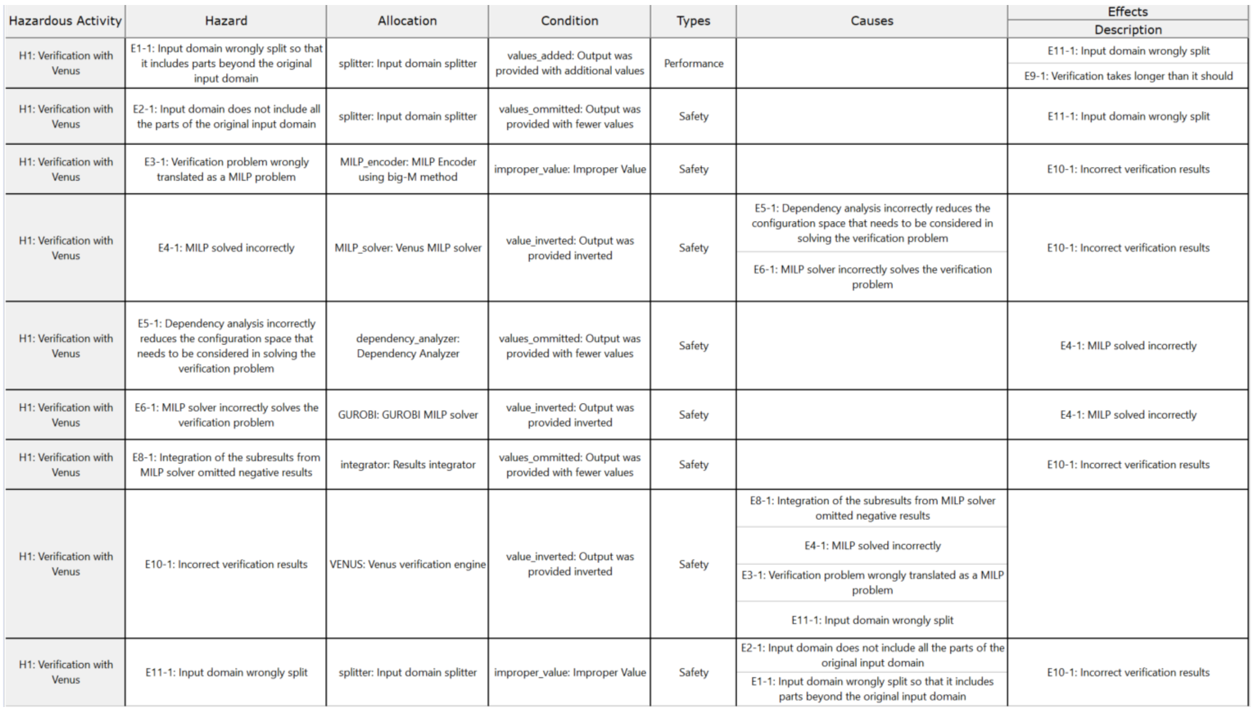
The functional decomposition in Figure 58 specifies the different functions of Venus, as highlighted in Figure 57. We define Venus with a splitter, MILP encoder, MILP solver and integrator functions. We further decompose the MILP solver function with a GUROBI solver and dependency analyzer functions. We add specific deviations to each function that are used to guide the hazard analysis.

```
deviation malfunction "Malfunction (Function available, but output incorrect)" {
    deviation improper_value "Improper Value" {
        deviation values_added "Output was provided with additional values"
        deviation values_ommitted "Output was provided with fewer values"
        deviation value_inverted "Output was provided inverted"

    }
}
function VENUS "Venus verification engine" system {
    deviations [value_inverted]
     function splitter "Input domain splitter" system
     {deviations [improper_value,values_added,values_ommitted]}
     function MILP_encoder "MILP Encoder using big-M method" system
     {deviations [improper_value]}
     function MILP_solver "Venus MILP solver" system{
         deviations [value_inverted]
        function GUROBI "GUROBI MILP solver" system
     {deviations [value_inverted]}
        function dependency_analyzer "Dependency Analyzer" system
     {deviations [values_added,values_ommitted]}
           }
    function integrator "Results integrator" system
    {deviations [values_ommitted]}
}
```

Figure 58. Venus Functional Decomposition

Venus Hazard Log

In the hazard analysis we identified six low level hazards, covering each Venus function. The Splitter function could split the domain wrongly to reduce the initial input domain or it could add parts beyond the initial input. Each sub-problem could be wrongly translated as MILP, which would render the subsequent analysis useless. In the case of MILP solving, Venus exploits the special bonds between neurons and reduces the configuration space that needs to be considered by the GUROBI solver. The two hazards there relate to incorrect dependency analysis and MILP solving.



Hazardous Activity	Hazard	Allocation	Condition	Types	Causes	Effects
						Description
H1: Verification with Venus	E1-1: Input domain wrongly split so that it includes parts beyond the original input domain	splitter: Input domain splitter	values_added: Output was provided with additional values	Performance		E11-1: Input domain wrongly split E9-1: Verification takes longer than it should
H1: Verification with Venus	E2-1: Input domain does not include all the parts of the original input domain	splitter: Input domain splitter	values_ommitted: Output was provided with fewer values	Safety		E11-1: Input domain wrongly split
H1: Verification with Venus	E3-1: Verification problem wrongly translated as a MILP problem	MILP_encoder: MILP Encoder using big-M method	improper_value: Improper Value	Safety		E10-1: Incorrect verification results
H1: Verification with Venus	E4-1: MILP solved incorrectly	MILP_solver: Venus MILP solver	value_inverted: Output was provided inverted	Safety	E5-1: Dependency analysis incorrectly reduces the configuration space that needs to be considered in solving the verification problem E6-1: MILP solver incorrectly solves the verification problem	E10-1: Incorrect verification results
H1: Verification with Venus	E5-1: Dependency analysis incorrectly reduces the configuration space that needs to be considered in solving the verification problem	dependency_analyzer: Dependency Analyzer	values_ommitted: Output was provided with fewer values	Safety		E4-1: MILP solved incorrectly
H1: Verification with Venus	E6-1: MILP solver incorrectly solves the verification problem	GUROBI: GUROBI MILP solver	value_inverted: Output was provided inverted	Safety		E4-1: MILP solved incorrectly
H1: Verification with Venus	E8-1: Integration of the subresults from MILP solver omitted negative results	integrator: Results integrator	values_ommitted: Output was provided with fewer values	Safety		E10-1: Incorrect verification results
H1: Verification with Venus	E10-1: Incorrect verification results	VENUS: Venus verification engine	value_inverted: Output was provided inverted	Safety	E8-1: Integration of the subresults from MILP solver omitted negative results E4-1: MILP solved incorrectly E3-1: Verification problem wrongly translated as a MILP problem E11-1: Input domain wrongly split	
H1: Verification with Venus	E11-1: Input domain wrongly split	splitter: Input domain splitter	improper_value: Improper Value	Safety	E2-1: Input domain does not include all the parts of the original input domain E1-1: Input domain wrongly split so that it includes parts beyond the original input domain	E10-1: Incorrect verification results

Figure 59. Internal Venus Hazard Analysis

Venus Tool Specification

Given that many verification methods are implemented by a collection of individual tools, rather than a single tool, we introduce the concept of a Toolset in our Tool Specification. A set of interconnected tools can be grouped into toolsets. We consider Venus as a toolset and define each function as a separate tool with its own tool specification. In the tool specification, we define tool inputs and outputs in terms of evidence types, and we additionally indicate different assumptions and guarantees over each tool's inputs and outputs. For example, an assumption on the NN input to the toolset for the Splitter component (Figure 60) is that it is a feed-forward ReLU neural network, since Venus supports that kind of network. Furthermore, the input for which the robustness should be checked needs to match the input structure of the network. The guarantees of the Splitter function ensure that the generated sub-problems are all part of the original verification problem, and that by composing all the sub-problems we get the initial verification problem. The tools specification for all the Venus tools are shown in Figure 60 and Figure 61.

Reusable Venus Argument

Each low-level hazard is addressed by at least one guarantee of the function to which the hazard is allocated. We argue that the risk of incorrect results with Venus is managed by addressing each identified hazard. Then we use the supporting guarantees from tool specification for each hazard to instantiate a supporting argument for each hazard using the tool specification.

```
tool T1 {
       description "Splitter"
       inputs [ NN : myevidence.EvidenceType.NN_model ,
           input_image : myevidence.EvidenceType.venus_input_image,
           perturbation_radius:real]
       assumptions [
       "The input LEC "+{NN}+" is a feed-forward ReLU neural network",
       "The input image "+{input_image }+" representation matches the LEC input
       structure"]
       outputs [sub_problem : myevidence.EvidenceType.venus_sub_problems ]
       guarantees ["Every "+{sub_problem}+" derived by the tool Splitter is a part of
       the verification problem defined by the input image "+{input_image} + "and the
       indicated perturbation radius" +{perturbation_radius},
       "For every part of the verification problem defined by the input image
       "+{input_image} + "and the perturbation radius " +{perturbation_radius}+
       "there is a sub-problem "+{sub_problem}+" generated by the tool Splitter",
       {sub_problem}+" matches the "+{NN}+" input structure"]
}

tool T2 {
       description "MILP Encoder"
       inputs [  sub_problem : myevidence.EvidenceType.venus_subproblem , NN :
       myevidence.EvidenceType.NN_model ]
       assumptions [
       "The input sub-problem "+{sub_problem}+" matches the LEC "+{NN}+" input
       structure"]
       outputs [ econded_MILP : myevidence.EvidenceType.MILP ]
       guarantees ["The resulting encoded MILP "+{econded_MILP}+" correctly
       represents the sub-problem "+{sub_problem}+" and the input LEC "+{NN} ]
}
```

Figure 60. Splitter and MILP Encoder Tool Specification

```
tool T3 {
       description "MILP Solver: Gurobi"
       inputs [ encoded_MILP : myevidence.EvidenceType.MILP , optimized_MILP :
       myevidence.EvidenceType.optimized_MILP ]
       assumptions [
       "Optimized MILP "+{optimized_MILP}+"is equivalent to the input encoded MILP
       "+{encoded_MILP}]
       outputs [ sub_problem_result : myevidence.EvidenceType.venus_result ]
       guarantees [
       "Gurobi solver outputs the correct solution "+{sub_problem_result}+ " for the
       optimized MILP" +{optimized_MILP}]
}

tool T4 {
       description "Dependency Analyzer"
       inputs [ encoded_MILP : myevidence.EvidenceType.MILP ]
       outputs [ optimized_MILP : myevidence.EvidenceType.optimized_MILP ]
       guarantees [
       "Optimized MILP "+{optimized_MILP}+"is equivalent to the input encoded MILP
       "+{encoded_MILP}]
}

tool T5 {
     description "Results Integrator"
     inputs [ all_sub_problem_results : myevidence.EvidenceType.venus_result_set,
         all_sub_problems: myevidence.EvidenceType.venus_sub_problems,
         input_image : myevidence.EvidenceType.venus_input_image,
         perturbation_radius:real]
     assumptions[
       "The set of all results "+{all_sub_problem_results}+ " includes results from
       every sub-problem "+{all_sub_problems}+" generated by Splitter for the given
       verification problem",
        "Composition of all the sub-problems "+{all_sub_problems}+" is equivalent to
       the initial verification problem"]
     outputs [ final_result : myevidence.EvidenceType.venus_result ]
     guarantees [
       "The final result "+{final_result}+" equals NOT Satisfied if there is at least
       one sub-problem derived by the Splitter "+{all_sub_problem_results}+" for
       which the result of verification equals Not Satisfied"]
}
```

Figure 61. MILP Solver, Dependency Analyzer and Results Integrator Tools Specification

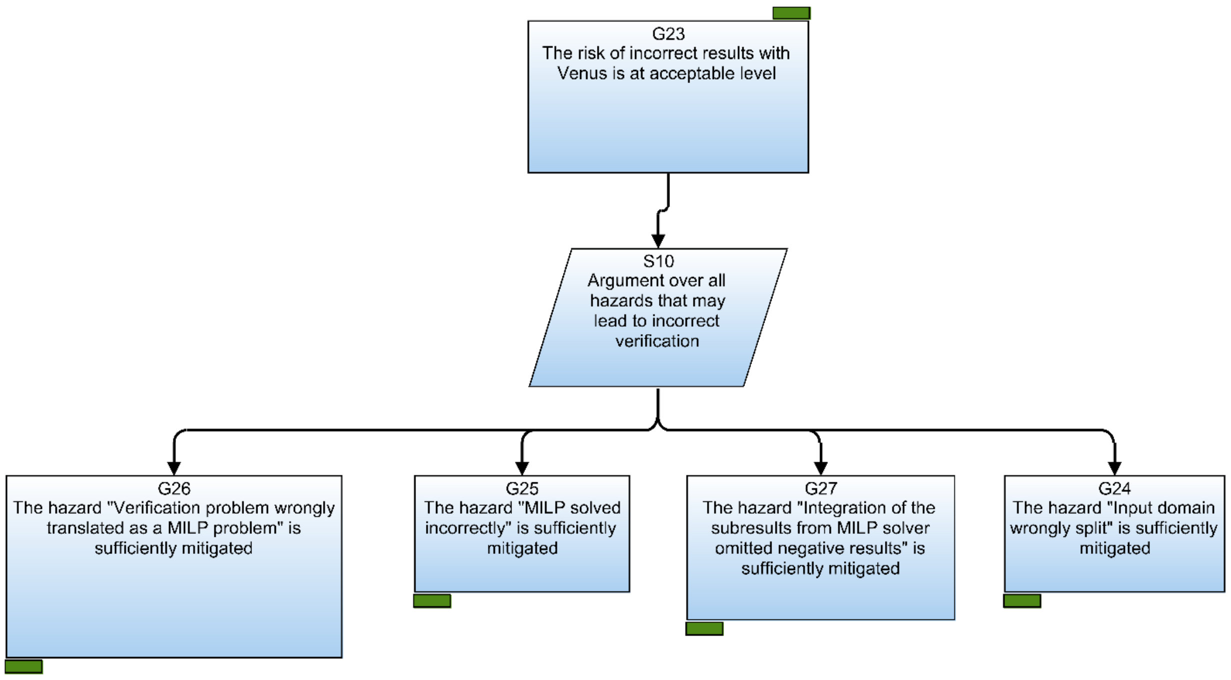

G23
The risk of incorrect results with Venus is at acceptable level
S10
Argument over all hazards that may lead to incorrect verification
G26
The hazard "Verification problem wrongly translated as a MILP problem" is sufficiently mitigated
G25
The hazard "MILP solved incorrectly" is sufficiently mitigated
G27
The hazard "Integration of the subresults from MILP solver omitted negative results" is sufficiently mitigated
G24
The hazard "Input domain wrongly split" is sufficiently mitigated

Figure 62. Reusable Argument Fragment for Mitigation of Internal Venus Risks

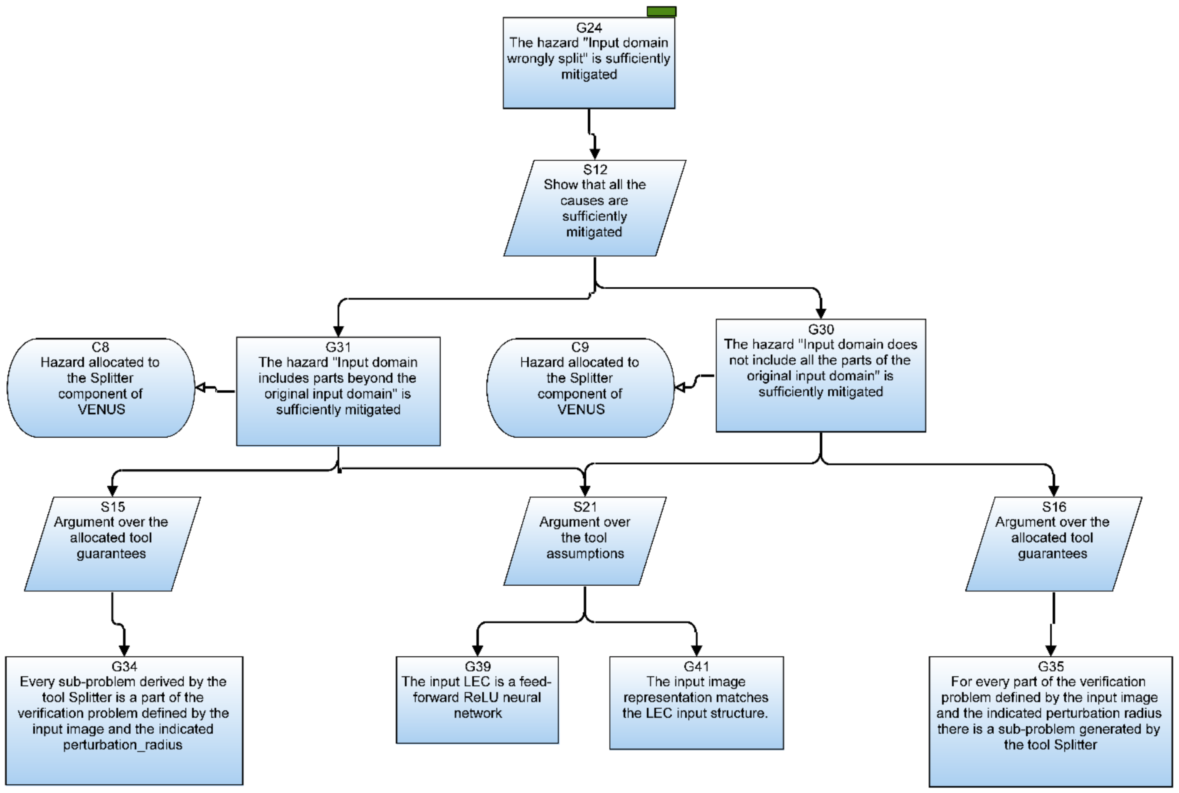

G24
The hazard "Input domain wrongly split" is sufficiently mitigated
S12
Show that all the causes are sufficiently mitigated
C8
Hazard allocated to the Splitter component of VENUS
G31
The hazard "Input domain includes parts beyond the original input domain" is sufficiently mitigated
C9
Hazard allocated to the Splitter component of VENUS
G30
The hazard "Input domain does not include all the parts of the original input domain" is sufficiently mitigated
S15
Argument over the allocated tool guarantees
S21
Argument over the tool assumptions
S16
Argument over the allocated tool guarantees
G34
Every sub-problem derived by the tool Splitter is a part of the verification problem defined by the input image and the indicated perturbation_radius
G39
The input LEC is a feed-forward ReLU neural network
G41
The input image representation matches the LEC input structure.
G35
For every part of the verification problem defined by the input image and the indicated perturbation radius there is a sub-problem generated by the tool Splitter

Figure 63. Argument Fragment for Incorrect Domain Splitting

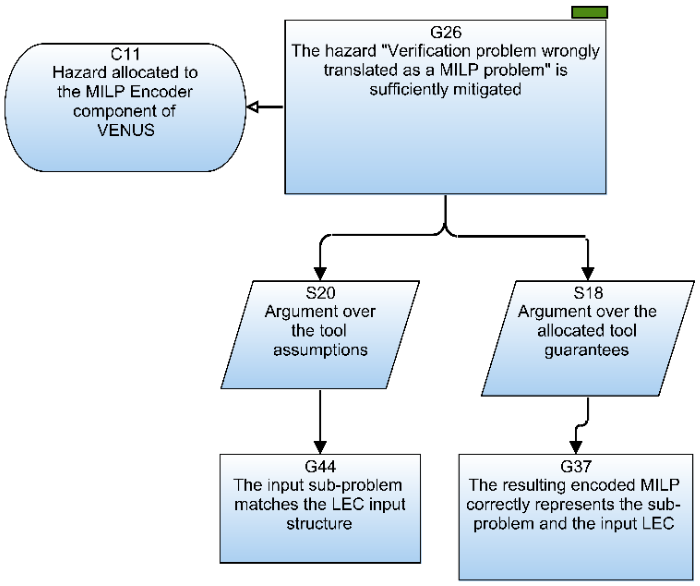

G26
The hazard "Verification problem wrongly translated as a MILP problem" is sufficiently mitigated
C11
Hazard allocated to the MILP Encoder component of VENUS
S20
Argument over the tool assumptions
S18
Argument over the allocated tool guarantees
G44
The input sub-problem matches the LEC input structure
G37
The resulting encoded MILP correctly represents the sub-problem and the input LEC

Figure 64. Argument Fragment for Verification Problem Translation into MILP

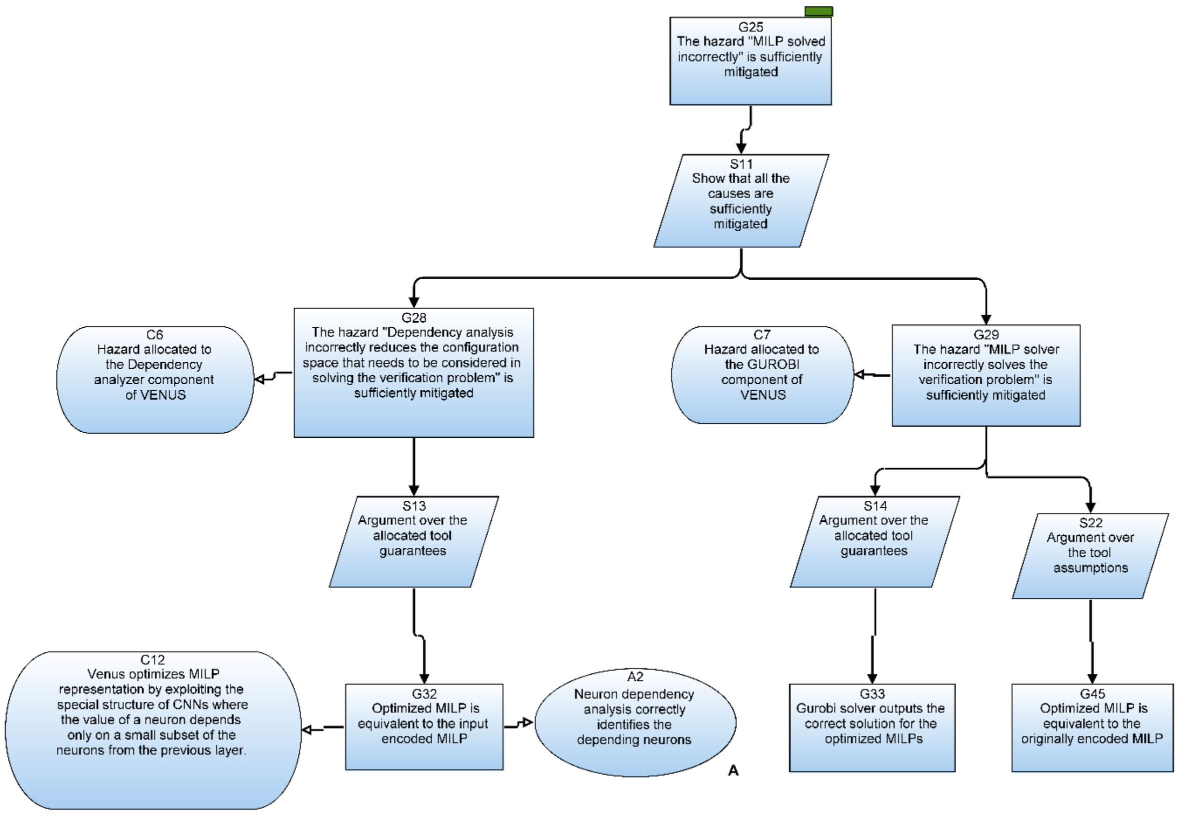

G25
The hazard "MILP solved incorrectly" is sufficiently mitigated
S11
Show that all the causes are sufficiently mitigated
G28
The hazard "Dependency analysis incorrectly reduces the configuration space that needs to be considered in solving the verification problem" is sufficiently mitigated
C6
Hazard allocated to the Dependency analyzer component of VENUS
C7
Hazard allocated to the GUROBI component of VENUS
G29
The hazard "MILP solver incorrectly solves the verification problem" is sufficiently mitigated
S13
Argument over the allocated tool guarantees
S14
Argument over the allocated tool guarantees
S22
Argument over the tool assumptions
C12
Venus optimizes MILP representation by exploiting the special structure of CNNs where the value of a neuron depends only on a small subset of the neurons from the previous layer.
G32
Optimized MILP is equivalent to the input encoded MILP
A2
Neuron dependency analysis correctly identifies the depending neurons
A
G33
Gurobi solver outputs the correct solution for the optimized MILPs
G45
Optimized MILP is equivalent to the originally encoded MILP

Figure 65. Argument Fragment for MILP Solver Correctness

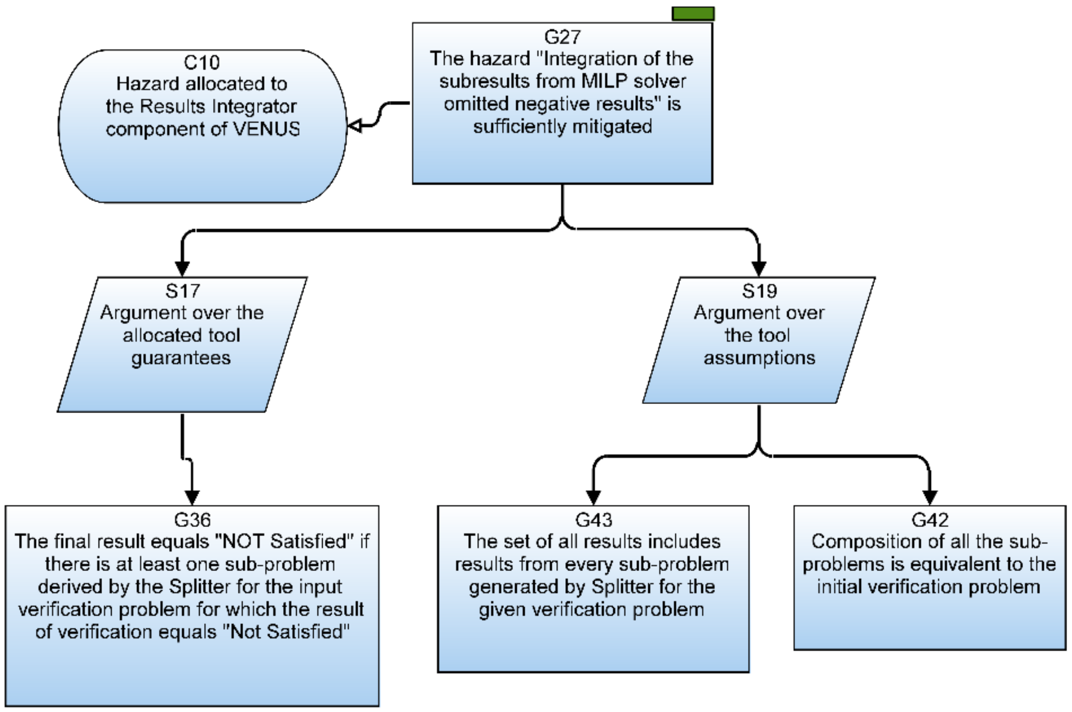

C10
Hazard allocated to the Results Integrator component of VENUS
G27
The hazard "Integration of the subresults from MILP solver omitted negative results" is sufficiently mitigated
S17
Argument over the allocated tool guarantees
S19
Argument over the tool assumptions
G36
The final result equals "NOT Satisfied" if there is at least one sub-problem derived by the Splitter for the input verification problem for which the result of verification equals "Not Satisfied"
G43
The set of all results includes results from every sub-problem generated by Splitter for the given verification problem
G42
Composition of all the sub-problems is equivalent to the initial verification problem

Figure 66. Argument Fragment for Final Result Integration

4.3.2.3 Venus Application for Object Detection with Open Categories (CP 2.2)

Boeing provided the CNN and a single data set with three classes of objects: ground vehicle, person, and airplane. The NN classifier will encounter novel objects in practice, such as novel types of ground vehicles and static objects. The CNN was trained using vehicles and persons, while the third class was used to test the performance of the trained network. Imperial specified the verification problem as a local robustness problem of the Boeing-provided CNN. For a correctly classified image, the verification problem checks that the CNN returns the same result for all the images within a specified perturbation radius. Venus was run for a set of correctly classified images from the dataset. Each image was verified for different perturbation radii, to determine how robust is the CNN for different perturbation radii. For each of the images and perturbation radii, the CNN and the verification problem are encoded as MILP, and the MILP solving based on splitting and dependency optimization is performed. The results showed that for a small perturbation radius, the CNN was robust for all the images. As the radius was increasing, the robustness was decreasing, and more misclassifications were detected.

Venus Tool Specification in CP2.2

The application specific tool specification captures the user perspective of using Venus in CP 2.2. To apply Venus, we consider as the main input the image data set provided by Boeing. Before applying the tool Venus, the first activity is to select a set of images for verification from the provided image data set. We capture this in the tool, A1. The assumptions emphasize that we expect the image data set provided by Boeing to be representative of the operational context, that the images are correctly classified, and that the number of images selected for verification is enough compared to the size of the Boeing image data set.

```
tool T1 {
        description "VENUS"
        inputs [ image_input : ev.EvidenceType.image_pickle_format , perturbation_radius : real ,
        input_CNN : ev.EvidenceType.mnist_NN ]
        outputs [ robust : boolean ]
        toolset TS1
}

tool A1 {
        description "Image Selection for Verification"
        inputs [ image_dataset : ev.EvidenceType.ImageDataSet , selection_count : int]
        assumptions[
        {image_dataset}+" is representative of the operational context",
        "Images in "+{image_dataset}+" are correctly classified",
        {selection_count}+" images selected for verification is enough compared to the size of the
        "+{image_dataset}]
        outputs [ selected_image_dataset : ev.EvidenceType.ImageDataSet ]
        guarantees[{selected_image_dataset}+" is a representative sample from "+{image_dataset},
        "The size of " +{selected_image_dataset} + " is "+{selection_count},
        Each image in pickle format in "+{selected_image_dataset}+" is the correct representation of the
        corresponding image in "+{image_dataset}
        ]
        toolset TS1
}

tool A2 {
        description "DataSetVerification"
        inputs [ input_CNN : ev.EvidenceType.mnist_NN , perturbation_radius : real ,
        verification_image_dataset : ev.EvidenceType.ImageDataSet ]
        assumptions[{verification_image_dataset}+" is a representative of the operational context ",
        {input_CNN}+" is a feed-forward ReLU neural network",
        "Image perturbations within the defined radius"+{perturbation_radius}+" represent the likely
        perturbations for the given operational context"]
        outputs [ results_dataset : ev.EvidenceType.venus_dataset_results , number_of_robust : int ]
        guarantees[
        {number_of_robust}+" images from "+{verification_image_dataset}+" is robust",
        {results_dataset}+" has one result for each image from " + {verification_image_dataset}]
        toolset TS1
}

tool A3 {
        description "CalculateRobustnessLevel"
        inputs [ venus_results : ev.EvidenceType.venus_dataset_results , perturbation_radius :
        real,input_CNN : ev.EvidenceType.mnist_NN , verified_images_set : ev.EvidenceType.ImageDataSet ]
        assumptions[
        {venus_results}+" has one result for each image from "+{verified_images_set},
        "Each result in "+{venus_results}+" was obtained for the same perturbation radius level
        "+{perturbation_radius}]
        outputs [ robustness_level : real ]
        guarantees [
        {input_CNN}+" is " + { robustness_level } + "% robust with respect to the perturbation radius " +
        { perturbation_radius } + " in the operational context defined by " + { verified_images_set }]
        toolset TS1
}
```

Figure 67. Tools Specification for Venus Application

Each of these assumptions will be instantiated for the specific tool usage and will need to be further assured in the corresponding argument. Furthermore, the A1 activity guarantees that the data set of selected images is representative of the original Boeing data set, that the size of the selected images data set is correct, and that the selected images transformed to the pickle format, which is needed for verification, correctly represent the original images.

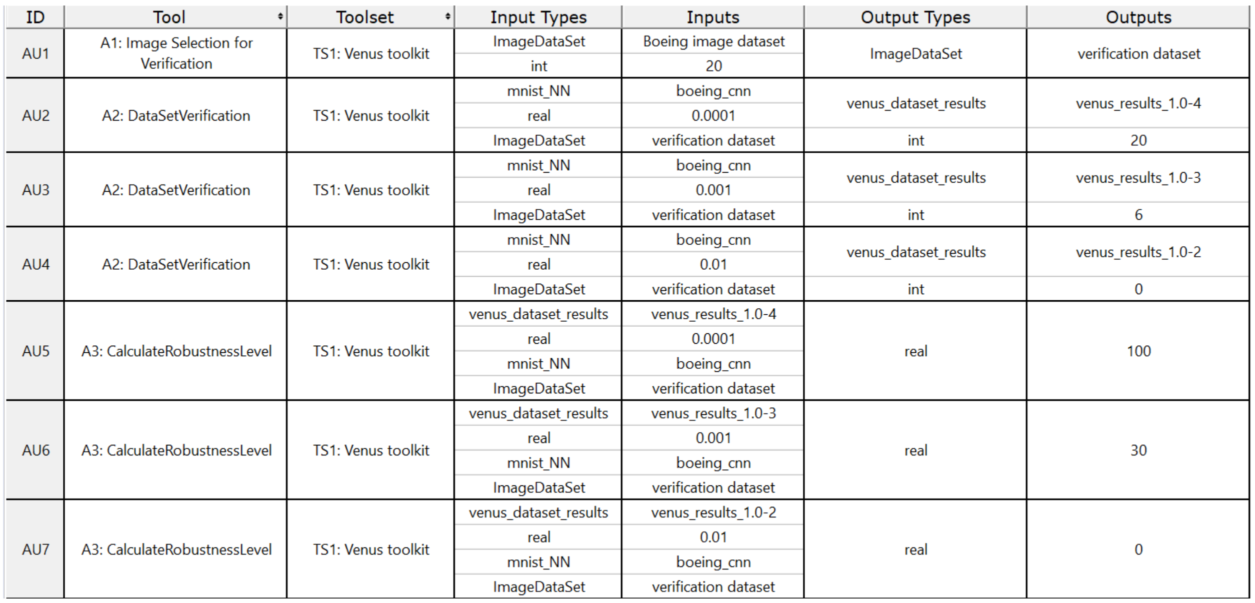
The next activity, A2, deals with the verification of the entire data set of selected images. It guarantees that all images in the dataset are verified using the same perturbation radius and that there is a single result for each image. A2 assumes that the image perturbations within the defined radius are likely for the given operational context. The tool T1 captures the Venus executable that takes in the NN in MNIST (Modified National Institute of Standards and Technology) format, an image in pickle format and a real value for the perturbation radius. It returns a yes/no value whether the NN is robust for the given image and the given perturbation radius. We run this tool for each image in the selected images data set. Once we have all the results, we perform the A3 activity to calculate the robustness level for the selected images data set.

Venus Tool Use Specification in CP2.2

We record each usage of the specified tools with concrete values as inputs and outputs in the Tool Uses table. That table is currently manually populated, but we have prepared everything for the tools such as Venus to be called directly from AdvoCATE and their results automatically recorded in the Tool Uses table.

In the tables in Figure 68, we show an excerpt of the recorded uses of the specified tools and activities. The verification is performed so that the first 20 images are selected from the Boeing image data set. Then three different data set verification activities are performed for the given NN and the selected images data set. One for each perturbation radius value 0.0001, 0.001, and 0.01.

The results indicate the number of robust images in the selected data set for the corresponding perturbation radius. To obtain these results, we captured each Venus invocation for each selected image, NN and perturbation radius. Finally, the calculated robustness level activity simply returns a percentage of robust images in the selected images data set for each perturbation radius.



ID	Tool	Toolset	Input Types	Inputs	Output Types	Outputs
AU1	A1: Image Selection for Verification	TS1: Venus toolkit	ImageDataSet	Boeing image dataset	ImageDataSet	verification dataset
			int	20		
AU2	A2: DataSetVerification	TS1: Venus toolkit	mnist_NN	boeing_cnn	venus_dataset_results	venus_results_1.0-4
			real	0.0001		
			ImageDataSet	verification dataset	int	20
AU3	A2: DataSetVerification	TS1: Venus toolkit	mnist_NN	boeing_cnn	venus_dataset_results	venus_results_1.0-3
			real	0.001		
			ImageDataSet	verification dataset	int	6
AU4	A2: DataSetVerification	TS1: Venus toolkit	mnist_NN	boeing_cnn	venus_dataset_results	venus_results_1.0-2
			real	0.01		
			ImageDataSet	verification dataset	int	0
AU5	A3: CalculateRobustnessLevel	TS1: Venus toolkit	venus_dataset_results	venus_results_1.0-4	real	100
			real	0.0001		
			mnist_NN	boeing_cnn		
			ImageDataSet	verification dataset		
AU6	A3: CalculateRobustnessLevel	TS1: Venus toolkit	venus_dataset_results	venus_results_1.0-3	real	30
			real	0.001		
			mnist_NN	boeing_cnn		
			ImageDataSet	verification dataset		
AU7	A3: CalculateRobustnessLevel	TS1: Venus toolkit	venus_dataset_results	venus_results_1.0-2	real	0
			real	0.01		
			mnist_NN	boeing_cnn		
			ImageDataSet	verification dataset		

Figure 68. Excerpt of the Venus Tool Use Table

Evidence Log

All the inputs and outputs from the different Tool Uses are captured in the Evidence Log. The definition of each Evidence artifact includes the description, type, and status of the Evidence. Additionally, information like evidence relations, location of the evidence, version, etc. can also be specified. Figure 69 represents an Evidence dependency graph for the venus_results_1.0-3 Evidence item. An excerpt of the Evidence log is shown in Figure 70.

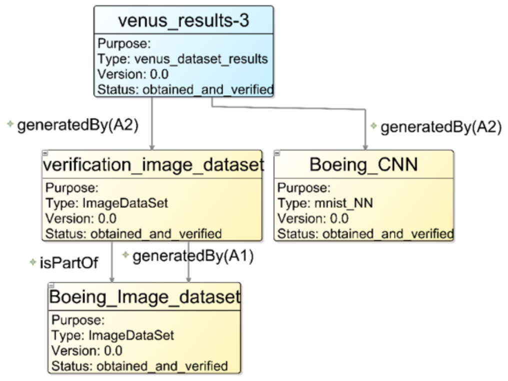


Figure 69. Evidence Dependency Diagram For Results With Perturbation Radius 0.001

```
artifact Boeing_Image_dataset {
      description "Boeing image dataset"
      type ImageDataSet
      status obtained_and_verified
}

artifact verification_image_dataset {
      description "Verification dataset"
      type ImageDataSet
      status obtained_and_verified
      generatedBy A1 with Boeing_Image_dataset
      isPartOf Boeing_Image_dataset
}

artifact Boeing_CNN {
      description "Boeing cnn"
      type mnist_NN
      status obtained_and_verified
}

artifact venus_results-4 {
      description "venus_results_1.0-4"
      type venus_dataset_results
      status obtained_and_verified
      generatedBy A2 with verification_image_dataset and Boeing_CNN
}

artifact venus_results-3 {
      description "venus_results_1.0-3"
      type venus_dataset_results
      status obtained_and_verified
      generatedBy A2 with Boeing_CNN and verification_image_dataset
}

artifact venus_results-2 {
      description "venus_results_1.0-2"
      type venus_dataset_results
      status obtained_and_verified
      generatedBy A2 with Boeing_CNN and verification_image_dataset
}
```

Figure 70. Excerpt from Evidence Log with Venus Artifacts

Challenge Problem-Specific Argument

The argument in Figure 71 through Figure 74 presents the instance of the Tool Uses for perturbation radius 0.001. The top-level claim states the obtained robustness level and can be used to support different parts of the higher-level assurance claims that deal with robustness of the CNN. The argument is broken down based on the Venus verification workflow activities, and then Venus tool usage. All the tool assumptions and guarantees are instantiated from the Tool Use information and are further enriched with contextual information. Each resulting claim needs to be further developed for the top-level claim to hold. The Venus trustworthiness argument developed separately is pointed to in the goal G23.

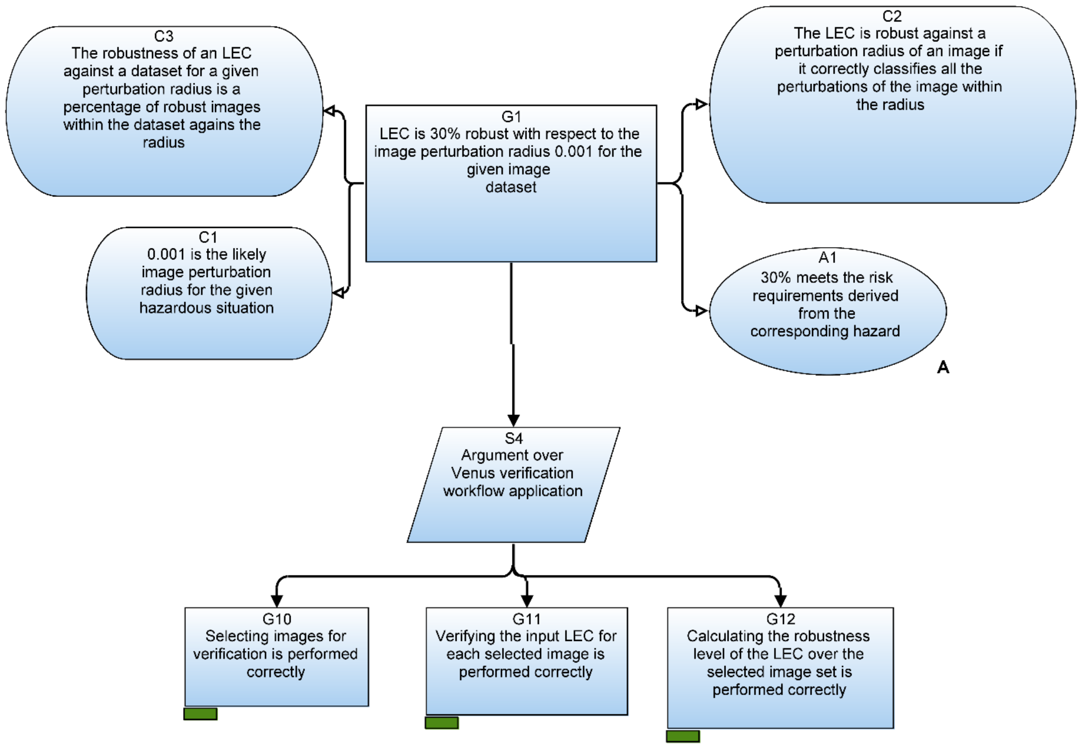

C3
The robustness of an LEC against a dataset for a given perturbation radius is a percentage of robust images within the dataset agains the radius
G1
LEC is 30% robust with respect to the image perturbation radius 0.001 for the given image dataset
C2
The LEC is robust against a perturbation radius of an image if it correctly classifies all the perturbations of the image within the radius
C1
0.001 is the likely image perturbation radius for the given hazardous situation
A1
30% meets the risk requirements derived from the corresponding hazard
A
S4
Argument over Venus verification workflow application
G10
Selecting images for verification is performed correctly
G11
Verifying the input LEC for each selected image is performed correctly
G12
Calculating the robustness level of the LEC over the selected image set is performed correctly

Figure 71. Top-Level Argument For Venus Verification Results

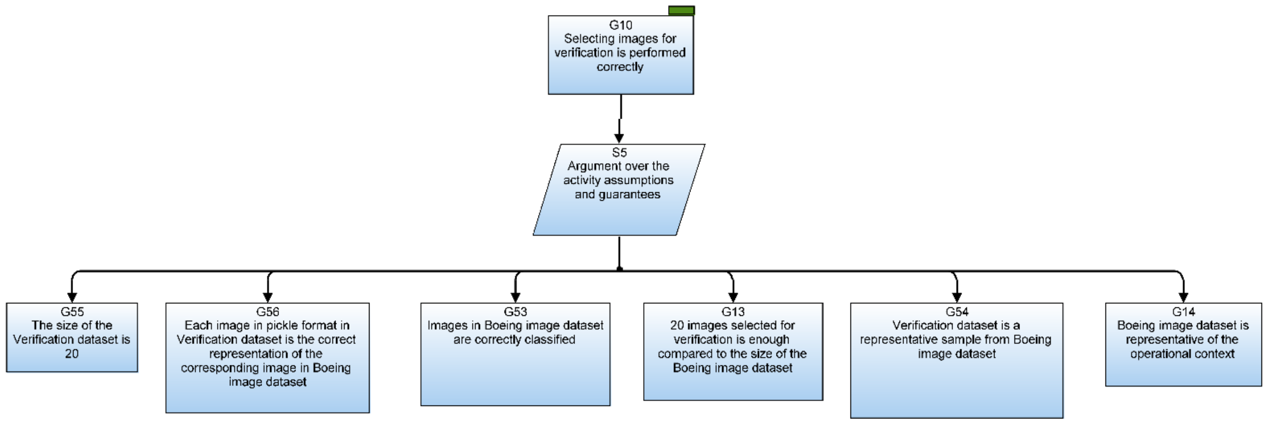

G10
Selecting images for verification is performed correctly
S5
Argument over the activity assumptions and guarantees
G55
The size of the Verification dataset is 20
G56
Each image in pickle format in Verification dataset is the correct representation of the corresponding image in Boeing image dataset
G53
Images in Boeing image dataset are correctly classified
G13
20 images selected for verification is enough compared to the size of the Boeing image dataset
G54
Verification dataset is a representative sample from Boeing image dataset
G14
Boeing image dataset is representative of the operational context

Figure 72. Argument Fragment For Image Selection in Venus

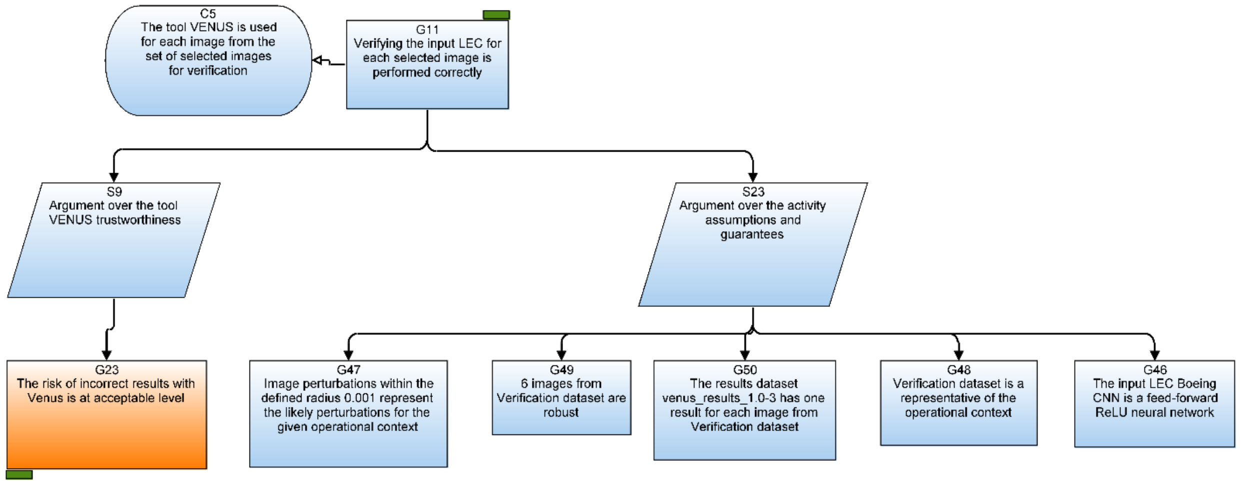

C5
The tool VENUS is used for each image from the set of selected images for verification
G11
Verifying the input LEC for each selected image is performed correctly
S9
Argument over the tool VENUS trustworthiness
S23
Argument over the activity assumptions and guarantees
G23
The risk of incorrect results with Venus is at acceptable level
G47
Image perturbations within the defined radius 0.001 represent the likely perturbations for the given operational context
G49
6 images from Verification dataset are robust
G50
The results dataset venus_results_1.0-3 has one result for each image from Verification dataset
G48
Verification dataset is a representative of the operational context
G46
The input LEC Boeing CNN is a feed-forward ReLU neural network

Figure 73. Argument Fragment Addressing Venus Verification Workflow

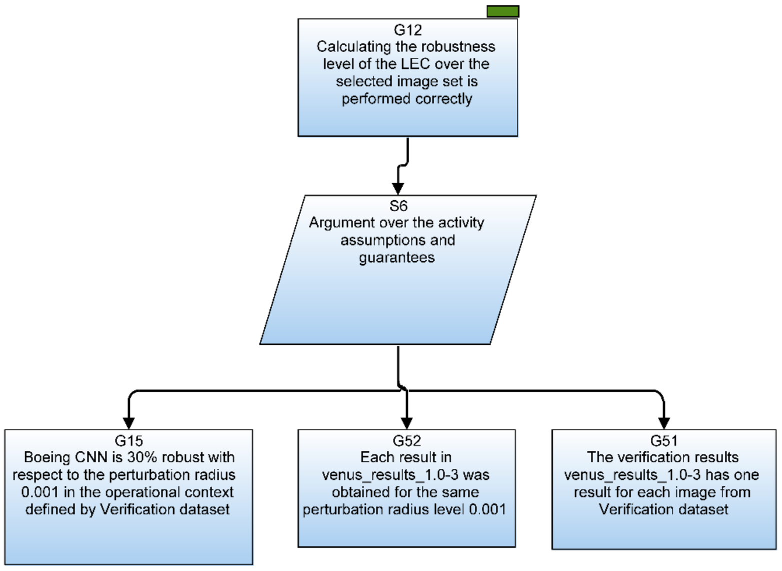

G12
Calculating the robustness level of the LEC over the selected image set is performed correctly
S6
Argument over the activity assumptions and guarantees
G15
Boeing CNN is 30% robust with respect to the perturbation radius 0.001 in the operational context defined by Verification dataset
G52
Each result in venus_results_1.0-3 was obtained for the same perturbation radius level 0.001
G51
The verification results venus_results_1.0-3 has one result for each image from Verification dataset

Figure 74. Argument Fragment for Final Calculation of a Robustness Level

4.4 AdvoCATE Extensions

Here we give an overview of some of the extensions we made to our DAC tool, AdvoCATE, in Phase 3.

4.4.1 AdvoCATE Metamodel

In the course of the QUASAR project, we made numerous extensions to the underlying assurance metamodel implemented in AdvoCATE. An assurance case consists of

- a collection of arguments
- a hazard log

- an assurance architecture, modeling risk scenarios
- a requirements log
- an evidence log, consisting of evidence artifacts and their dependencies
- a tools log, modeling external tools used to create evidence.

Here we describe the assurance artifacts which constitute an assurance case, as well as constraints and relationships between them.

4.4.1.1 Events, Controls, and Barriers

An *event* is a description of a situation or change of situation. Events are defined at the safety architecture level and so are not tied to a specific *hazardous activity* (HA). Each hazardous activity has multiple scenarios associated with it, each of which is characterized by a given *system state* (SS) and *environmental condition* (EC). Hazardous activities, system states, and environmental conditions are independent of each other, and all defined at the safety architecture level. An event can occur in different scenarios, at most once per scenario, and each such occurrence of an event is called an *event instance*.

Event instances correspond to hazards; more specifically, a *hazard* is an event instance that is *visible* in the hazard log, meaning that it appears in its own row. Causes and effects of hazards are also event instances, and if deemed visible, will also appear as hazards in their own right.

Similarly to the relation between events and event instances, *barriers* and *controls* are defined at the safety architecture level and can be applied in different scenarios to give distinct *barrier instances* and *control instances*, respectively. However, the uniqueness criteria are different: whereas an event can have at most one instance per (HA, SS, EC) scenario (i.e., CES; note, however, that event instance naming is only unique to the HA), barriers and controls can be used multiple times (each use being represented by a distinct barrier or control instance), but at most once per path segment (between event instances).

A barrier can be used with multiple controls in a segment, each of which is a distinct control instance (as well as a distinct control). However, each control instance is associated with the same barrier instance. Moreover, every instance of a control (throughout the safety architecture) must be associated with instances of the same barrier (but different barrier instances). Mitigations of hazards correspond to barrier instances.

Hazard tables record mitigations of hazards, specifically mitigations of the hazard causes (corresponding to so-called prevention barriers). Hazard consequences can also be mitigated (via recovery barriers), but these are only shown in the table if the consequence is, itself, a hazard.

4.4.1.2 Requirements

Requirements describe implementation constraints for barrier instances and control instances. The requirement is specific to the instance, rather than being at the barrier or control level, since each such application of the barrier/control represents a different scenario with potentially different implementation concerns. Requirements are also implicitly associated with hazards, by being assigned to the hazard's mitigations. In this role, they are called mitigation requirements.

4.4.1.3 Traceability

There are three kinds of tracing links:

- automatically generated trace links (either between different representations of the same underlying model element; e.g., hazard ⟷ event instance, mitigation ⟷ barrier, or as a consequence of how an element is created, e.g., evidence ⟶ tool, or an internal source)
- allocations (user-designated, and having specific semantics; e.g., hazard ⟶ item)
- rationale (to arguments/evidence) and contextual/substantiation (from arguments/evidence).

Sources

- Requirement source: internally, this can be a barrier or control instance. It can also be an external string, recorded in the sources table.
- Hazard source: As for requirements, we can have internal (in this case, a barrier) and external sources (a string). The hazard source is given at the event level.

Allocations

- Requirement to item (function or component) – the requirement allocation: the item implements the requirement
- Function to component: the component implements the function
- Requirement to evidence artifact: the verification allocation - the item provides evidence that the requirement is met
- Hazard to item – the hazard allocation: the item is the origin of the hazard (i.e., the hazardous item, which must be unique, that is to blame for the hazard). Together with the hazard condition, it characterizes the hazard
    - Event to item: the hazard allocation is actually an allocation from event to item, applicable to all instances of that event; by allocating to one instance in a hazard table, it assigns to the event and thus to all its instances
    - Event instance to barrier: this models whether a hazard (i.e., a visible event instance) has been identified in a BHA.
- Barrier instance / control instance to requirement: the requirement describes the intended functionality of the barrier or control instance
- Barrier/control to item: the item implements the barrier or control.

Various consistency relations are enforced between allocations.

Rationale

Various artifacts within an assurance case can be provided with rationale, in the form of an association to an argument or evidence artifact. The rationale can have potentially different interpretations, depending on the artifact and user intent. For example, an argument for a barrier could justify the fitness for purpose of the barrier, similarly for a control. Rationale for the elements of the risk analysis could justify the severity of terminating consequences, likelihood of initiating threats, non-derived integrity of controls and barriers, etc.

Going in the other direction, individual nodes of an argument can be associated with other elements of the assurance case, to explicate, substantiate, or otherwise provide additional context to that node of the argument. In particular:

- an evidence artifact substantiates a solution node (or relates to a contextual node)
- a requirement corresponds to a claim or context (or another node)
- an event instance, control instance, or barrier is referred to by an argument node
- a pattern node used to generate an argument node
- a data tree node is used to generate an argument node.

An argument, itself, will be associated with the data tree and pattern whose nodes are used to generate the argument's nodes.

4.4.2 Validations

The AdvoCATE metamodel (implemented using the Eclipse Modeling Framework) enforces various structural constraints, as described above, by construction, but we enforce additional constraints - called *validations* in Eclipse terminology. These are implemented using the Xtext framework, which allows richer constraints to be implemented, is more efficient than implementing directly in the model, and also allows some user control over whether these constraints are mandatory or optional, and in the former case whether these should give rise to *errors* or *warnings*. An additional reason for separating validations from core model well-formedness is that it is convenient to allow users to temporarily construct invalid assurance cases, while providing warnings about issues that should ultimately be addressed, but which do not currently impede progress. An error, on the other hand, indicates a state which could only arise through an incorrect action on the part of the user, will prevent certain actions from being applied, and should be addressed promptly.

Moreover, although the GUI prevents users from creating many invalid states, it is always possible to directly edit the DSL, and so we need to validate appropriately. Some validations are implemented with a corresponding *quick-fix* – an action that allows users to correct the error.

We have implemented over a hundred validations, of which we now give a selection.

4.4.2.1 Arguments

Multiple parents, multiple roots, non-goal root, goal-to-goal, solution has contextual, cycles, solution has evidence

4.4.2.2 Assurance Architecture

- Check Repeated Controls: Checks if a Control is repeated on a non-escalation path.
- Check Event Directly Connected To Event: Checks if two connected Events have a Control between them.
- Check Event Causality Hierarchy: Checks if an Event and its consequence have allocations with a common ancestor, and the consequence allocation is neither a parent or a sibling to the event allocation.
- Check Intermediate Event Has Allocation: Checks if an unallocated event has instances with both a consequence and a cause that are allocated
- Check Escalation Control Duplicates: Checks if the control is used as both an escalation factor control, and a regular control, on the same path
- Check Circular Consequences: Checks if the safety architecture contains any cyclical chains of event instances, considered over all CESs. Such a cycle would indicate a failure of the causal consistency between CESs.

- Check Short Circuits: Checks if there is both a direct link between two nodes, as well as a longer path containing a control instance
- Check Multiple Barrier Instances Between Events: Checks if a barrier has multiple instances between two event instances
- Check Barrier Requirement Allocation: Checks if a Barrier is missing an allocated function or component that is allocated to the requirements of its Barrier Instances
- There are other similar checks: Barrier → Function → Allocation, etc.
- Check Escalates And Mitigated By: Checks if an event instance has paths that both escalate and are mitigated by a control
- Check Sub-Hazards Are Valid: Checks if an event has an instance which does not lead to any of its parent's event instances
- Check Controls With Same Component Are Not on Common Path: Checks if two control instances are on a common path, then their controls are not allocated to any shared components

4.4.2.3 Functional Architecture

- Check Allocation Same Group: Checks if a function has multiple allocations belonging to the same group (i.e., component and sub-component as allocation)
- Check Function Deviation Has Event: Check if some function has a deviation, but there is no event to which this function and deviation are allocated
- Check Deviation Hierarchy: Checks if a function's children have deviations of a higher order than itself (if they are within the same deviation tree)
- Check Allocation Hierarchy: Checks if a function's children have allocations of a higher order than itself (if they are within the same component tree)

4.4.3 Views

As assurance cases grow in size and complexity, it becomes increasingly challenging both to manage their development, and to understand and assess them. Given the communicative role of assurance cases, the latter problem is probably the more serious. In addition to user-definable views (developed in earlier phases of the project, and not described here), AdvoCATE now additionally provides several built-in structuring mechanisms that allow assurance cases to be structured and abstracted into meaningful fragments.

4.4.3.1 Argument Views

Splits View

Arguments can be manually modularized by being decomposed into meaningful fragments by applying a *splitting* action on a selected argument node.

The first time an argument is split the original argument diagram is left unchanged, so that the user can always refer to the full argument, and a new diagram is created that copies the split region, consisting of the node at which the split was done (the "split node"), plus all nodes above or below that node.

Split nodes in the split diagrams will be annotated with a "continued elsewhere" decoration on the node in the upper argument, and a "developed from" annotation in the corresponding node in the lower argument. Navigation to specific split regions is available in the right-click menu, and double-clicking on the “developed from / continued elsewhere” annotation will navigate directly to the corresponding region. If this region was not named yet (as would be the case when navigating from the very first split region for an argument), a dialog will appear where its name must be provided to proceed. Hence, to create the other half split of the original argument, the user just needs to navigate back from the initial split and give the subsequent split region a name.

Split regions may be split further, and subsequent regions may be left unnamed, in effect hiding those parts of the argument. Unnamed regions can be navigated to only via the “continued elsewhere” decoration double-click action, which will then require them to be given a name.

For an example of the splits view—which can also be considered as an argument architecture—see Section 4.1.1.4, Figure 16.

Provenance View

The provenance view of an argument provides a graphical representation of the assurance artifacts from which an argument is created, as well as other associated assurance artifacts and their relation to the argument. In particular, it consists of a tree whose nodes represent

- the argument, itself, which gives the root
- patterns from which all or part of the argument are generated (if the argument consists of composed instance arguments, there can be multiple such patterns), including built-in patterns for the hazard and requirement logs
- data trees used to instantiate the patterns, including the data trees generated from the hazard and requirement logs
- any assurance artifacts linked as contextual elements (individual CES's or the requirements log)
- evidence artifacts linked to argument nodes,

and whose links represent the various assurance and association dependencies, such as `inContextOf` and `isGeneratedBy`. Some limitations of the provenance view currently are that it does not include external data (XML used to generate pattern data trees), nor evidence dependencies and tools.

4.4.3.2 Assurance Architecture Views

Phases View

A *Phase* represents a meaningful period within the lifetime of the system under assurance. While a CES is defined with respect to a particular scenario represented by a Hazardous Activity, System State, and Environmental Condition, phases are at a level above that, in which multiple CESs can be grouped. Phases can be organized hierarchically, so that multiple *Sub-Phases* are grouped within a single parent Phase.

Phases and their Sub-Phases are declared in the Safety Architecture DSL. A CES can then optionally be associated with a Phase, also in the DSL. In Example 1, we show a DSL fragment with two top-level phases each of which has two sub-phases:

```
phase P1 "Approach" subphases [
    phase SP1 "Base",
    phase SP2 "Final"]

phase P2 "Landing" subphases [
    phase SP1 "Flare",
    phase SP2 "Landing roll"]
```

Example 1. Example DSL Fragment with Two Top-level Phases

The subphases cannot be an empty list - if there are no subphases it should just be omitted. The names of phases (like other top-level model elements) must be unique, but subphases in distinct phases are allowed to have the same name.

A CES can then optionally be associated with a phase - the phase name must be fully qualified (i.e., the full path of hierarchical phases) and needs to be quoted. For example:

```
CES CES1 {
      system state SS2 environmental condition EC1 phase "P2.SP1"
           ... }
```

Example 2. Fully Qualified Phase Name

Note that two CESs can be in the same phase, but each CES can be in at most one phase. Inter-CES links are used to show connections between CESs. The (generated) Phase View shows CESs inside their respective subphases and phases, along with the inter-CES links, labeled by connecting event instances. From the specification of phases, subphases, and their relation to the CESs we generate the *phases view*. See Figure 15 for an example.

4.4.3.3 Table Views

Table views enable the extraction of data from the assurance case and its representation in tabular form. The use of table spanning (alignment of multiple sub-rows with a single parent row) allows dependencies between data to be represented in the table. For example, a table that displays goals, strategies, context and solution nodes, and associated evidence artifacts can be specified as follows.

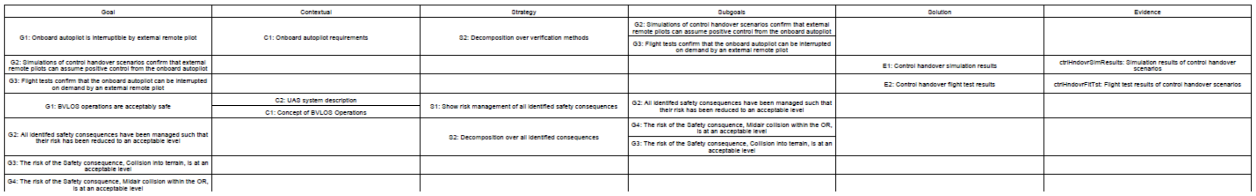
The rows keyword is used to extract the initial set of data from the assurance case that is used to construct the rows of the table, in which case, a simple query to give all goals of the argument. Next, each column of the table is specified, in turn. The first column can be optionally designated a header column, so that it is highlighted and used to label the rows. The initial row of each column (i.e., the titles) is a header row by default. If a header column is not specified, then one is generated that uses row numbers.

Each column consists of a label (giving the title of the column), a value expression, and a display expression. The value expression computes a value from corresponding cells in parent columns of the current column, and the display expression computes a string from that value (which could simply be the value, itself, if it is displayable). This allows dependencies between cells of the table to be represented. The special value expression, row, designates the value of the current (top-level) row.

```
table AllArgTable
rows ArgumentGoal

header column G {
      label "Goal"
      value row
      display name + ": " + description }
column C {
      label "Contextual"
      value G.inContextOf
      display name + ": " + description }
column S {
      label "Strategy"
      value G.isSupportedBy such that type = Strategy
      display name + ": " + description }
column SG {
      label "Subgoals"
      // for now we'll assume we don't have Goal -> Goal
      value S.isSupportedBy such that type = Goal
      display name + ": " + description }
column E {
      label "Solution"
      value G.isSupportedBy such that type = Solution
      display name + ": " + description }
column EA {
      label "Evidence"
      value E.evidenceArtifacts
      display  name + ": " + description }
```

Example 3. Table View Specification for Argument Nodes

In Example 3, for each goal of the argument, we compute all of its strategies by querying all nodes that are related to it by the `isSupportedBy` link and have node type Strategy. We then display the strategy nodes using a string constructed from the identifier (called name in the model) and description fields of the node.



Goal	Contextual	Strategy	Subgoals	Solution	Evidence
G1: Onboard autopilot is interruptible by external remote pilot	C1: Onboard autopilot requirements	S2: Decomposition over verification methods	G2: Simulations of control handover scenarios confirm that external remote pilots can assume positive control from the onboard autopilot		
			G3: Flight tests confirm that the onboard autopilot can be interrupted on demand by an external remote pilot		
G2: Simulations of control handover scenarios confirm that external remote pilots can assume positive control from the onboard autopilot				E1: Control handover simulation results	ctrlHndovrSimResults: Simulation results of control handover scenarios
G3: Flight tests confirm that the onboard autopilot can be interrupted on demand by an external remote pilot				E2: Control handover flight test results	ctrlHndovrFltTst: Flight test results of control handover scenarios
G1: BVLOS operations are acceptably safe	C2: UAS system description	S1: Show risk management of all identified safety consequences	G2: All identifed safety consequences have been managed such that their risk has been reduced to an acceptable level		
	C1: Concept of BVLOS Operations				
G2: All identifed safety consequences have been managed such that their risk has been reduced to an acceptable level		S2: Decomposition over all identified consequences	G4: The risk of the Safety consquence, Midair collision within the OR, is at an acceptable level		
			G3: The risk of the Safety consequence, Collision into terrain, is at an acceptable level		
G3: The risk of the Safety consequence, Collision into terrain, is at an acceptable level					
G4: The risk of the Safety consquence, Midair collision within the OR, is at an acceptable level					

Figure 75. Table View of Argument

```
table argHazReq-t-matrix        // generate a row for every argument node
rows ArgumentNode
header column N {
      // if no column is specified as a header column,
      // there will be a default header column with no label
      // that shows the name (row numbers is default for non-t)
      label “Nodes”

      // may be omitted to have a blank label
      // the value for the column is implicitly “row”.
      // Syntax should also allow it to be explicitly
      // specified as {value row display name} or {display row.name}
      display name }

column Hazard H such that isHazard {
      // for every one of these things, we have a column label name
      // since N is the same as row, substituting row.associatedNodes
      // or associatedNodes will result in the same table
      display if N.associatedNodes contains H then “X” else “” }

column Requirement R {
      // for every one of these things, we have a column label name
      display if N.associatedRequirements contains R then “X” else “” }
```

Example 4. Traceability Matrix Specification Linking Argument Nodes to Hazards and Requirements

Traceability matrices are used to depict relations between two sets of artifacts, such as hazards and requirements. More generally, heterogeneous traceability matrices specify relations between multiple sets of artifacts and can be thought of as a combination of simple matrices. In contrast to the types of tables described above, which have fixed columns, traceability matrices use queries to generate the columns (so do not have a fixed width), and use a special header column to give the labels of the rows. For example, to specify an ArgumentNode x (Hazard | Requirement) matrix, which shows correspondences between argument nodes and hazards and requirements, we can use the following syntax.

Example 4 has heterogenous columns, but we can also specify heterogeneous rows, to give a matrix with type `(ArgumentNode | EvidenceArtifact) x (EvidenceArtifact | Requirement)` where the (row/column) cell semantics is argument/evidence = “links to”, evidence/evidence = “depends on”, argument/requirement = “links to”, and evidence/req = “N/A”. The syntax is:

```
table complex-t-matrix
rows ArgumentNode

header column H {
   label “Arg/Ev”
   display “Node “ + name }

column EvidenceArtifact CE {
   label “Evidence “ + name
```

```
    display if row = ArgumentNode such that evidenceArtifacts contains CE
          then “links to” else “” }

column Requirement R {
    label "Requirement" + name

    // starts a second section of rows, followed by redefinitions
    // of all the columns to work with the type of the section:
    display if row = ArgumentNode such that associatedRequirements
                contains R then “links to” else “” }

rows EvidenceArtifact

header column H {
    label “Arg/Ev”
    // optional; must be the same as defined in the previous
    // section if present
    display “Evidence “ + name }

column EvidenceArtifact CE {
    label “Evidence “ + name
    display if row = EvidenceArtifact such that dependencies.to contains CE
          then “depends on” else “” }

column Requirement R  {
    label "Requirement" + name
    display “n/a” }
```

Example 5. Traceability Matrix Specification Linking Argument Nodes and Evidence to Evidence and Requirements

4.4.4 Ontologies

The core AdvoCATE assurance model can be extended by user-defined ontologies. This enhances our model-based approach to assurance by 1) formulating domain-specific extensions to the underlying models, and 2) querying the resulting extended models. Our goal in integrating ontologies into model-based assurance is to provide the benefits of formalism while retaining the key communicative purpose of DACs, without sacrificing their comprehensibility. By mapping DAC components to a domain-specific ontology we facilitate DAC validation, and by applying domain- and stakeholder-specific queries to core DAC components that have been semantically enriched using ontologies, we provide additional stakeholder insights.

General purpose query languages for assurance arguments and associated models have been investigated, as have languages more targeted at assurance arguments, though neither has exploited integrations with ontologies. Ontologies have been widely used in requirements development, though less so for ACs. As such, so far as we are aware, ontology-integrated model-based DACs represent a novel extension to the state of the art and prevailing practice of DAC development.

Each of the DAC components has a model-based representation, which the tool user interface displays using a variety of formats—each component has a domain specific language (DSL), and some also have tabular or graphical representations. DSLs are built with Xtext, tables with NatTable, and graphical diagrams using Sirius. We refer to the collection of interrelated models as the integrated assurance case model, or simply AC model. We employ model transformations to generate artifacts from the AC model, in particular, assurance arguments.

Though formal approaches have been taken to the construction of ACs, either incorporating formal reasoning or integrating with external models with formal semantics, this introduces a tension with one of the fundamental purposes of ACs: to communicate and convince. We believe ontology-backed ACs can provide the advantages of both informal and formal approaches.

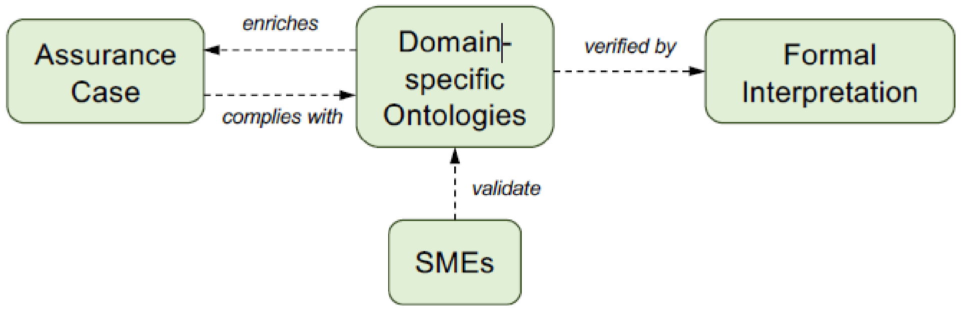


Figure 76. Ontology-backed Assurance Case Concept

Figure 76 illustrates how this works - the AC model is embedded in a user-extensible ontology that contains information from the assurance case, which can then be extended with domain-specific concepts. The ontology can be validated by subject matter experts (SMEs) and serves as a semi-formal specification of the domain that can, optionally, be mapped to a formal semantics for verification. Elements can, in turn, be used to construct parts of the assurance case through the use of an ontology-backed structured language.

The ontology provides a vocabulary for, for example, claims of the assurance arguments. Well-formedness of claims and soundness of some forms of reasoning can be determined by the ontology. It also provides a vocabulary for domain-specific queries that are also used in patterns to generate arguments. We will focus on the queries and patterns here.

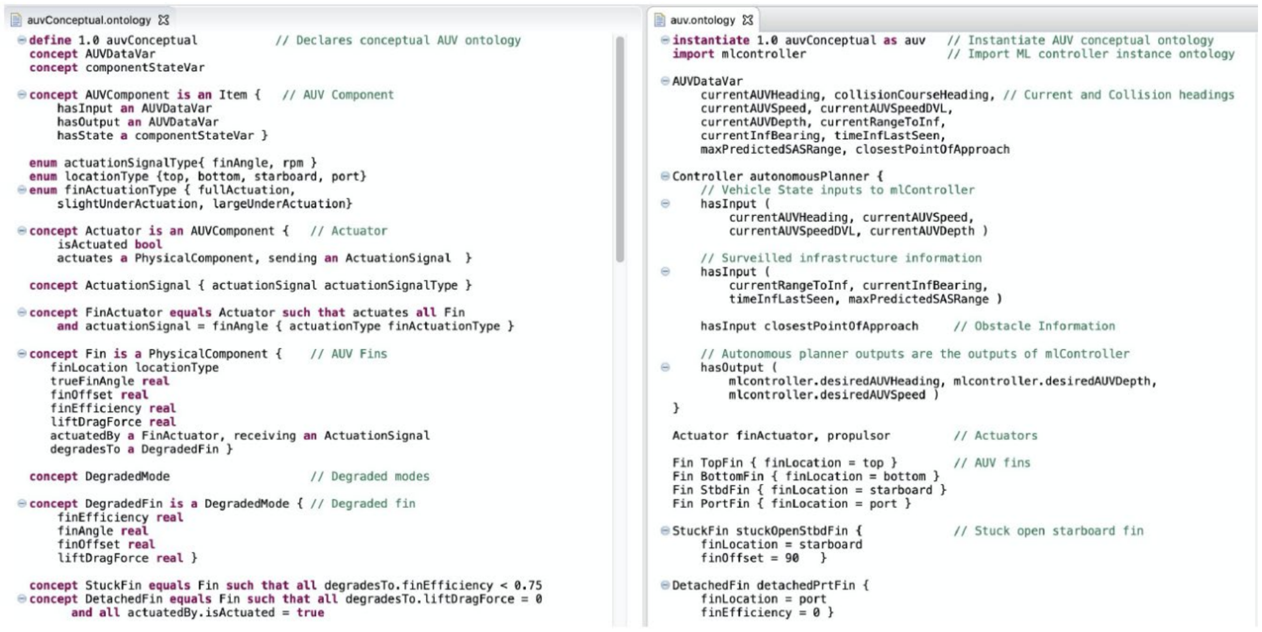
We map elements of the AC model to concepts, relations, and their instances in a derived ontology that is user-extensible. This ontology, itself, then forms part of an extended model. Since the ontologies are, in effect, also part of our model, we will sometimes use core model to refer to the non-ontological part. For example, AdvoCATE includes a simple notion of physical architecture, consisting of a hierarchy of components. In the ontology, we represent this by a concept `Component`, whose instances are the actual components of a given system. A relation `subComponent` represents the containment relation of the architecture. The user can then define new concepts and relations to enrich the model, such as concepts for component input and output, and relations for connections.

Figure 77 illustrates some features of our ontology definition language, which has an object-oriented flavor and is reasonably verbose. We distinguish conceptual and instance ontologies, where the former defines concepts and their relations, and the latter instantiates them. Concept declarations optionally give super-concepts, attributes, and relations to other concepts. Attributes have types (primitive, enumerated, list, record, and any combination).

In the AUV conceptual ontology example (Figure 77, left) the concept Actuator is a sub-concept of `AUVComponent`, with the boolean attribute `isActuated` and the relations `actuates` and `sending`, to the concepts `PhysicalComponent` and `ActuationSignal`, respectively. We can also define concepts from other concepts using union, intersection, negation, and quantification along relations. We can lift attributes from the target concept of a relation to the relation, itself. Here, a `DetachedFin` is defined to be a Fin such that every `DegradedFin` it `degradesTo` has no (zero) `liftDragEfficiency` despite being `actuatedBy` every `FinActuator` that `isActuated`.

4.4.5 Queries

Figure 78 shows the grammar used to construct queries over the assurance model. We have defined our own languages for queries and ontologies, rather than use existing languages such as the Web Ontology Language (OWL) and SPARQL because it enables a tighter integration with our core model and a similar style of DSL.



```
define 1.0 auvConceptual           // Declares conceptual AUV ontology
concept AUVDataVar
concept componentStateVar

concept AUVComponent is an Item {   // AUV Component
    hasInput an AUVDataVar
    hasOutput an AUVDataVar
    hasState a componentStateVar }

enum actuationSignalType{ finAngle, rpm }
enum locationType {top, bottom, starboard, port}
enum finActuationType { fullActuation,
    slightUnderActuation, largeUnderActuation}

concept Actuator is an AUVComponent {   // Actuator
    isActuated bool
    actuates a PhysicalComponent, sending an ActuationSignal  }

concept ActuationSignal { actuationSignal actuationSignalType }

concept FinActuator equals Actuator such that actuates all Fin
    and actuationSignal = finAngle { actuationType finActuationType }

concept Fin is a PhysicalComponent {    // AUV Fins
   finLocation locationType
   trueFinAngle real
   finOffset real
   finEfficiency real
   liftDragForce real
   actuatedBy a FinActuator, receiving an ActuationSignal
   degradesTo a DegradedFin }

concept DegradedMode                    // Degraded modes

concept DegradedFin is a DegradedMode { // Degraded fin
    finEfficiency real
    finAngle real
    finOffset real
    liftDragForce real }

concept StuckFin equals Fin such that all degradesTo.finEfficiency < 0.75
concept DetachedFin equals Fin such that all degradesTo.liftDragForce = 0
      and all actuatedBy.isActuated = true
```

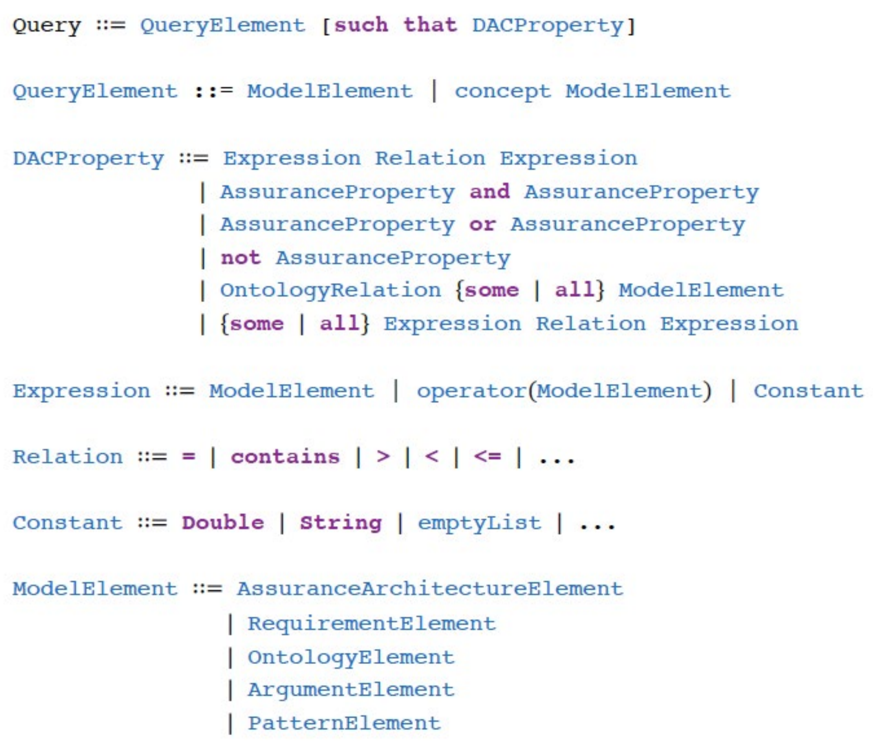
```
instantiate 1.0 auvConceptual as auv   // Instantiate AUV conceptual ontology
import mlcontroller                     // Import ML controller instance ontology

AUVDataVar
    currentAUVHeading, collisionCourseHeading, // Current and Collision headings
    currentAUVSpeed, currentAUVSpeedDVL,
    currentAUVDepth, currentRangeToInf,
    currentInfBearing, timeInfLastSeen,
    maxPredictedSASRange, closestPointOfApproach

Controller autonomousPlanner {
    // Vehicle State inputs to mlController
    hasInput (
        currentAUVHeading, currentAUVSpeed,
        currentAUVSpeedDVL, currentAUVDepth )

    // Surveilled infrastructure information
    hasInput (
        currentRangeToInf, currentInfBearing,
        timeInfLastSeen, maxPredictedSASRange )

    hasInput closestPointOfApproach     // Obstacle Information

    // Autonomous planner outputs are the outputs of mlController
    hasOutput (
        mlcontroller.desiredAUVHeading, mlcontroller.desiredAUVDepth,
        mlcontroller.desiredAUVSpeed )
}

Actuator finActuator, propulsor         // Actuators

Fin TopFin { finLocation = top }        // AUV fins
Fin BottomFin { finLocation = bottom }
Fin StbdFin { finLocation = starboard }
Fin PortFin { finLocation = port }

StuckFin stuckOpenStbdFin {             // Stuck open starboard fin
    finLocation = starboard
    finOffset = 90   }

DetachedFin detachedPrtFin {
    finLocation = port
    finEfficiency = 0 }
```

Figure 77. Ontology Definition Language Features, Conceptual and Instance Ontology

Figure 79 shows three example queries over the integrated AC model. These show a range of examples querying arguments, requirements, and functions, based on properties of associated data, such as description, evidence, mitigation, and allocation. Quantifiers can be used (e.g., to express that all or some elements have some property) and nested, as well as metrics (e.g., to express that the number of associated artifacts with some property is bounded). We can query both the model and the embedding of the model in the ontology. To do the latter, we use the keyword `concept`.

Figure 80 shows a query for goal nodes of arguments in the AC, that contain claims referring to the reinforcement learning controller, and that are eventually supported (i.e., `followed`) by at least one solution node that is related to verification evidence. Here, `eventually` is used to form the reflexive transitive closure of a relation. The second query, in Figure 80, looks for requirements allocated to the autonomous planner (`autonomousPlanner`), and that represent the requirements to implement the mitigations of hazards that are, in turn, allocated to the AUV fins (`Fin`) and whose hazard condition involves either a stuck open starboard fin (`stuckOpenStbdFin`) or detached port fin (`detachedPrtFin`). These items correspond to the concepts and instances defined in the corresponding ontologies (Figure 77).

Note, here, that we use "allocation" in two distinct ways: in the first part of the query, it refers to a requirements allocation, which is a responsibility assignment of the requirement to, say, a component in the physical decomposition model, also reflected as an instance in the instance ontology. In the second part of the query, it refers to a hazard allocation, that is, the location of the hazard.



```
Query ::= QueryElement [such that DACProperty]

QueryElement ::= ModelElement | concept ModelElement

DACProperty ::= Expression Relation Expression
              | AssuranceProperty and AssuranceProperty
              | AssuranceProperty or AssuranceProperty
              | not AssuranceProperty
              | OntologyRelation {some | all} ModelElement
              | {some | all} Expression Relation Expression

Expression ::= ModelElement | operator(ModelElement) | Constant

Relation ::= = | contains | > | < | <= | ...

Constant ::= Double | String | emptyList | ...

ModelElement ::= AssuranceArchitectureElement
               | RequirementElement
               | OntologyElement
               | ArgumentElement
               | PatternElement
```

Figure 78. Query Grammar

Goals in arguments referring to the control LEC and eventually followed by at least one solution node related to verification evidence	ArgumentNode such that type = goal and description = "*controlLEC*" and eventually SupportedBy some (ArgumentNode such that type = solution and some relatedEvidence.type = verification)
Requirements allocated to the Backseat Driver, which are also requirements for mitigating hazards due to the detached AUV fin degradation mode	Requirement such that allocation contains BackSeatDriver and isRequirement for some Mitigation for some (Hazard such that allocation contains Fin and condition contains detachedPrtFin))
Functions allocated to a component containing the control LEC and that are affected by any disturbance events	Function such that allocatedTo Component such that contains (ControlLEC affectedBy some DisturbanceEvent)

Figure 79. Example Queries

```
ArgumentNode such that type = goal
  and description = "*rlController*" and
    eventually SupportedBy some (ArgumentNode
      such that type = solution and
        some relatedEvidence.type = verification)
```

```
Requirement such that allocation contains autonomousPlanner
  and isRequirement for some Mitigation for some
    (Hazard such that allocation contains Fin and
      (condition contains stuckOpenStbdFin or
         condition contains detachedPrtFin))
```

Figure 80. Example Queries from the AUV Assurance Case

4.4.6 Dynamic Arguments

Although we have investigated the notion of dynamically updating arguments in earlier work [DHP2015], we now believe that it is more useful to update argument status in real-time, while maintaining argument structure. Status can be used for different purposes, such as showing which part of an argument is currently "active" (that is, which branches reason about the effectiveness of the safety measures currently being applied). It can also be used to display confidence in argument claims.

To that end, we extended the argument view mechanism that was developed in earlier phases of the project to be able to access real-time values provided to AdvoCATE via external ports. The views are updated as data is received. By integrating this with the output of assurance measures, we are able to use these dynamic argument views to visualize LEC confidence and its bearing on an argument.

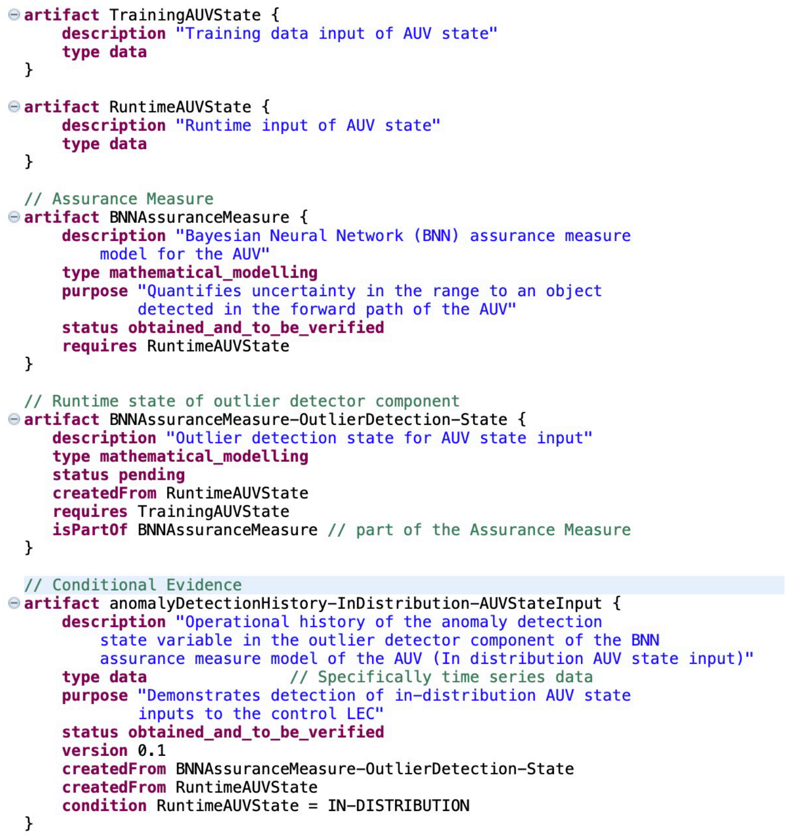
Dynamic views are specified by associating formulas with selected argument nodes, and mapping these formulas to visualizable node properties, such as colors. Node status can be defined in terms of the values of any model artifacts that can be queried, so in particular, we can use all those argument nodes which are linked to the node by an `inSupportedBy` link. In other words, we can define the status of a node as a function of the status of its subclaims. This gives us, in effect, an inference rule for propagating confidence through an assurance argument. Rather than being hard-coded in the logic, the property language flexibly allows for different propagation rules. Figure 81 shows an excerpt of the dynamic property syntax, which extends the query language.

```
Condition := Expression Operator Expression
           | not Condition
           | Condition {and | or} Condition
           | Condition for n [of m]
```

Figure 81. Dynamic Properties Grammar (Excerpt)

The `for` syntax is used to state that a property must hold for a certain number of the most recent (discrete) time steps. For example, to state that a monitored signal *x* has been received for the last three time steps, we would use `%x% != null for 3`. Here, the `%x%` syntax is used to designate external variables. To state that the AUV control LEC inputs for the current time step are out of distribution (defined in terms of the state of a random variable in the anomaly detection component of assurance measure being less than some threshold), we simply use `%non_conf% < 20`.

We can also associate dynamic formulas with evidence artifacts. This allows us to specify conditional evidence, where the formula characterizes the validity of the evidence. Figure 82 shows a fragment of an evidence log DSL, with conditional evidence. Figure 83 shows a view of the evidence model, including conditions. (For another example, see Figure 20).



```
artifact TrainingAUVState {
    description "Training data input of AUV state"
    type data
}

artifact RuntimeAUVState {
    description "Runtime input of AUV state"
    type data
}

// Assurance Measure
artifact BNNAssuranceMeasure {
    description "Bayesian Neural Network (BNN) assurance measure
        model for the AUV"
    type mathematical_modelling
    purpose "Quantifies uncertainty in the range to an object
            detected in the forward path of the AUV"
    status obtained_and_to_be_verified
    requires RuntimeAUVState
}

// Runtime state of outlier detector component
artifact BNNAssuranceMeasure-OutlierDetection-State {
   description "Outlier detection state for AUV state input"
   type mathematical_modelling
   status pending
   createdFrom RuntimeAUVState
   requires TrainingAUVState
   isPartOf BNNAssuranceMeasure // part of the Assurance Measure
}

// Conditional Evidence
artifact anomalyDetectionHistory-InDistribution-AUVStateInput {
    description "Operational history of the anomaly detection
        state variable in the outlier detector component of the BNN
        assurance measure model of the AUV (In distribution AUV state input)"
    type data            // Specifically time series data
    purpose "Demonstrates detection of in-distribution AUV state
            inputs to the control LEC"
    status obtained_and_to_be_verified
    version 0.1
    createdFrom BNNAssuranceMeasure-OutlierDetection-State
    createdFrom RuntimeAUVState
    condition RuntimeAUVState = IN-DISTRIBUTION
}
```

Figure 82. Evidence Log with Conditional Evidence

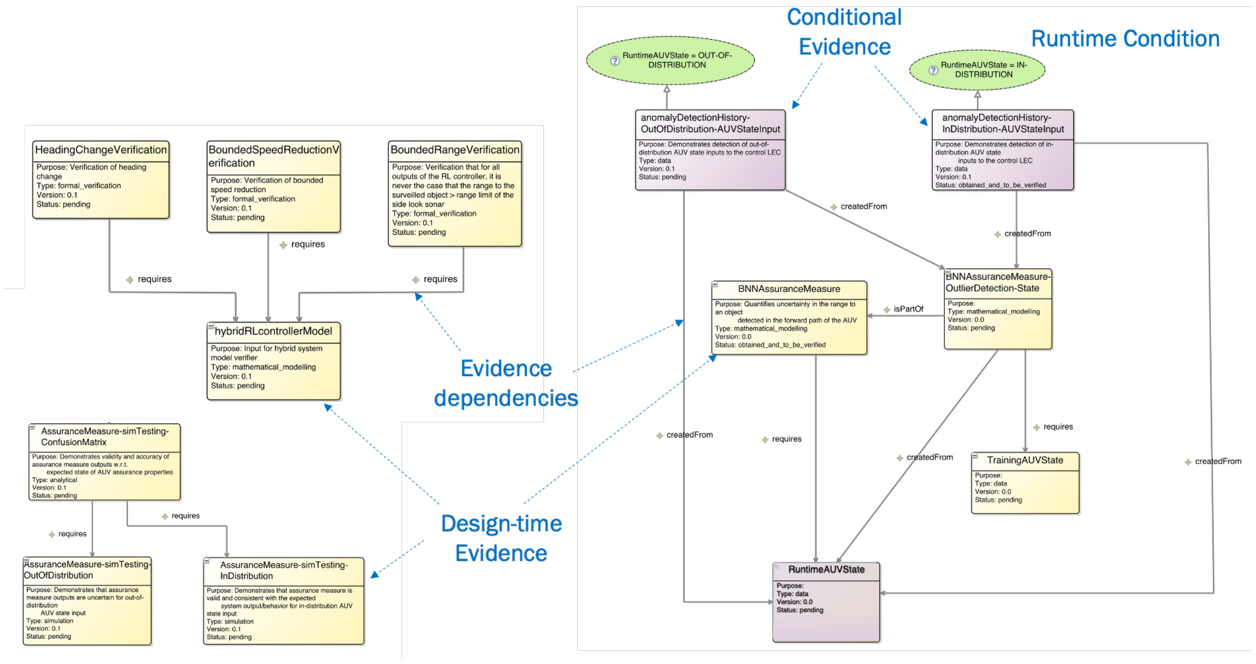

Conditional Evidence
Runtime Condition
Evidence dependencies
Design-time Evidence
RuntimeAUVState = OUT-OF-DISTRIBUTION
RuntimeAUVState = IN-DISTRIBUTION
anomalyDetectionHistory-OutOfDistribution-AUVStateInput
anomalyDetectionHistory-InDistribution-AUVStateInput
HeadingChangeVerification
BoundedSpeedReductionVerification
BoundedRangeVerification
hybridRLcontrollerModel
AssuranceMeasure-simTesting-ConfusionMatrix
AssuranceMeasure-simTesting-OutOfDistribution
AssuranceMeasure-simTesting-InDistribution
BNNAssuranceMeasure
BNNAssuranceMeasure-OutlierDetection-State
TrainingAUVState
RuntimeAUVState
requires
createdFrom
isPartOf

Figure 83. Evidence Diagram with Conditional Evidence

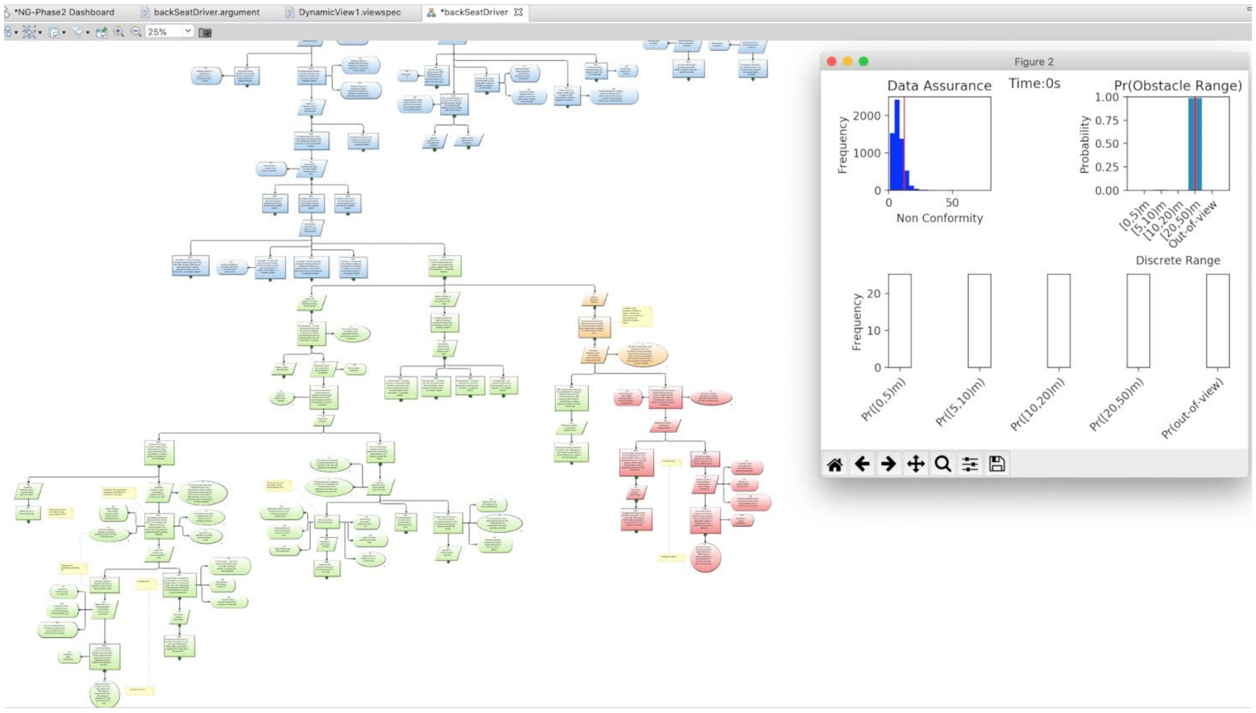


Figure 84. Dynamic Argument View

Figure 84 shows a bird's eye view of an argument whose status is shown by coloring the nodes. In particular, nodes that are green in color are those for which *all* associated conditions—specified using the dynamic properties grammar (Figure 81)—evaluate to TRUE. Likewise, nodes that are red in color have all their associated conditions evaluate to FALSE. Nodes that are orange in color have two or more of their conditions evaluate to FALSE. In the example in Figure 84, the variables invoked in the applicable conditions receive their values through a simulation (shown in the window to the right of the argument structure).

Effectively, the dynamic argument view shows which prong of the argument is being exercised by the simulation, and its status as a function of the attached conditions and the associated variables.

4.4.7 Integrating Evidence and Tools

Evidence Model

Evidence is used in various places in the assurance case, principally to substantiate solution nodes in arguments and to justify data in the risk analysis. Here we use "evidence" in a general sense to encompass all external artifacts which are linked to the assurance case in a justifying or contextual role; for example, verification artifacts such as test data and simulation results; manufacturer data sheets, formal specifications, and models; documentation such as user guides, test plans, concepts of operations, etc.

The evidence log records all the evidence used in the assurance case, as well as any evidence that it is planned to use in future. Evidence can be designated as pending, to indicate that it is anticipated that it will be received, while the assurance case is still being developed.

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 (evidence assertions), and then work back to determine how the evidence was created, and which assumptions it relies on. If an evidence artifact is self-contained, then it will have no dependencies on other evidence. But often 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.

Tools Log

The tools log is used to record external tools which are used to create evidence artifacts, in this case the simulator. The tools and evidence logs are interrelated and capture the chain of dependencies through which evidence is constructed, along with the supporting assumptions. Tools are characterized in terms of their inputs and outputs, each of which is either an evidence artifact or a primitive type.

A tool specification gives assumptions on the inputs and guarantees on the outputs. This is especially useful if a tool is used more than once. A tool use represents an application of the tool to concrete evidence artifacts. Next, create a pattern for the tool. Then, by applying the pattern to a tool use, we can create an argument fragment which reasons about the properties of the tool outputs (the tool guarantees providing the corresponding evidence assertions). Just as tools can be recursively chained together, their patterns can be similarly composed to construct an argument that reasons over the construction of the evidence that is ultimately constructed.

5.0 CONCLUSIONS

We have described work carried out in the QUASAR project, giving an account of the platform-specific DACs created for challenge problems in the air domain and undersea domain platforms. We also described the assurance measures and how they were applied to the challenge problem demonstrations. Lastly, we described advances in assurance case technology that were implemented as AdvoCATE tool extensions.

We now outline several areas where we believe further work would be useful to build upon the advances made in the QUASAR project.

5.1 Platform-Specific Dynamic Assurance Cases

The DACs comprised core assurance components including: i) hazard analyses and mitigation requirements captured via hazard and requirements logs respectively; ii) safety architecture models describing the organization of the safety risk mitigations in terms of scenarios that clarify the event sequences in which mitigations are to be invoked; iii) assurance rationale captured in the form of structured arguments; and iv) an evidence model that records a variety of forms of evidence necessary to support the assurance claims made.

The focus of the DACs was on LECs, more specifically the underlying NN models. Some of the key avenues to advance the maturity of the DACs include:

- Definition and assurance of the properties of the data used to develop (train, validate, and test) the LECs. This includes characterization of the ODDs, defining how data is to be sampled from the ODDs so that LEC assurance properties and system safety objectives are met.
- Refining safety and assurance objectives allocated from a system-level to the LEC, and translating those into quantitative performance metrics applicable to the LEC and the underlying models.
- Translating LEC assurance objectives into criteria concerning coverage of the ODD by the LEC and its implementation.
- Providing a rigorous basis for the assurance case architecture and the underlying argument architecture (both of which have been considered here, albeit informally).
- Addressing assurance case update during design time in conjunction with LEC model updates so that the assurance case is consistent with model development iterations. The idea is to have assurance concerns be an additional parameter considered during LEC development, in particular model training and learning performance assessment.

5.2 Tool Support

5.2.1 Tool Workflows

We have described our approach for incorporating results from formal verification tools in assurance cases. To that end, we have developed an approach for capturing the verification method workflow in terms of tool specifications that describe how the different tools fit together and what are the different assurance constraints on the tool usage for their results to be deemed valid. We use assumption and guarantee statements for each tool to capture the different assurance constraints over the tool's inputs and outputs which are, in turn, modeled as evidence artifacts.

We distinguish between tool specifications, which represent the reusable information about a tool, and tool use specifications, which capture the information specific to each tool use. While the tool specifications are parametrized, tool use specifications represent instances of those parametrized specifications. To integrate the tool information into assurance arguments, we have developed a default argument pattern that can be semi-automatically instantiated from the tool use information. We also allow for customization of the pattern, so that each tool can have its own tailored argument pattern. In the case of tool chains, pattern instantiation can be applied recursively, which enables automatic generation of an end-to-end tool integration argument.

While we initially focused the tool specification to be strictly about tools, we discovered that some crucial steps in verification method workflows are not performed by tools, but are actually carried out manually. Although we have used the current syntax to capture both kinds of verification steps, extensions are needed to explicitly capture this distinction. Moreover, by examining the different verification method workflows used in the project, we identified numerous iterative methods, which require greater support for different kinds of loops. However, it is not just that we need to represent loops between tools, but we need to make usage constraints over those loops, e.g., to maintain consistency of the verification method application.

5.2.2 Active Integration of Assurance Case and Verification Tools

We examined the different formal tools used in the project for verification of neural networks and explored the assurance challenges associated with those tools. We have reported here some of our results with Venus and VerifAI, but we have also examined application of the Verisig verification tool. In each tool application we examined, we identified some gaps in their application that could affect their integration in an assurance case. The assurance challenges were mainly related to whether or not the tool is appropriate for the given verification problem, the inputs to the verification are adequate, and what the results could be used to support.

To try and address some of these challenges, we used our workflow approach to capture all those different assurance constraints to guide the user to properly apply the tool and to appropriately use the results in an assurance case. However, tool usage and tool use specifications require manual effort. Towards facilitating automation, we have set the foundations with the workflows approach for tool use specification by extending AdvoCATE with active integrations for different tools. To that end, we have explored active integrations with Venus and Verisig, such that they can be invoked directly from within AdvoCATE, and their results automatically captured in the form of tool uses and evidence. This automation could also help us to automatically check validity of the different tool constraints to make sure that the results obtained from these tool applications can be safely used in an assurance case.

5.2.3 Generation of Assurance Measures

Implementing the assurance measures required the creation of several scripts and models, integrated with the structure of the overall assurance cases, and the training of those models. The architectural design of the measures required a significant amount of domain expertise, and experimentation to determine the most effective design for each challenge problem. Thus, this remains a time-consuming, labor- and expertise-intensive part of the overall assurance process.

In order to alleviate these issues, we have been working towards the creation of a high-level DSL for specifying the structure of the assurance measure, extracting the relevant information from other parts of the assurance case, and generating the corresponding low-level scripts, in effect, allowing compilation for a range of target platforms, while allowing end-users to choose between different (pre-coded) design options. The key elements of the DSL include:

1. Mission objectives
2. System measurements, and their relationship to the safety architecture
3. Model structure for high and low-level assurance measures, and their architecture
4. Thresholds for various system measurements.

The mission objectives identify specific outcomes required for mission success, such as avoiding collisions. Each mission objective is linked to a system measurement that is used to determine the success or failure of the system in accomplishing the mission objective. In general, system measurements are any signals that we can monitor from the system, including outputs of various system components, such as FLS pings and HSD commands as well as outputs of LEC models. System measurements and mission objectives are both linked to events from the safety architecture. This allows us to determine dependencies for the overall assurance measure structure. System measurements linked to events indicate that the event can be measured by some manipulation/computation using the related system measurements. This gives us the options for inputs to our low-level assurance measures. The combination of mission objective, its related system measurement, and its related events provide the information needed for the assurance measures.

For the high-level assurance measure, we can create a Bayesian network based on the dependencies of events from each mission objective. The low-level assurance measures compute confidence in the success of the system meeting a mission objective. We then use a threshold for the confidence to determine success or failure. Inputs for the low-level measures can include any system measure related to any event linked to the mission objective. The output is defined by the mission objective's identified system measurement.

5.3 Methodology

We have developed a generic methodology for constructing assurance cases that starts with hazard and risk analysis, is followed by design of the safety system and requirements formulation, then creation of rationale with assurance arguments, and integrates substantiating evidence. Although we have applied and refined this methodology in the context of the challenge problems working with our TA4 partners, it remains to create an assurance case methodology that is customized to LECs. As part of this, we would also systematize the approach to creating (dynamic) assurance measures that are aligned with the (static) assurance case [ADP2020].

When sending assurance cases to our TA4 partners, we have found that a significant amount of effort is expended (on the TA3 side) in explaining the structure of the assurance case, (internally for TA3, but presumably also on the TA4 side) in formulating review comments and, in turn (TA3) in addressing those comments, iterating on the assurance case, and tying the changes in the new assurance case back to the review comments and the previous iteration of the assurance case. This currently manual process could be supported by developing and integrating a review and assessment workflow into the assurance case methodology and tool framework.

6.0 REFERENCES

[ADP2019] Asaadi, E., Denney, E., and Pai, G., "Towards Quantification of Assurance for Learning-enabled Components", *Proceedings of the 2019 European Dependable Computing Conference*, Sep. 2019, pp. 55-62.

[ADP2020] Asaadi, E., Denney, E., Henderson, R., Menzies, J., Pai, G., and Petroff, D.,"Dynamic Assurance Cases: A Pathway to Trusted Autonomy", **53**(12), Dec. 2020, pp. 35-46.

[ASSR2020] Amini, A., Schwarting, W., Soleimany, A., and Rus, D., "Deep evidential regression", *Advances in Neural Information Processing Systems*, **33**, 2020, pp. 14927-14937.

[ASTM2021] Subcommittee F38.01 on Airworthiness, "Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions", *ASTM-F3269-21*, ASTM International, West Conshohocken, PA, 2017.

[CDP2017] Clothier, R., Denney, E., and Pai, G., "Making a Risk Informed Safety Case for Small Unmanned Aircraft System Operations," *17th AIAA Aviation Technology, Integration, and Operations Conference* (ATIO 2017), AIAA Aviation Forum, June 2017.

[DHP2015] Denney, E., Habli, I., and Pai. G., "Dynamic Safety Cases for Through-Life Safety Assurance", *37th International Conference on Software Engineering* (ICSE 2015) – *New Ideas and Emerging Results*, May 2015.

[DP2018] Denney, E., and Pai, G., "Tool Support for Assurance Case Development," Automated Software Engineering Journal, **25**(3), Sep. 2018, pp. 435-499.

[DPW2019] Denney, E., Pai, G., and Whiteside, I., "The Role of Safety Architectures in Aviation Safety Cases," *Reliability Engineering and System Safety Journal*, **191**, 2019.

[GSN2021] Assurance Case Working Group, "Goal Structuring Notation Community Standard Version 3," *SCSC-141C*, Safety Critical Systems Club, May 2021.

[RFB2015] Ronneberger, O., Fischer, P., Brox, T. "U-Net: Convolutional Networks for Biomedical Image Segmentation" *Proceedings of the 18th International Conference on Medical Image Computing and Computer-Assisted Intervention*, Oct. 2015.

[VAI2019] Dreossi, T., Fremont, D., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., and Seshia, S., "VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems", *31st International Conference on Computer Aided Verification (CAV)*, July 2019.

[VAI2020] Kim, E., Gopinath, D., Pasareanu, C., and Seshia, S. A, "A Programmatic and Semantic Approach to Explaining and Debugging Neural Network Based Object Detectors", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 11128-11137.

[VAI2021] Torfah, H., Junges, S., Fremont, D., and Seshia, S., "Formal Analysis of AI-based Autonomy: From Modeling to Runtime Assurance", *Proceedings of 21st International Conference on Runtime Verification* (RV 2021)*,* Oct. 2021*,* pp. 311–330.

[VEN2020] Botoeva, E., Panagiotis K., Kronqvist, J., Lomuscio, A, and Misener, R., "Efficient verification of relu-based neural networks via dependency analysis", *Proceedings of the AAAI Conference on Artificial Intelligence*, **34**(4), 2020, pp. 3291-3299.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AAM	Assurance Architecture Model
AC	Assurance Case
APM	Assurance Policy Model
AQM	Assurance Quantification Model
AUC	Area Under the Curve
AUV	Autonomous Underwater Vehicle
AVL	Autonomous Visual Landing
BHA	Barrier Hazard Analysis
BNN	Bayesian Neural Network
BTD	Bow Tie Diagram
CES	Controlled Event Structure
CFIT	Controlled Flight Into Terrain
CNN	Convolutional Neural Network
CONOPS	Concept of Operations
CP	Challenge Problem
CPA	Closest Point of Approach
CPS	Cyber Physical System
CTE	Cross-track Error
DAC	Dynamic Assurance Case
DH	Decision Height
DNN	Deep Neural Network
DOF	Degrees of Freedom
DSL	Domain Specific Language
EC	Environmental Condition
FLS	Forward Looking Sonar
GSN	Goal Structuring Notation
GUI	Graphical User Interface
HA	Hazardous Activity
HAT	Height Above Touchdown
HSD	Heading, Speed and Depth
ILS	Instrument Landing System

LEC	Learning Enabled Component
LE-CPS	Learning Enabled Cyber Physical System
LSTM	Long Short-Term Memory
MAP	Mean Average Precision
MILP	Mixed Integer Linear Programming
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology
MTL	Metric Temporal Logic
NG	Northrop Grumman
NN	Neural Network
OKS	Object Keypoint Similarity
ODD	Operational Design Domain
OOD	Out-of-Distribution
OWL	Web Ontology Language
ReLU	Rectified Linear Unit
RV	Random Variable
SME	Subject Matter Expert
SPARQL	SPARQL Protocol And RDF Query Language
SS	System State
TA	Technical Area
TLOS	Target Level of Safety
UQ	Uncertainty Quantification
VFR	Visual Flight Rules
VMC	Visual Meteorological Conditions
WCET	Worst Case Execution Time
XML	Extensible Markup Language