Heterogeneous Aviation Safety Cases: Integrating the Formal and the Non-formal

Ewen Denney, Ganesh Pai and Josef Pohl
SGT / NASA Ames Research Center
Moffett Field, CA 94035, USA
Email: {ewen.denney, ganesh.pai, josef.pohl}@nasa.gov

***Abstract*—We describe a method for the automatic assembly of aviation safety cases by combining auto-generated argument fragments derived from the application of a formal method to software, with manually created argument fragments derived from system safety analysis. Our approach emphasizes the heterogeneity of safety-relevant information and we show how such diverse content can be integrated into a single safety case. We illustrate our approach by applying it to an experimental Unmanned Aircraft System (UAS).**

***Keywords*-Safety, Safety cases, Automation, Aviation, Heterogeneity, Unmanned Aircraft, Formal Methods.**

I. Introduction

Safety cases are among the state of the art in technologies for safety management, and their development has been a common practice for the certification of safety-critical systems in the nuclear, defense and rail domains [1]. The requirement for a safety case has also been considered in emerging international standards and national guidelines [2], [3]. The move towards safety cases represents a departure from highly prescriptive safety standards, where certification, particularly of software-intensive systems, is often obtained by compliance with predefined objectives and processes.

A safety case is "a structured argument supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment" [4]. In brief, it comprises a set of safety claims (goals), supporting evidence (solutions), an argument explaining what the evidence is, why the evidence supports the claims together with the assumptions made, and the context in which the claims, the argument, and the evidence are to be considered. Safety cases are typically constructed manually; this is time consuming, expensive, and also quickly becomes unmanageable during iterative systems and software development. Safety cases that reason about software details specifically, are likely to grow super-linearly in the size of the underlying software due to the increased number of requirements that software must satisfy.

This problem is compounded when considering the prevalence and importance of domain-specific content, e.g., for aviation systems, the size of a safety case, the diversity of its content, and the level of assurance provided by such diverse content especially pose challenges: aviation safety cases need to reconcile heterogeneous content, including elements from the system design, e.g., physical formulae, system operation, e.g., maintenance procedures, and software to allow a comprehensive safety assessment. For software, the heterogeneity of evidence, context and assumptions is especially evident when considering sources such as simulation runs, unit tests, results of model-checking, proofs of correctness, and the variety of tools used to generate them.

Furthermore, each of these sources provides a different level of assurance. This affects the trustworthiness of the evidence in the safety case and, in turn, the confidence that can be placed in the the overall argument. Evidence from formal verification, such as proofs of correctness, raises the level of assurance that can be claimed for mathematical and safety-critical software, typical in aviation systems. Indeed, proofs are acknowledged to provide the highest level of such assurance, compared with "non-formal" approaches.

To address these challenges of managing complexity, heterogeneity, and improving trustworthiness, there is a need for mechanisms for (a) the automated creation and assembly of evidence-based safety arguments, and (b) improving the confidence that can be placed in those arguments, by integrating formal and non-formal reasoning. This paper describes our method for semi-automatically creating and assembling heterogeneous safety cases, to include arguments from both formal and non-formal reasoning, with application to a real aviation system. The paper makes the following contributions:

(1) A methodology is given for semi-automatically creating a meticulous, end-to-end safety argument (Section III). We illustrate its application on a real, experimental unmanned aircraft system (Section II), and give example fragments of an interim safety case (Section IV). To our knowledge, few, if any, such examples exist in practice or in the literature, where a safety argument is developed starting from the level of the system down to low-level software implementation details.

(2) Our approach combines traditional safety analysis techniques with formal methods: we auto-generate safety argument fragments for the software components of the example system from formal proofs of correctness. Then, we automatically assemble these with the manually created safety arguments, obtained from traditional safety analysis, which are applicable to the wider system context.

(3) We characterize the heterogeneity inherent in arguing

safety by identifying a diverse set of elements that contributes to a comprehensive safety argument (relevant for aviation in general and to the example system in particular). We also illustrate how some of these are combined into a safety case fragment for the example system.

II. Illustrative Example

A. Target System Description

Our target system is the Swift *Unmanned Aircraft System* (UAS) from NASA Ames. The UAS comprises a single airborne system: the electric Swift Unmanned Aerial Vehicle (UAV), a primary and secondary ground control station (GCS) and communication links.

The UAV can be controlled on the ground by a pilot, or can fly autonomously by following a pre-programmed or uploaded nominal flight plan (i.e., a mission): a sequence of commands (determining a set of waypoints) from takeoff to landing. The pilot might take-off, land, or intercept at any time. The off-nominal plan describes the actions of the Contingency Management System (CMS), the failsafe trajectory, the procedures to be followed on the ground, and so on. The CMS is required is to keep the aircraft on range, so that if any crash occurs, it happens "inside the box".

The UAS can be operated in a semi-autonomous manner, i.e., in the *pilot in control* (PIC) and *computer in control* (CIC) modes. The ground station is operated by a GCS Operator (GCSO) who calls out important state information (e.g., the true airspeed) to the pilot. There may also be a secondary (or research) pilot, who controls the aircraft in the *secondary pilot in control* (SIC) mode. The GCSO can transmit and upload commands to the UAV via the GCS, and the pilot can control the UAV using a transmitter with a joystick and trim tab. The primary pilot always instigates a change of control via a safety switch.

B. System and Software Architecture

The overall system architecture is layered, with several loosely-coupled *modules* implemented on the *Reflection Virtual Machine* (RVM): a multi-component, event-driven, real-time, configurable software framework. The architecture consists of several execution layers: the base layer is the Windows XP Embedded operating system running on the UAV hardware. The CGL physics library runs on top of this, followed by the RVM. Finally, the flight software (FSW) is implemented on top of the RVM as a collection of interconnected modules (including the autopilot itself), and script files which describe specific mission configurations, such as flight plans and specific parameters.

The two main modules of the autopilot are the flight management system (`FMS`) and the controller (`AP`). Each plays a role in controlling the aircraft control surfaces and, in turn, three dimensional aircraft movement; namely forward motion and rotation around the lateral (pitch), longitudinal (roll), and vertical (yaw) axes. The *aileron* affects roll, *elevators* (and throttle/speed control) affect pitch, while the *rudder* affects yaw.

COMMAND_	FMSLATMODE_	FMSLONMODE_
STOP	DISENGAGED	DISENGAGED
FLYTODIRECT	FLYTOWAYPOINT	ALTITUDE_ATTAIN
FLYTOTRACK	TRACKTOWAYPOINT	ALTITUDE_HOLD
CIRCLE	CIRCLE	TO_ACCEL2VROT
TAKEOFF	TO_ACCEL2VROT	TO_FULLCLIMB
APPROACH	WINGSLEVEL	GLIDE
LAND	FLARE	FLARE
INVALID		TAXISTOP

Figure 1. FMS commands and modes.

C. Example Operation and Control

A flight plan consists of a sequence of commands (Figure 1), to the UAV. Figure 2 depicts a landing profile, and the transitions therein, when a `LAND` command is issued. Based on the current state and the command being evaluated, the control system periodically updates the control surface positions, the calculations for which are phased in two "directional" modes: longitudinal and lateral. Each mode has several cases of relevant calculations, e.g., the `LAND` command invokes only some of the cases shown under the lateral (`FMSLATMODE`) and longitudinal (`FMSLONMODE`) modes (Figure 1). During landing, if the current state (phase) is "Descent", only the `TRACKTOWAYPOINT` lateral mode will be considered by the `FMS` with respect to the aileron output control calculation. The transition criteria are defined in the code but make use of system parameters which are set via scripts.

Based on the current command, the `FMS` determines an appropriate mode to be set in the `AP`, and evaluates different cases of calculations, e.g., given the command in question, to update the lateral control surface a specific `APLATMODE` is set in the `FMS`. Then in the `AP`, this is the case considered in computing the updated lateral control surface values. The value computation is performed using Proportional-Integral-Derivative (PID) controllers (loops). As such, each PID loop will affect either a lateral, longitudinal, or speed control surface, and will result in a value which will be output to (or used in a calculation of the eventual output to) the actuator of a single control surface.

D. Low-level Computations

Now, we describe a single sequence of computations through the autopilot `FMS` and `AP` modules, executed under specific mode and command conditions. The sequence results in a signal to the aileron and a consequent change in the UAV heading. The particular calculations occur in the phases of a `LAND` command, namely *Approach*, *Descent* (or *Glide*), and *Flare* (Figure 2). These phases are represented in the code as mode transitions.

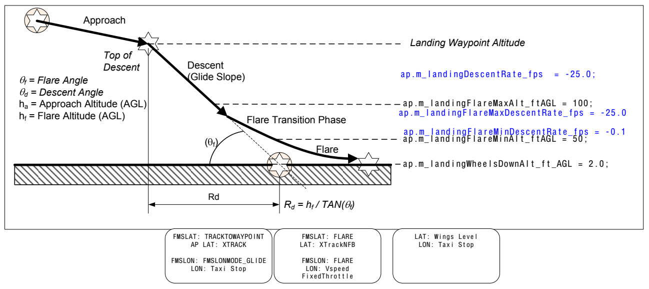

Approach
Landing Waypoint Altitude
Top of Descent
Descent (Glide Slope)
θ_f = Flare Angle
θ_d = Descent Angle
h_a = Approach Altitude (AGL)
h_f = Flare Altitude (AGL)
ap.m_landingDescentRate_fps = -25.0;
ap.m_landingFlareMaxAlt_ftAGL = 100;
ap.m_landingFlareMaxDescentRate_fps = -25.0
Flare Transition Phase
ap.m_landingFlareMinDescentRate_fps = -0.1
ap.m_landingFlareMinAlt_ftAGL = 50;
(θ_f)
Flare
ap.m_landingWheelsDownAlt_ft_AGL = 2.0;
Rd
$R_d = h_f / TAN(\theta_f)$
FMSLAT: TRACKTOWAYPOINT
AP LAT: XTRACK
FMSLON: FMSLONMODE_GLIDE
LON: Taxi Stop
FMSLAT: FLARE
LAT: XTrackNFB
FMSLON: FLARE
LON: Vspeed
FixedThrottle
LAT: Wings Level
LON: Taxi Stop

Figure 2. LAND command induced landing profile.

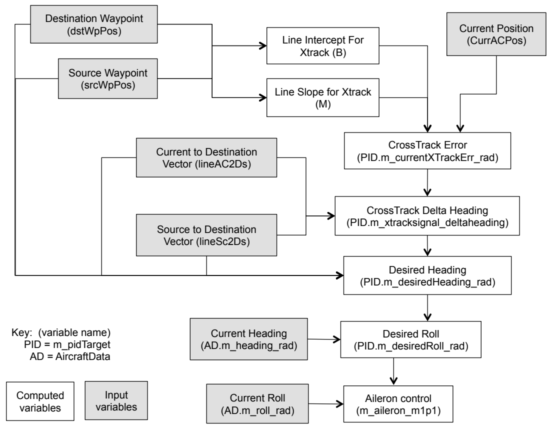

Destination Waypoint (dstWpPos)
Source Waypoint (srcWpPos)
Line Intercept For Xtrack (B)
Line Slope for Xtrack (M)
Current Position (CurrACPos)
CrossTrack Error (PID.m_currentXTrackErr_rad)
Current to Destination Vector (lineAC2Ds)
CrossTrack Delta Heading (PID.m_xtracksignal_deltaheading)
Source to Destination Vector (lineSc2Ds)
Desired Heading (PID.m_desiredHeading_rad)
Key: (variable name)
PID = m_pidTarget
AD = AircraftData
Current Heading (AD.m_heading_rad)
Desired Roll (PID.m_desiredRoll_rad)
Computed variables
Input variables
Current Roll (AD.m_roll_rad)
Aileron control (m_aileron_m1p1)

Figure 3. Dependencies in aileron control computation.

The variable `m_aileron_m1p1` in the code holds the value that will be routed to the aileron actuator through the *Reflection* framework. The calculation of the value stored in `m_aileron_m1p1` eventually traces back to the `FMS` class and the `Autopilot` class.

Figure 3 shows how the actual computations needed to make adjustments in the aileron control surface is dependent on a number of cascading calculations. At each level, a value is derived from PID loops and then used, in conjunction with the aircraft state, in a calculation at a lower level. For the aileron control surface, this corresponds to the cross-track, heading, roll angle, and aileron PID loops as shown. To adjust the aileron, the adjustment to the current heading is to be determined. Then, from the UAV state information (i.e., the current position, the source and the destination waypoints), a new heading is derived.

Thereafter, geometric calculations determine the UAV heading and position relative to the line connecting the source and destination. The distance of the UAV from this line is the cross-track (Xtrack) error. This is the parameter passed to the PID loop that determines the heading change needed to reach the destination waypoint, the delta heading. The desired heading takes into account both the heading based on the line from the source to destination and the delta heading.

From the difference in the current heading and the desired heading (the error), a new desired roll can be determined. Finally, to initiate a change in the roll (and hence, the heading), the aileron must be moved by an amount which will produce the desired roll. This is computed from the aileron PID loop which takes the roll error (the difference between the current aircraft roll and the desired roll as calculated) as an argument. The PID loop output is finally stored in the variable `m_aileron_m1p1`.

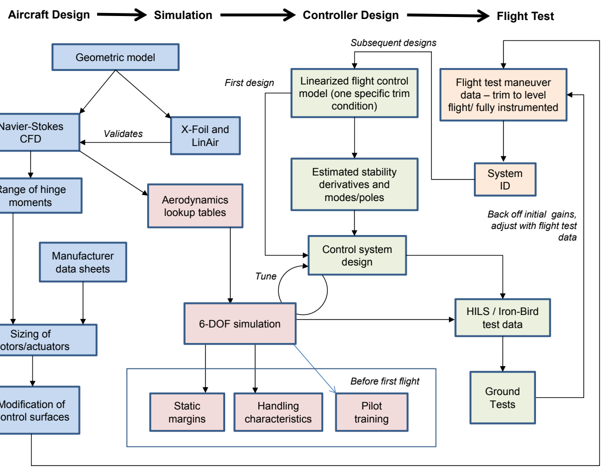

Aircraft Design
Simulation
Controller Design
Flight Test
Geometric model
Subsequent designs
First design
Linearized flight control model (one specific trim condition)
Flight test maneuver data – trim to level flight/ fully instrumented
Navier-Stokes CFD
Validates
X-Foil and LinAir
Range of hinge moments
Aerodynamics lookup tables
Estimated stability derivatives and modes/poles
System ID
Back off initial gains, adjust with flight test data
Manufacturer data sheets
Control system design
Tune
6-DOF simulation
HILS / Iron-Bird test data
Sizing of motors/actuators
Before first flight
Modification of Control surfaces
Static margins
Handling characteristics
Pilot training
Ground Tests

Figure 4. Abstract design flow for the Swift UAV.

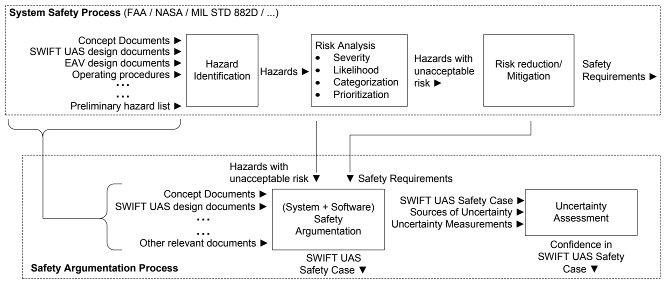
III. Approach

A. Characterizing Heterogeneity

We begin by explicitly characterizing the heterogeneity to be considered for system safety and safety argumentation. This is an essential step in tailoring a generic safety methodology for a specific application domain; in our case, aviation. We believe the main value of this, for the system safety process, is to manage the wider context of safety, i.e., to identify and manage hazards arising from system *interactions*. In addition, although evidence-based argumentation is largely product focused, safety implications also arise from other related sources, e.g., process and procedural deviations during operation, incorrect/implicit assumptions, etc.

Consider the high-level design flow for (a part of) the control system and the control surfaces of the UAV (Figure 4). The design begins with generated models of the intended system, which are related to the actual hardware to be mounted on the aircraft. The models also provide six degrees of freedom (6-DOF) simulation used to fine tune the UAV implementation. The control system is iteratively tested, and at each iteration it is also tuned against the simulation and refined with different parameters.

From Figure 4, it is evident that heterogeneous sources of assumptions, context and evidence exist, and ought to be considered, when analyzing the safety of this system. In general, these include elements such as: (1) *procedural, development and safety standards* imposing design and safety



System Safety Process (FAA / NASA / MIL STD 882D / ...)
Concept Documents ▶
SWIFT UAS design documents ▶
EAV design documents ▶
Operating procedures ▶
Preliminary hazard list ▶
Hazard Identification
Hazards ▶
Risk Analysis
Severity
Likelihood
Categorization
Prioritization
Hazards with unacceptable risk ▶
Risk reduction/ Mitigation
Safety Requirements ▶
Hazards with unacceptable risk ▼
▼ Safety Requirements
Concept Documents ▶
SWIFT UAS design documents ▶
Other relevant documents ▶
(System + Software) Safety Argumentation
SWIFT UAS Safety Case ▶
Sources of Uncertainty ▶
Uncertainty Measurements ▶
Uncertainty Assessment
SWIFT UAS Safety Case ▼
Confidence in SWIFT UAS Safety Case ▼
Safety Argumentation Process

Figure 5. Safety assurance methodology.

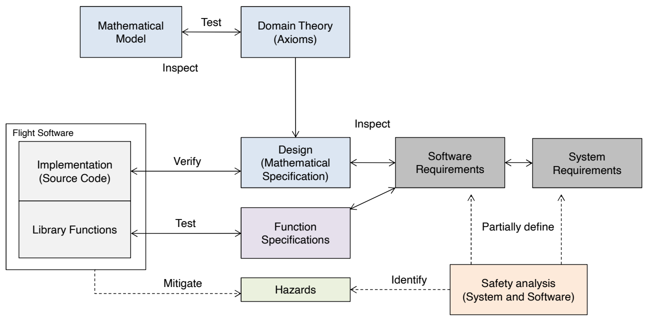


Figure 6. Software verification methodology.

constraints on the system; (2) *procedures* describing operations, e.g., for maintenance (3) *mathematical theory*, e.g., the theory of aerodynamic stability (4) *implicit assumptions*, i.e., simplifying assumptions such as decoupled dynamics (5) *vehicle flight logs*, e.g., which show the system behavior under specific flight conditions (6) *calibration experiments* such as those used to calibrate sensors (7) *hardware tests*, e.g., static load tests used for determining actuator sizes (8) *aircraft models*, e.g., geometric, and computational fluid dynamics (CFD) models (9) *data-sheets* providing component parameters and specifications (10) *simulations* which progressively evaluate the actual system to be flown (11) *software models* of the sensors, actuators, commands, and flight management (12) *range safety calculations* providing estimates on the expected casualty rate based on the area in which the aircraft is operated (13) *expert opinion* often including decisions which might not be explicitly documented.

B. Safety Assurance Methodology

The main philosophy of our safety assurance methodology (Figure 5) is that the system safety process drives the safety argumentation process.

1) System Safety Process: Our system safety process (Figure 5) is based on the framework of a safety risk management plan [5], and includes safety considerations into system design at an early stage through hazard identification, risk analysis, and risk management. In brief, hazard identification and risk analysis involves respectively (i) determining those situations or conditions relevant to the system which, if left uncontrolled, have the potential to cause an undesirable event, and (ii) characterizing the consequences, severity and likelihood of such situations/conditions. Risk management broadly uses the outcomes of risk analysis to prioritize and mitigate risks. Note that Figure 5 does not indicate the iterative nature of the safety process, and that it is phased with system development. Thus, system-level safety requirements would be initially derived during requirements engineering, while lower-level safety requirements are developed during system design. Additionally not all of the input shown is required, or used, in all the iterations.

As input to these steps, we use the identified heterogeneous sources (Section III-A) including, but not limited to, the concept of operations, available design documentation, and previously identified hazards (captured in a preliminary hazard list). We define mitigation measures for those hazards considered as having unacceptable risk, from the outcome of hazard identification and risk analysis. One specific outcome of the risk reduction/mitigation step is requirements on system safety. These requirements take several forms, including constraints on the design, guidelines and/or procedures for maintenance, operation, etc. See Section IV-A for an example of such analysis.

2) Safety Argumentation: The outcome of the system safety process, in effect, triggers the safety argumentation process (Figure 5). The general idea is to use a structured safety case, to systematically argue that all identified hazards have been eliminated or mitigated, such that mishap risks have been reduced to acceptable levels.

The safety argumentation process is applied starting at the level of the system in the same way as the system safety process, and it is repeated at the software level. Abstractly, the main steps in creating a software (and system) safety case are to: (a) create safety claims, e.g., indicating the mitigation of relevant hazards, and (b) link the evidence that supports the claims via an appropriate and structured argument. These steps are non-trivial, but can be operationalized through well-developed methodologies [6], [7].

3) Software Verification Methodology: We use our software verification methodology (Figure 6) to create part of the *software* safety argument, (in particular, the lower levels). This also has connections to the wider system safety process, as shown. To verify the flight software, we use formal verification of the implementation against a mathematical specification and test low-level library functions against their specifications. In this paper we concentrate on formal verification using AUTOCERT [8], and defer testing, and verification using other tools to future work.

The specification formalizes software requirements which, in turn, are derived from system requirements during the safety analysis. The formal verification takes place in the context of a logical domain theory (i.e., a set of axioms and function specifications). Axioms can be subjected to

```
assumptions
    airplaneData->m_heading_rad::
       current(heading).
    airplaneData->m_pos_altitude_ft::
       current(altitude).
    airplaneData->m_pitch_rad::
       current(pitch).
    airplaneData->m_pos_north_ft::
       pos(north).
    airplaneData->m_pos_east_ft::
       pos(east).
    airplaneData->m_roll_rad::
       current(roll).
    srcWpPos ::pos(ne).
    dstWpPos ::pos(ne).
    currACPos::pos(ne).

requirements
    output->m_aileron_m1p1 ::desired(aileron).
    output->m_elevator_m1p1::desired(elevator).
```

Figure 7. Specification (excerpt): assumptions and requirements.

```
definition(calc_error
 , ['calculate error in', T]
 , Y-Z
 , [Y::current(T), Z::desired(T)]
 , [_::error(T)]
).

definition(initial_heading
 , ['initial heading from', Src, 'to', Dst]
 , CalcHeadAngle(Dst[0], Dst[1], Src[0], Src[1])
 , [Dst::pos(ne), Src::pos(ne)]
 , [_::rad]
).
```

Figure 8. Annotation schemas (excerpt): heading and error.

increasing levels of scrutiny, going from simply assuming that they are valid, to inspecting them, up to testing them against a computational model which, itself, is inspected [9].

4) Formal Verification: Figure 7 shows a specification fragment consisting of assumptions about the current aircraft state and flight plan, and requirements on signals to the control surfaces. `currACPos` is the current aircraft position in the *North-East* frame of reference, and `srcWpPos` and `dstWpPos` are the current and next waypoints, respectively, in the flight plan.

Given these assumptions on the aircraft state, we must show that the code that implements the *descent* phase of the LAND command (Figure 2) correctly modifies the aileron and elevator. To do so, we verify the code against its specification using AUTOCERT, which infers logical annotations on the code under verification, using mathematical annotation schemas (Figure 8). AUTOCERT then applies a verification condition generator (VCG), which uses the annotations and function specifications (Figure 9) to generate a set of verification conditions (VCs), logical conjectures which are sent to a suite of automated theorem provers (ATPs) along with the axioms (Figure 10). Most VCs conjecture that the regulation of a variable within some bounds maintains a given property.

Both the axioms and verification conditions are given in first-order logic, and use predicates that denote geometric

```
function(aileron_out
  , ['convert roll error in', E,
     'to desired aileron in', D]
  , m_pid_RollErr2Aileron->Update1(E)
  , [E::error(roll)]
  , [D::desired(aileron)]
).
```

Figure 9. Function specification: roll error to aileron.

desired_heading_from_delta:
$$\forall X.\forall D.\, \mathtt{has\_unit}(X, \mathtt{desired}(\mathtt{delta}(\mathtt{heading}))) \wedge \mathtt{has\_unit}(D, \mathtt{initial}(\mathtt{heading})) \Rightarrow \mathtt{has\_unit}(D+X, \mathtt{desired}(\mathtt{heading}))$$

desired_heading_lower_bound:
$$\forall H.\, \mathtt{has\_unit}(H, \mathtt{desired}(\mathtt{heading})) \wedge (H < -\pi) \Rightarrow \mathtt{has\_unit}(2\pi, \mathtt{desired}(\mathtt{heading}))$$

desired_heading_upper_bound:
$$\forall H.\, \mathtt{has\_unit}(H, \mathtt{desired}(\mathtt{heading})) \wedge (\pi < H) \Rightarrow \mathtt{has\_unit}(-2\pi, \mathtt{desired}(\mathtt{heading}))$$

Figure 10. Axioms (excerpt): desired heading.

and navigational concepts, including physical units such as `heading` and `roll`, control surfaces like `aileron` and `elevator`, frames of reference, such as `lla` (lat-long-alt) and `ne` (North-East), and aspects of properties such as `desired(T)`, `current(T)`, `error(T)`, where `T` is a unit in the aircraft state. The axioms express the definition of geometric properties, and the various adjustments in orientation through different quadrants, based upon dependencies on the aircraft state and the desired value needed to accomplish the movement of the control surface.

An annotation schema describes a computation pattern along with its logical pre- and post-conditions. Schemas are compiled into low-level patterns that match against the code (for details, see [10]). For example, `calc_error` states that if `Y` and `Z` represent current and desired values of some state variable `T`, then `Y-Z` computes the error in `T`. After matching against the corresponding code fragment and inserting pre- and post-conditions, the tool will then recursively search for definitions of `Y` and `Z`, and so on. Similarly, `initial_heading` matches against code that calls a library function to compute the heading in radians, given that `Dst` and `Src` represent the aircraft position in the `ne` frame. Figure 9 specifies a PID function: given the error on the roll as input, the output is the desired value of the aileron. Specifications, schemas, and axioms can be extended with additional non-logical information [11] such as justifications and contexts (not shown here). Formally verifying the requirement `output-> m_aileron_m1p1::desired(aileron)` generates 51 VCs which can then be proven using a suite of first-order ATPs.

C. Assembling the Safety Case

There are two ways in which a formal method can be integrated with the construction of a safety case. Going in one direction, the output of AUTOCERT can be transformed into a safety case fragment. Going in the other, safety case fragments can be transformed into formal specifications that are then input to AUTOCERT.

1) From Formal Proofs to Safety Cases: AUTOCERT generates a document (in XML) with information describing the formal verification of requirements. The core of this is the chain of information relating requirements back to assumptions. Recall that Figure 3 illustrated the sequence of calculations used to compute the aileron control variable from several inputs drawn from the current aircraft state and the flight plan.

Each step therein is described by (1) an annotation schema for the definition of a program variable, (2) the associated VCs that must be shown for the correctness of that definition, and (3) the variables on which that variable, in turn, depends.

We derive the goals (and subgoals) of the safety case from the annotation schema. The subgoals are the dependent variables from those annotation schema. We represent each VC related to a goal as a subgoal. An argument for a VC is a proof, generated using a subset of the axioms. This proof forms the evidence connected to the VC goal, and includes the prover used as a context. Function specifications from external libraries used in the software and its verification also appear as goals. Arguments for these goals can be made with evidence such as testing or inspection. Each subgoal derived from an annotation schema is a step in the verification process. Figure 13 gives an example of such a step.

During the process of merging the manually created and the auto-generated safety cases, we replace specific nodes of the manually created safety case with the tree fragments generated from AUTOCERT; specifically, the top-level goals of the latter are grafted onto the appropriate lowest-level nodes of the former. These nodes are denoted with unique comments, `autocert:id`, relating the node to a tree in the automatically created file, meaning that the goal with tag `id` is to be solved with AUTOCERT.

2) From Safety Cases to Formal Specifications: Often, a safety case fragment may be created before the software verification is completed. In this case, we can use the `autocert:id` annotations on the nodes to generate a formal specification. Based on the type of node in which the identifier occurs, the tool infers whether the labeled node is a requirement or an assumption. After running AUTOCERT on the generated specification, we can graft the resulting proofs back into the safety case.

IV. SAFETY ANALYSIS

A. Hazard Analysis

During hazard identification and preliminary hazard analysis (PHA) for the Swift UAS, we systematically identified and documented the known hazards, while brainstorming for new hazards. Hazards identified thus far include those induced from subsystem failures, deviations from procedures, as well as interactions. We performed this step in close cooperation with the Swift UAS engineering team, using documents related to the concept of operations, preliminary design, operating procedures, as well as other heterogeneous information (as identified in Section III-A), as input.

We also applied failure modes and effects analysis (FMEA) as part of bottom-up reasoning for hazard identification. Although the engineering of the Swift UAS is ongoing and FMEA is typically employed when low level system details are sufficiently well known, in our case FMEA was feasible due to the significant domain knowledge of the engineering team and, in part, due to the reuse of certain systems from the EAV.

Table I shows an excerpt of the hazard analysis, considering failure hazards in the avionics software (specifically, the autopilot) during the *descent* phase of the landing profile induced by the `LAND` command (Figure 2). In Table I, the risk category for the autopilot controller failure hazard is given as 2B, which is considered as *unacceptable risk*, based on the risk classification table given in [5]. The table also shows the consequent definition of the relevant functional safety requirements which, when correctly implemented, are expected to mitigate the hazard.

B. Swift UAS Safety Case

In this paper, we use the Goal Structuring Notation (GSN) [6] to document our safety case. We present an end-to-end "slice" of the overall safety case for the Swift UAS shown as a bird's eye view in Figure 11. The safety case starts with a top-level safety goal: *The Swift UAS is safe in the context of a specific mission, in a specific configuration, on the defined range where it is to be operated, under specific weather conditions*. This goal is justified, in part, through an argument that the airborne system (the UAV) is safe. This itself consists of the arguments assuring the mitigation of UAV hazards including, in particular, the mitigation of the autopilot controller failure hazard during descent (Table I). The latter is demonstrated, in part, by an argument that the autopilot is correct. This, in turn, is justified by an argument generated from a formal proof of correctness (Section III-B4) of the computations of the values of the aileron during descent (Section II-D).

Thus, what we have shown in Figure 11 is only a small part of the overall Swift UAS safety case, which also contains arguments assuring the safety of the ground system, the communication infrastructure, and the operation, besides that of the airborne system. Broadly, the safety case slice in Figure 11 comprises a *manually created* fragment and an *automatically generated* fragment. Each of these is now described in greater detail.

1) Manually Created Safety Argument: The manually created safety case is based on the hazard analysis performed in Section IV-A. We make the case for the safety of the airborne system by argument over all identified hazards (which can be considered as an instance of the *hazard directed breakdown pattern* [12]) and create the argument

Table I
EXCERPT FROM SWIFT UAS HAZARD ANALYSIS.

ID	HAZARD / SCENARIO DESCRIPTION	POTENTIAL CAUSES	EFFECT ON SYSTEM	LIKELIHOOD	SEVERITY	RISK CATEGORY	MITIGATION MEASURES	CORRECTIVE ACTION	SAFETY REQUIREMENT
PHA_DE.APP _AVCS_012	Flight management system (FMS) failure	1. Deadlocks 2. Timing errors 3. Memory corruption 4. Incorrect specification 5. Incorrect implementation 6. Inaccurate/ incorrect assumptions 7. Wrong interpretation of theory	Incorrect computation of control surface signals	Remote	Major	3B	(1) Ground station pilot controller overrides autopilot (2) Failsafe autopilot intervenes when failure of autopilot detected	Verify that specification is consistent with theory	[FSP_AVCS_004] When FMS failure is detected, it is always the case that failsafe autopilot eventually takes control within a specified time duration
PHA_DE.APP _AVCS_015	Autopilot controller module failure		Loss of flight	Remote	Hazardous	2B		(1) Verify legality of issued commands (2) Guarantee correct interpretation of commands	[A1] Commands must be interpreted correctly [A2] The autopilot executes safe maneuvers for all commands during descent.

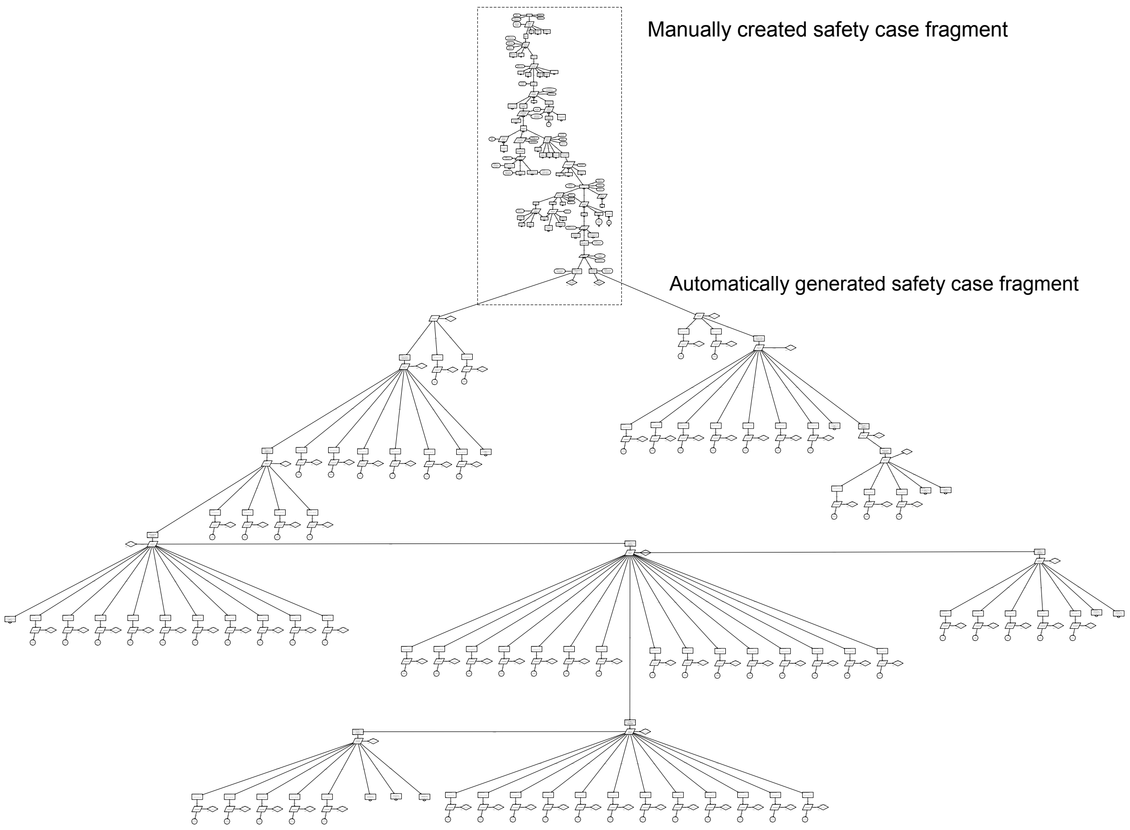

Manually created safety case fragment
Automatically generated safety case fragment

Figure 11. Safety case fragment of the airborne system in the Swift UAS: bird's eye view.

structure in *layers*, i.e., we first develop the top-level safety claim into claims about the mitigation of UAV hazards during its phases of operation, e.g., UAV failure hazards during descent, and then further develop this into claims linking

(a) the UAV sub-systems and the avionics software
(b) the avionics software and the autopilot module
(c) the autopilot module and the autopilot controller (`AP`)
(d) the `AP` module and the PID controller for the aileron.

The argument fragment which justifies the claims at this last level is then automatically generated. Figure 12 shows an excerpt from the manually created safety case fragment (highlighted in Figure 11). In particular, it shows how the claim of mitigating software failures during descent is broken down by using the strategy of making correctness arguments, into the claims that (1) the avionics software modules and commands are correct, and (2) the autopilot module is correct.

Arguing correctness is not always required when making a safety claim. However, in our case the correctness of the avionics software is related to functional safety, i.e., incorrect behavior is unsafe behavior. This justification is explicit in the argument fragment through the definition of correctness of the software components. In the particular case of the failsafe autopilot (which is part of the avionics software and also forms a part of the Swift UAS CMS), its correct behavior is required to assure safety.

Figure 12 also shows how the claim of mitigating avionics software failures during descent is linked to an identified high-level functional safety requirement (Table I), i.e., "the

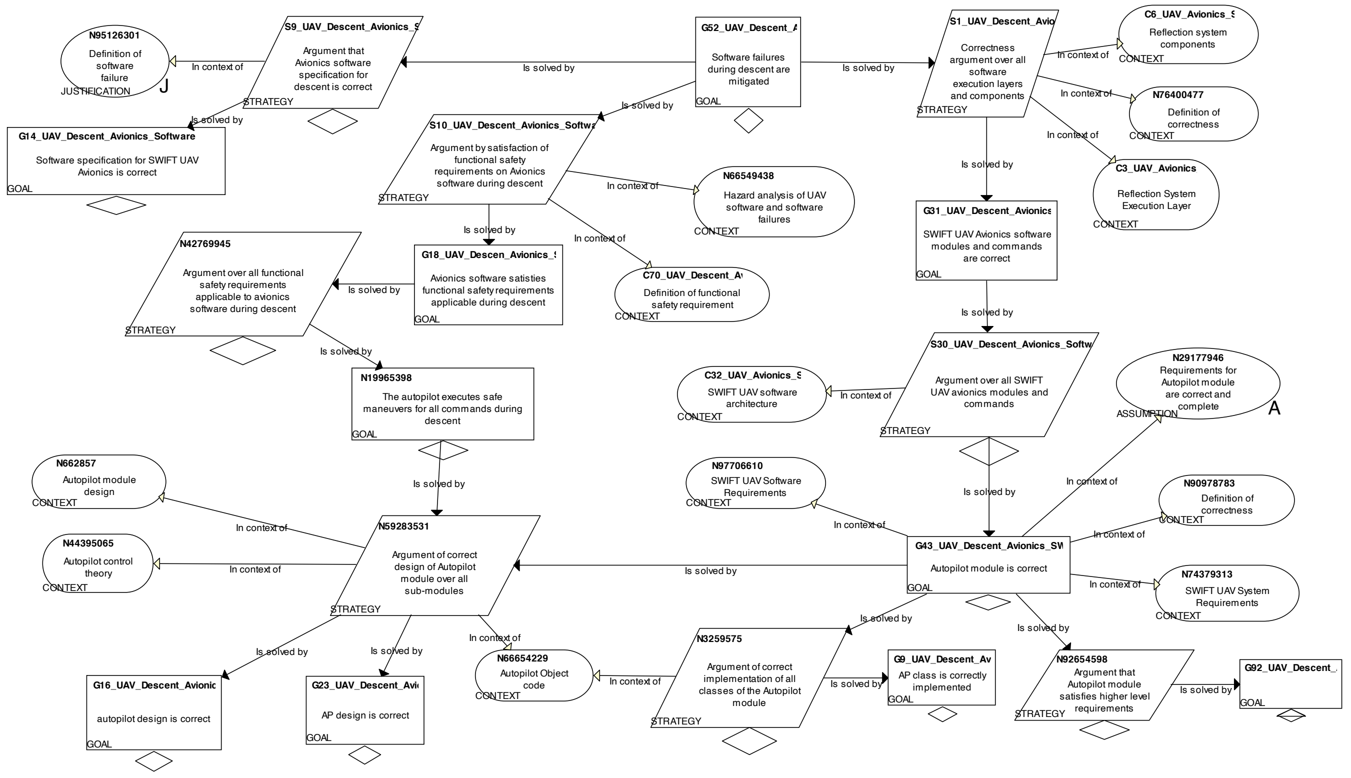

N95126301
Definition of software failure
JUSTIFICATION
J
In context of
S9_UAV_Descent_Avionics_S
Argument that Avionics software specification for descent is correct
STRATEGY
Is solved by
G52_UAV_Descent_A
Software failures during descent are mitigated
GOAL
Is solved by
S1_UAV_Descent_Avio
Correctness argument over all software execution layers and components
STRATEGY
In context of
C6_UAV_Avionics_S
Reflection system components
CONTEXT
In context of
N76400477
Definition of correctness
CONTEXT
In context of
C3_UAV_Avionics
Reflection System Execution Layer
CONTEXT
Is solved by
G14_UAV_Descent_Avionics_Software
Software specification for SWIFT UAV Avionics is correct
GOAL
Is solved by
S10_UAV_Descent_Avionics_Softw
Argument by satisfaction of functional safety requirements on Avionics software during descent
STRATEGY
In context of
N66549438
Hazard analysis of UAV software and software failures
CONTEXT
Is solved by
G31_UAV_Descent_Avionics
SWIFT UAV Avionics software modules and commands are correct
GOAL
In context of
C70_UAV_Descent_Av
Definition of functional safety requirement
CONTEXT
N42769945
Argument over all functional safety requirements applicable to avionics software during descent
STRATEGY
Is solved by
G18_UAV_Descen_Avionics_S
Avionics software satisties functional safety requirements applicable during descent
GOAL
Is solved by
S30_UAV_Descent_Avionics_Softw
Argument over all SWIFT UAV avionics modules and commands
STRATEGY
In context of
C32_UAV_Avionics_S
SWIFT UAV software architecture
CONTEXT
N29177946
Requirements for Autopilot module are correct and complete
ASSUMPTION
A
Is solved by
N19965398
The autopilot executes safe maneuvers for all commands during descent
GOAL
N662857
Autopilot module design
CONTEXT
Is solved by
N97706610
SWIFT UAV Software Requirements
CONTEXT
Is solved by
In context of
N90978783
Definition of correctness
CONTEXT
In context of
N44395065
Autopilot control theory
CONTEXT
In context of
N59283531
Argument of correct design of Autopilot module over all sub-modules
STRATEGY
Is solved by
In context of
G43_UAV_Descent_Avionics_SW
Autopilot module is correct
GOAL
In context of
In context of
N74379313
SWIFT UAV System Requirements
CONTEXT
Is solved by
Is solved by
In context of
Is solved by
Is solved by
N3259575
Argument of correct implementation of all classes of the Autopilot module
STRATEGY
G16_UAV_Descent_Avionic
autopilot design is correct
GOAL
G23_UAV_Descent_Avi
AP design is correct
GOAL
N66654229
Autopilot Object code
CONTEXT
In context of
Is solved by
G9_UAV_Descent_Av
AP class is correctly implemented
GOAL
N92654598
Argument that Autopilot module satisfies higher level requirements
STRATEGY
Is solved by
G92_UAV_Descent_
GOAL

Figure 12. Excerpt from the manually created safety case fragment.

autopilot executes safe maneuvers for all commands during descent". For this work, we have further developed the claim that the AP module (class) is correctly implemented. Subsequently, arguing this claim amounts to arguing that the implementation of the PID controller for each control surface is correct. Here, we are concerned with the correct controller updates for the aileron. From this point forward, the safety argument is automatically generated.

2) Automatically Generated Safety Argument: We automatically create the safety case fragments using the verification information generated by AUTOCERT. Figure 13 shows an example of such auto-generated content relating to the proof for the requirement on the `m_aileron_m1p1` variable. The schema `aileron_out` is the context for correctness, and this is verified by proving a number of theorems (only the proof status of the first is shown here) with the specified prover.

The evidence node denotes that a proof was found, but the details of the proof are omitted. We also see a portion of the chain of dependent variables: specifically the dependency of the correctness of `m_aileron_m1p1` on the correctness of `rollError_rad` and the correctness of `m_desiredroll_rad`. Each variable links to its own verification information. Finally, Figure 13 shows the inclusion of a context node describing the domain theory that was used. The resulting automatically generated safety case gives the complete argument for verification of the software components.

Once the automatically generated safety case fragment is created, it must be merged with the manually created fragment described earlier. To do this, we use unique identifiers and associate the top-level goals in the auto-generated safety case, with the incomplete goals from the manually created fragment, i.e., for each top level requirement from the AUTOCERT verification, we match nodes in the auto-generated fragment with nodes in the manually created fragment via the identifier. In the case of the autopilot software, we verify two requirements (shown to be descending from the manually created fragment, in Figure 11).

To complete the argument started in the manually created safety case, we graft the two auto-generated argument trees onto the appropriate, manually created nodes.

V. RELATED WORK

Systematic approaches to develop safety arguments in general [13], [7], and specifically for software [14], are based on goals-based argumentation frameworks; they indicate the role of safety processes and evidence selection in creating the system and/or software safety case. Higher order logic PVS has been applied to formalize a top-level safety argument to support mechanized checking, e.g., of soundness, of the argument in [15], whereas [16] applies classical logic to

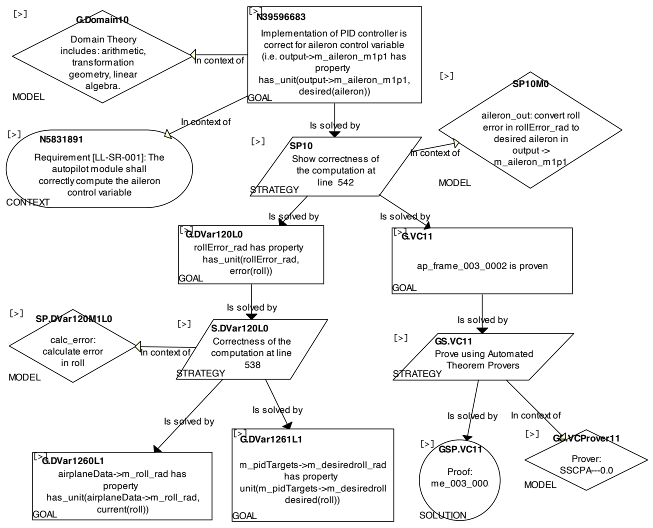

[>]
G.Domain10
Domain Theory includes: arithmetic, transformation geometry, linear algebra.
MODEL
[>]
N39596683
Implementation of PID controller is correct for aileron control variable (i.e. output->m_aileron_m1p1 has property has_unit(output->m_aileron_m1p1, desired(aileron))
GOAL
In context of
In context of
[>]
N5831891
Requirement [LL-SR-001]: The autopilot module shall correctly compute the aileron control variable
CONTEXT
Is solved by
[>]
SP10
Show correctness of the computation at line 542
STRATEGY
In context of
[>]
SP10M0
aileron_out: convert roll error in rollError_rad to desired aileron in output -> m_aileron_m1p1
MODEL
Is solved by
Is solved by
[>]
G.DVar120L0
rollError_rad has property has_unit(rollError_rad, error(roll))
GOAL
[>]
G.VC11
ap_frame_003_0002 is proven
GOAL
Is solved by
Is solved by
[>]
SP.DVar120M1L0
calc_error: calculate error in roll
MODEL
In context of
[>]
S.DVar120L0
Correctness of the computation at line 538
STRATEGY
[>]
GS.VC11
Prove using Automated Theorem Provers
STRATEGY
Is solved by
Is solved by
Is solved by
In context of
[>]
G.DVar1260L1
airplaneData->m_roll_rad has property has_unit(airplaneData->m_roll_rad, current(roll))
GOAL
[>]
G.DVar1261L1
m_pidTargets->m_desiredroll_rad has property unit(m_pidTargets->m_desiredroll desired(roll))
GOAL
[>]
GSP.VC11
Proof: me_003_000
SOLUTION
[>]
GC.VCProver11
Prover: SSCPA---0.0
MODEL

Figure 13. Excerpt from the auto-generated fragment.

derive safety cases using formal methods in model-based development. Formal methods have been also applied in other incarnations of goal-based argumentation, so-called *assurance cases* [17], and *dependability cases* [18]. In [19], an approach for identifying assurance deficits in safety arguments with a view towards improving their trustworthiness is given. Along these lines, the role of diversity and uncertainty in dependability/safety cases has been addressed in [20] and [21], [22] respectively.

Four aspects of our work distinguish it from the existing literature:

(1) The safety argument that we have created (Figure 11) provides a level of detail going well beyond the state of the practice. Safety cases typically leave many details implicit or informal, and rarely go down to the level of software implementations. Making safety-relevant data and its connections to requirements explicit is highly worthwhile since a safety case serves primarily as a form of communication.

(2) Thus far, safety cases have not generally combined manually developed and auto-generated fragments. Where automation is used, it tends to be as a black box that provides a single piece of evidence, and not a full argument fragment. We have demonstrated the feasibility of automated assembly of manually developed safety case fragments with those generated automatically (in Section III-C).

(3) We have combined traditional safety analysis techniques with formal methods (in Section III-B). Although formal methods have been used in safety cases, much of the existing work does not deal with the wider context of safety or with argument generation.

(4) Safety is inherently heterogeneous; we have characterized the diversity of the information sources pertinent to safety (in Section III-A) and explicitly reflected it in the argument fragments created, e.g., autopilot control theory forms the *context* when arguing the validity of autopilot software (Figure 12). Furthermore, we view "formal" and "non-formal" sources not as opposites, but as complementary and equally relevant. We have also highlighted how software is considered in the system context with explicit justification for the constituent parameters and specifications, when viewed from a safety perspective (Figure 13).

VI. Concluding Remarks

We have shown that it is feasible to automatically assemble auto-generated fragments of safety cases derived from a formal verification method, i.e., proofs of correctness using AutoCert, with manually created fragments derived from safety analysis. We illustrated our approach describing an end-to-end slice of the overall safety case for the Swift UAS.

Our intent, here, is primarily to show how heterogeneous safety aspects, especially those arising from formal, and non-formal reasoning, can be communicated in a unified way, rather than to claim safety improvement. The comprehensive safety case (the creation of which is ongoing) will include arguments for the safety of the GCS, communications, and operations, besides addressing other aspects of the airborne system, i.e., other properties of the flight software, both correctness and safety, carefully accounting for a greater range of faults, modes and commands of the autopilot, and carrying the analysis through all the execution layers.

For the argument fragment described here, as well as for the comprehensive safety case, several additional aspects are noteworthy. Firstly, how our safety argument is initially structured, we believe, is likely to play an appreciable role in its comprehensibility and complexity, e.g., one strategy to develop the initial safety claim is to argue that the relevant hazards across all operating phases of the UAS have been managed. An alternative strategy is to develop the top-level claim, first, into claims on the subsystems of the UAS, and then across the relevant operating phases. The former allows us to address, at the outset, those hazards which can change risk categories depending on the mission phase, e.g., failure of the nose wheel actuator may not be hazardous during the cruise phase but it is during landing. Whereas, the latter facilitates the creation of a safety argument which may be both easier to maintain and better modularized.

Second, there is a need for a quantitative assessment framework for the arguments created [23] to support decision making, e.g., whether an argument is valid and/or covers sufficient information. We have identified an initial set of metrics that we believe will support this broad goal, and Table II shows a subset thereof. Due to space constraints, we do not report on their specification here. To illustrate their role, however, consider COV_H: this metric indicates reasonably high (about 73%), but not complete, coverage of the hazards considered, reflecting the fact that not all claims arising from the hazards in the argument fragment have been fully developed, i.e., terminate in evidence. In

Table II
QUANTITATIVE METRICS FOR FIGURE 11 (EXCERPT).

Measure	**Value**	**Description**
C	144	Total number of claims.
H_c	0.1428	Coverage of hazards identified.
COV_H	0.7344	Coverage of considered hazards.
$COV_{\mathcal{R}_{HL}}$	0.8667	Coverage of high-level safety requirements.

large safety cases, such metrics can conveniently summarize the state of the safety argument during system evolution. An implicit assumption underlying the coverage measures here is that the argument chains in the safety case are themselves valid. Thus, assessing argument validity, e.g., via quantitative uncertainty assessment [22], and including this notion into the earlier coverage measures, appears to be a promising mechanism to support such decision making.

Third, the inclusion of auto-generated fragments, and all relevant sources of information, will lead to increasingly large safety cases. Since the primary motivation of a safety case is to communicate safety relevant information to the relevant stakeholders, we believe that a safety case should be viewed not as a static, unchanging artifact, rather as one amenable to manipulation in various automated ways, e.g., by generating traceability matrices. We believe that this will ameliorate the integration of safety cases into existing process-based methodologies.

Finally, we believe that the work presented here is a promising step towards increased safety assurance, particularly in UAS. Safety analysis is imperative to determine the required regulations (and whether existing regulations are sufficient and/or how they ought to be augmented): [24] identifies several UAS hazards and their implications for regulation, while [25] gives the properties for an effective framework for airworthiness certification for UAS. Research on the generation of safety cases, thus affords the development of a framework for assuring safety in tandem with the identification of UAS-relevant hazards.

ACKNOWLEDGMENTS

This work was funded by NASA contract NNA10DE83C.

REFERENCES

[1] R. Bloomfield and P. Bishop, "Safety and assurance cases: Past, present and possible future – an Adelard perspective," in *Proc. 18th Safety-Critical Sys. Symp.*, Feb. 2010.

[2] International Organization for Standardization (ISO), "Road Vehicles-Functional Safety," ISO 26262 Draft Standard, Baseline 15, 2010.

[3] K. Davis, "Unmanned Aircraft Systems Operations in the U.S. National Airspace System," Interim Operational Approval Guidance 08-01, FAA Unmanned Aircraft Systems Program Office, Mar. 2008.

[4] UK Ministry of Defence (MoD), *Safety Management Requirements for Defence Systems*, Defence Standard 00-56, Issue 4, 2007.

[5] Federal Aviation Administration, *System Safety Handbook*, FAA, Dec. 2000.

[6] Goal Structuring Notation Working Group, "GSN Community Standard Version 1," Nov. 2011. [Online]. Available: http://www.goalstructuringnotation.info/

[7] P. Bishop and R. Bloomfield, "A methodology for safety case development," in *Proc. 6th Safety-critical Sys. Symp.*, 1998.

[8] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," in *IEEE Aerospace Conf. Electronic Proc.*, 2008.

[9] K. Ahn and E. Denney, "A framework for testing first-order logic axioms in program verification," *Software Quality Journal*, pp. 1–42, Nov. 2011.

[10] E. Denney and B. Fischer, "Generating customized verifiers for automatically generated code," in *Proc. Conf. Generative Prog. and Component Eng. (GPCE)*, Oct. 2008, pp. 77–87.

[11] N. Basir, E. Denney, and B. Fischer, "Building heterogeneous safety cases for automatically generated code," in *AIAA Infotech@Aerospace Conf.*, 2011.

[12] T. Kelly and J. McDermid, "Safety case patterns – reusing successful arguments," in *Proc. IEE Colloq. on Understanding Patterns and Their Application to Sys. Eng.*, 1998.

[13] T. Kelly, "Arguing safety: A systematic approach to managing safety cases," Ph.D. thesis, Univ. of York, 1998.

[14] R. Weaver, "The safety of software – constructing and assuring arguments," Ph.D. thesis, Univ. of York, 2003.

[15] J. Rushby, "Formalism in safety cases," in *Proc. 18th Safety-Critical Sys. Symp.*, Feb. 2010, pp. 3–17.

[16] N. Basir, E. Denney, and B. Fischer, "Deriving safety cases for hierarchical structure in model-based development," in *29th Intl. Conf. Comp. Safety, Reliability and Security (SafeComp)*, 2010.

[17] E. Lee, I. Lee, and O. Sokolsky, "Assurance cases in model-driven development of the pacemaker software," in *Proc. 4th Intl. Symp. Leveraging Application of Formal Methods, Verification and Validation (ISoLA)*, Part II, LNCS 6416, Oct. 2010, pp. 343–356.

[18] Y. Matsuno, H. Takamura, and Y. Ishikawa, "Dependability case editor with pattern library," in *Proc. 12th IEEE Intl. Symp. High-Assurance Sys. Eng. (HASE)*, 2010, pp. 170–171.

[19] R. Hawkins, T. Kelly, J. Knight, and P. Graydon, "A new approach to creating clear safety arguments," in *Proc. 19th Safety-Critical Sys. Symp.*, Feb. 2011.

[20] B. Littlewood and D. Wright, "The use of multilegged arguments to increase confidence in safety claims for software-based systems: A study based on a BBN analysis of an idealized example," *IEEE Trans. Soft. Eng.*, vol. 33, no. 5, pp. 347–365, May 2007.

[21] R. Bloomfield, B. Littlewood, and D. Wright, "Confidence: its roles in dependability cases for risk assessment," in *Proc. 37th Intl. Conf. Dependable Sys. and Networks (DSN)*, 2007.

[22] E. Denney, I. Habli, and G. Pai, "Towards measurement of confidence in safety cases," in *Proc. 5th Intl. Symp. Empirical Soft. Eng. and Measurement (ESEM)*, Sept. 2011.

[23] A. Wassyng, T. Maibaum, M. Lawford, and H. Bherer, "Software certification: Is there a case against safety cases?" in *Foundations Comp. Soft., Modeling, Dev. and Verification of Adaptive Sys.*, LNCS 6662, 2011, pp. 206–227.

[24] K. Hayhurst, J. Maddalon, P. Miner, M. DeWalt, and G. McCormick, "Unmanned aircraft hazards and their implications for regulation," in *25th IEEE/AIAA Digital Avionics Sys. Conf.*, Oct. 2006, pp. 1–12.

[25] R. Clothier, J. Palmer, R. Walker, and N. Fulton, "Definition of an airworthiness certification framework for civil unmanned aircraft systems," *Safety Science*, vol. 49, no. 6, pp. 871–885, 2011.