Automating the Assembly of Aviation Safety Cases

Ewen Denney, *Member, IEEE*, and Ganesh Pai, *Member, IEEE*

***Abstract*—Safety cases are among the state of the art in safety management mechanisms, providing an explicit way to reason about system and software safety. The intent is to provide convincing, valid, comprehensive assurance that a system is acceptably safe for a given application in a defined operating environment, by creating an argument structure that links claims about safety to a body of evidence. However, their construction is a largely manual, and therefore a time consuming, error prone, and expensive process. We present a methodology for automatically assembling safety cases which are auto-generated from the application of a formal method to software, with manually created safety cases derived from system safety analysis. Our approach emphasizes the heterogeneity of safety-relevant information, and we show how diverse content can be integrated into a single argument structure. To illustrate our methodology, we have applied it to the Swift Unmanned Aircraft System (UAS) being developed at the NASA Ames Research Center. We present an end-to-end fragment of the resulting interim safety case comprising an aircraft-level argument manually constructed from the safety analysis of the Swift UAS, which is automatically assembled with an auto-generated lower-level argument produced from a formal proof of correctness of the safety-relevant properties of the software autopilot.**

***Index Terms*—Safety cases, system safety, software safety, safety assurance, unmanned aircraft systems, formal methods.**

Acronyms and Abbreviations

AP	Autopilot Controller
ATP	Automated Theorem Prover
CPDS	Common Payload Data System
FHA	Functional Hazard Analysis
FMEA	Failure Modes and Effects Analysis
FMS	Flight Management System
GCS	Ground Control Station
GSN	Goal Structuring Notation
PHA	Preliminary Hazard Analysis
PRA	Probabilistic Risk Assessment
PID	Proportional-Integral-Derivative
RVM	Reflection Virtual Machine
UAS	Unmanned Aircraft System
UA	Unmanned Aircraft
VC	Verification Condition

Manuscript received March 03, 2013; revised November 11, 2013; accepted March 26, 2014. This work was supported in part by the Assurance of Flight Critical Systems (AFCS) element of the System-wide Safety Assurance Technologies (SSAT) project in the Aviation Safety Program of the NASA Aeronautics Research Mission Directorate (ARMD), and in part by NASA contract NNA10DE83C. Associate Editor: S. Shieh.

The authors are with SGT Inc., NASA Ames Research Center, Moffett Field, CA 94035 USA (e-mail: ewen.denney@nasa.gov; ganesh.pai@nasa.gov).

Digital Object Identifier 10.1109/TR.2014.2335995

I. Introduction

CERTIFICATION is a core activity during the development of many safety-critical systems in which assurance must be provided that the system (and its software) will operate safely, and as intended, by demonstrating compliance with the applicable regulations to a government authority.

In aviation, regulations such as the federal aviation regulations, standards and guidelines, e.g., ARP 4761 [1], recommend or prescribe the means for compliance, offering guidance on best practice engineering methods, analysis techniques, and assurance processes.

For software in particular, assurance largely involves an appeal to the satisfaction of a set of process objectives set forth in guidance documents such as DO-178C [2]. A fundamental limitation, however, is that a correlation has not been demonstrated between the application of best practice methods and processes, and the achievement of a specified level of safety integrity [3]. Furthermore, the rationale connecting the recommended assurance processes to system safety is largely implicit [4]. Consequently goals-based safety arguments, also known as *safety cases*, are increasingly being considered in emerging standards and national guidelines as an alternative means to show that critical systems are acceptably safe, e.g., the ISO 26262 functional safety standard for automotive systems [5], and the U.S. Food and Drug Administration draft guidance on the production of infusion pump systems [6].

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" [7]. Safety cases[1] are among the state of the art in technologies for safety management, with their development already being a common practice for the certification of defense, rail, and nuclear systems [8]. Their use has also emerged in aviation, e.g., in the safety of flight operations [9]. We can document a safety case in a variety of ways, including as formatted reports containing a combination of textual descriptions and diagrams. Graphical notations, such as the Claims-Argument-Evidence notation [10], and the Goal Structuring Notation (GSN) [11], have emerged over the past decade providing a graphical syntax to document the argument structure embodying a safety case. In our work, and in this paper, we have used the (GSN).

For the most part, argument structures are constructed manually, making the safety case development process more time consuming, error prone, and expensive. They also quickly become difficult to manage, comprehend, and evaluate during iterative systems and software development due to the volume of

[1]Although we use the terms argument structure and safety case interchangeably in this paper, a safety case is the argument structure *together* with all the documents to which it refers.

information that must be assimilated. For instance, the preliminary safety case for co-operative airport surface surveillance operations [12] is about 200 pages, and is expected to grow as the interim and operational safety cases are created. Due to the increased number of requirements that software must satisfy, safety cases that reason about software details specifically are likely to grow super-linearly with the size of the underlying software.

Upon considering the prevalence and importance of domain-specific content, this problem is compounded. Safety case size, the diversity of its content, and the level of assurance, especially pose challenges for aviation systems. To reflect a comprehensive safety assessment, aviation safety cases need to reconcile heterogeneous content, such as physical formulae from the system design, maintenance procedures during system operation, and software. Especially for software, the heterogeneity of evidence, context, and assumptions is evident when considering non-formal sources such as simulation runs, unit tests, artifacts from formal verification such as the results of model-checking, proofs of correctness, and the variety of tools used to generate them. Each of these sources provides a different level of assurance, which in turn affects the trustworthiness of the evidence in the safety case, and consequently the confidence that can be justifiably placed in the top-level claim. For instance, we can use evidence from formal verification, such as proofs of correctness, to raise the level of assurance that can be claimed for mathematical and safety-critical software, typically found in aviation systems. In fact, compared to other non-formal sources of evidence, proofs are acknowledged to provide the highest level of such assurance.

Thus, there is a need both for increased automation in the creation and assembly of safety arguments from heterogeneous sources, and for integrating formal reasoning along with comparatively non-formal reasoning to improve the level of assurance that can be provided. Towards this end, our paper makes the following contributions.

1. We give a methodology for automatically assembling a safety case, integrating system safety analysis and the application of formal reasoning to software. We illustrate our approach by applying it to a real aviation system, the Swift Unmanned Aircraft System (UAS) being developed at NASA Ames.
2. Our approach highlights the heterogeneity of safety-relevant information. We characterize this inherent diversity, identifying a varied 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 elements are integrated into a single argument structure for the example system.
3. Specifically, our method automatically combines a manually created, aircraft-level safety case fragment derived from system safety analysis, with an auto-generated, lower-level safety case derived from formally verifying the safety-related properties of the autopilot software.
4. We give fragments of the resulting end-to-end safety case, i.e., a safety argument containing safety claims made at the system level justified by, and linked to, low-level software implementation details. In particular, we explicitly highlight the contribution of software assurance to system safety assurance. To our knowledge, few if any such examples [13], [14] exist in practice, or in the literature.

II. Related Work

This paper builds upon, and substantially extends, our previous work [13], [14], [33] in integrating formal and non-formal methods to create aviation safety cases. In particular, this paper describes the underlying safety assurance methodology in greater detail; additionally, using an argument architecture and its modular realization, the paper more extensively highlights our structured approach to creating the assurance argument for the Swift UAS. We also consider more elements in the overall safety assurance, transitioning the preliminary safety case in our previous work to an interim safety case. The argument presented in this paper also better illustrates the logical dependencies between heterogeneous items of safety information, e.g., the logical dependency between the claims related to software, the reliability of input sensors, calibration data, and the results of reviews (See Section VII.D). In [34], first-order logic has been applied to derive safety cases using formal methods in model-based development; whereas in [35], a lightweight method is given for automating the assembly of safety arguments using safety data from the early stages of development, i.e., hazards and requirements.

Existing approaches for systematically developing safety arguments in general [20], [36], and specifically for software [37], are based on goals-based argumentation frameworks, which indicate the role of safety processes and evidence selection in creating the system or software safety case. Our work is different from existing approaches in the notion of argument assembly with which we not only automatically generate certain argument fragments, but also automatically assemble them in the appropriate locations of the overall system-level argument.

Formal methods have also been applied in other incarnations of goal-based argumentation, so-called *assurance cases* [38], and *dependability cases* [39], while the role of diversity and uncertainty in safety and dependability cases has been addressed in [19], [40], and [41].

III. The Swift Unmanned Aircraft System

A. System Description

The Swift Unmanned Aircraft System (UAS), under development at NASA Ames, is our running example to illustrate our approach for system and software safety assurance. The UAS consists of the electric Swift Unmanned Aircraft (UA), a primary and secondary Ground Control Station (GCS), and communication links.

The UA can fly autonomously by following a *mission*, i.e., a pre-programmed or uploaded nominal flight plan. Effectively, this is a sequence of commands determining a set of waypoints from take-off to landing. A pilot on the ground can also control the UA during take-off and landing, or can intercept the flight plan at any time. Additionally, the UA can be operated semi-autonomously in the pilot-in-control and computer-in-control modes.

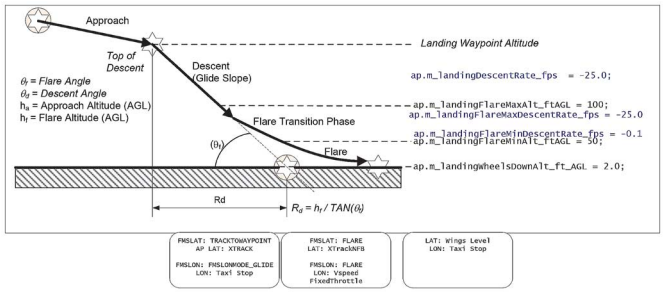

Fig. 1. LAND command induced landing profile for the Swift UAS [13].

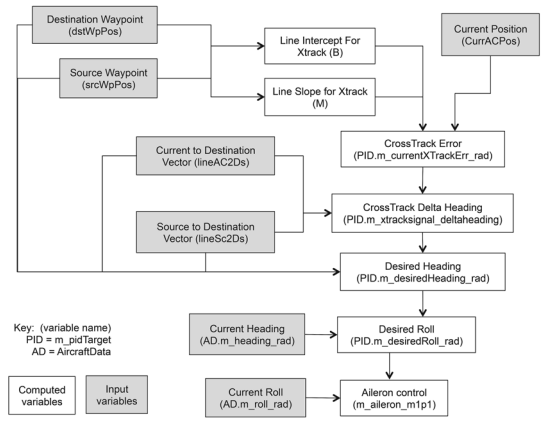

Fig. 2. Logical dependencies in aileron control computation [13].

B. Software System Architecture

The software system architecture is layered. A physics library and the Reflection Virtual Machine (RVM) execute atop the base layer, which is the Windows XP Embedded operating system, and which itself runs on the UA hardware. The RVM is a multi-component, event-driven, real-time, configurable software framework supporting the operation of several loosely-coupled software modules.

The flight software is itself one such module running on the RVM, as a collection of interconnected modules, one of which is the autopilot. The two main sub-modules of the autopilot are the Flight Management System (FMS), and the Autopilot Controller (AP), each of which regulate the aircraft control surfaces (*ailerons* and *elevators*), and in turn the aircraft movement, i.e., forward motion, rotation around the lateral (pitch), longitudinal (roll), and vertical (yaw) axes. In addition to these features, the software contains script files describing mission configurations such as flight plans, and mission-specific parameters.

C. Operation and Control

A flight plan consists of a sequence of pre-programmed or uploaded commands [14]. Based on the current state and the command being evaluated, the control system periodically updates the control surface positions. The relevant calculations occur in either of the two directional modes, each of which in turn has several cases of relevant computations.

Fig. 1 illustrates the Swift UAS landing profile when a LAND command [14] is issued. In the autopilot software, the phases shown appear as mode transitions. During the flight path, the flight software only invokes some of the cases for the FMS lateral and longitudinal modes. The software defines the transition criteria, using system parameters that are set via scripts. Based on the issued command, the FMS determines an appropriate mode to be set in the AP, and evaluates different cases of calculations, e.g., for the LAND command, a specific mode is set in the FMS so as to update the lateral control surfaces (aileron). Proportional-Integral-Derivative (PID) controllers (loops) perform the value computations; as such, each PID loop will affect either a lateral, longitudinal, or speed control surface. Its result is a value that will be output to (or used in a calculation of the eventual output to) the actuator of a single control surface.

Thus, to adjust the aileron, the flight software determines the change to the current heading, after which it derives a new heading from the aircraft state (i.e., its current position, its source, and destination waypoints). We further illustrate this adjustment next, giving a single computation sequence through the FMS and AP modules, executed under specific mode and command conditions, the outcome of which is the aileron value, and a consequent change in the aircraft heading. For more details on the operation and control, refer to [14].

D. Low-Level Computations

To realize the adjusted values of the aileron control surface when a LAND command is given, a sequence of low-level computations, specified as mathematical definitions, are executed in software (Fig. 2). Effectively, PID loops derive a value in several of the computation steps, which is then used in conjunction with the aircraft state in the subsequent computation step. For the aileron, the *cross-track*, *heading*, *roll angle*, and the *aileron* PID loops are relevant.

Specifically, first, geometric calculations determine the current UA position and heading, relative to the *source to destination vector*, i.e., the imaginary line connecting the source and the destination waypoints (shown in Fig. 2, as the input variables CurrACPos, and AD.m_heading_rad). The distance of the UA from this line is the *crosstrack error*, which is the parameter passed to the PID loop that determines the heading change needed, i.e., the *delta heading*, to reach the destination waypoint. The *desired heading* computation then takes into account both the source to destination vector and the delta heading. From the difference in the current and desired headings, i.e., the error, a new *desired roll* can be determined. To initiate the change in the roll (and hence the heading), the aileron is to be moved by an amount to produce the desired roll. The aileron PID loop which computes this value takes the *roll error*, i.e., the difference between the current and desired aircraft roll, as a parameter. The variable m_aileron_m1p1 contains the computed roll value, to be routed to the aileron actuator through the RVM.

As is often the case with numerical calculations, the code is not particularly complex, but uses a variety of mathematical definitions with various side conditions, and calls to various library functions. To verify this code, we need to establish the properties of interest, give mathematical definitions and equations corresponding to the steps of the computations, specify the library

```
definition(calc_error
, ['computing the difference between current and desired values in', T]
, Y-Z
, [Y::current(T), Z::desired(T)]
, _::error(T)
).

function(initial_heading
, ['initial heading from ', XPos, 'to', YPos]
, cglGeom_CalculateHeadingAngle_rad(XN, XE, YN, YE)
, [XPos=(XN,XE)::pos(ne), YPos=(YN,YE)::pos(ne)]
, _::current(heading)
).

function(line_slope_intercept
, ['LineSlopeIntercept']
, cglGeom_CalculateLineSlopeIntercept(LineM, LineB, SrcWpPos, DstWpPos)
, [SrcWpPos::pos(ne), DstWpPos::pos(ne)]
, (LineM,LineB)::line_slope
).

function(crosstrack_deltaheading
, ['computing the cross track delta heading']
,  'm_pid_CrossTrackErr2Heading->Update1'(XTE)
, [XTE::error(xtrack)]
, _::desired(delta(heading))
).

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

Fig. 3. Math and function schemas (excerpt).

functions, and axiomatize the domain. Axioms will typically express the preservation of properties under geometric translation, and bounds on physical constants.

Subsequently, we will use domain theory to verify this sequence of computations (Section VI.B), automatically construct a fragment of the safety case from the verification (Section VI.C), and then combine the resulting fragment with a manually created fragment (Section VII.C).

IV. Domain Theory

Section III.D described the sequence of calculations used to compute the aileron control variable from several inputs drawn from the current state of the UA and its flight plan. We take correctness of this computation to mean that the code implements a mathematical specification of a given property. This specification will be expressed (Section VI.B) as a formal requirement, and proven using a domain theory mainly formalizing geometric and navigational equations.

The domain theory consists of annotation and function schemas (Fig. 3), and axioms (omitted here, see [13], [14]) that relate to aspects of the software in question. In relation to the safety assurance process, the domain theory forms part of the context in which the formalization and subsequent verification will occur (Fig. 6). These items of context form part of the input during argument development and its activities (See Fig. 5), as well as part of the assurance argument produced (See Fig. 16 for an example).

An annotation schema declaratively captures all information that is required to handle the verification of a programming idiom, that is, a coding pattern that solves a specific problem. In AutoCert, a schema is specified as a Prolog fact of the form `SchemaType(Name, Explanation, Pattern, DepVars, DefVar).`

$$
\begin{array}{rcl}
\mathit{Property} & ::= & \mathit{Quantity} \mid \mathit{DerivedQuantity} \\
\mathit{Quantity} & ::= & \mathit{ScalarQuantity} \mid \mathit{VectorQuantity} \\
\mathit{ScalarQuantity} & ::= & \text{heading} \mid \text{roll} \mid \text{aileron} \mid \text{xtrack} \mid \text{pitch} \mid \text{elevator} \mid \\
 & & \text{line\_slope} \mid \ldots \\
\mathit{VectorQuantity} & ::= & \text{pos}(\mathit{Frame}) \mid \ldots \\
\mathit{Frame} & ::= & \text{ba} \mid \text{lla} \mid \text{ne} \mid \text{north} \mid \text{east} \mid \ldots \\
\mathit{DerivedQuantity} & ::= & \text{delta}(\mathit{Quantity}) \mid \text{desired}(\mathit{Quantity}) \mid \\
 & & \text{current}(\mathit{Quantity}) \mid \text{error}(\mathit{Quantity})
\end{array}
$$

Fig. 4. Grammar of domain-specific terms.

The schema *name* is used for identification purposes. The *explanation* describes the schema in natural language terms. The *pattern* describes the underlying idiom. The next two slots contain the schema's logically *dependent* and *defined* variables (which can be omitted if the defined variable does not appear in the pattern), together with their respective properties. The schema *type* is either `function` (used for function calls) or `definition` (used for other expressions).

Pre-, and post-conditions are expressed as properties on the schema's (logically) dependent, and defined variables, respectively. For example, `Y::current(T)` means that the variable `Y` has property `current(T)`, that is, it represents the current value of some quantity `T`. Properties represent various scalar and vector physical quantities, or derived measures based on those. We define vector quantities with respect to a frame of reference (used to represent different coordinate systems of various dimensions). Fig. 4 gives the grammar of properties of the domain theory.

Fig. 3 shows an excerpt of example schemas. Here, `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`. We compile schemas into low-level patterns that match against the code (see [15] for details). After matching against the corresponding code fragment, and inserting pre- and post-conditions, the tool recursively searches for definitions of `Y` and `Z`, and so on. Similarly, in Fig. 3, `initial_heading` matches against code that calls a library function to compute the heading in radians, where `Dst` and `Src` represent the aircraft position in the `ne` frame. Also, in Fig. 3, `aileron_out` specifies a PID function where the output is the desired value of the aileron, given the error on the roll as input. We can extend specifications, schemas, and axioms with additional non-logical information [16] such as justifications and contexts (not shown here).

Axioms are first-order formulas that provide the necessary facts used to prove that the generated Verification Conditions (VCs) are correct. Axioms can define mathematical properties, and express assumptions about both the real world and the system under verification, in this case the code. They mainly give the definition of geometric properties, and involve adjustments of orientations in radians through different quadrants based upon logical dependencies on the aircraft state and the desired value needed to accomplish the movement of the control surface. Many of these axioms describe how a property is

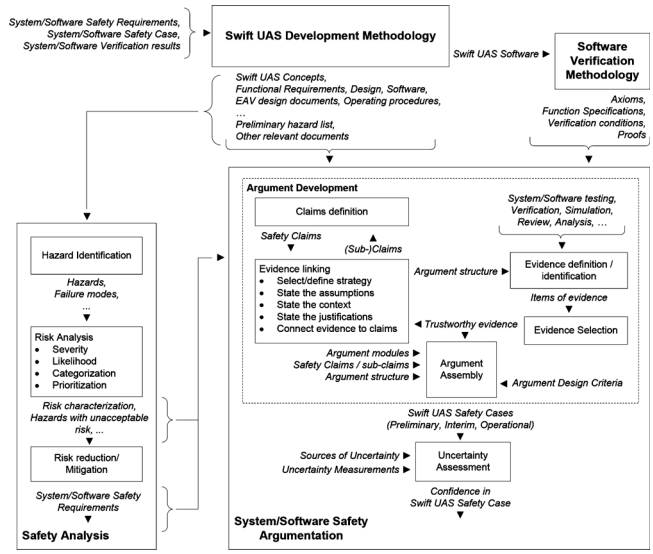

Fig. 5. Safety assurance methodology showing the data flow between the processes for safety analysis, safety argumentation, system development, and software verification.

preserved under addition or subtraction of 2π. Others describe the validity of reversing the sign of a value while maintaining a property.

Like the schemas, axioms and verification conditions also use properties that denote geometric and navigational concepts, including physical quantities such as `heading` and `roll`, control surface settings like `aileron` and `elevator`, and frames of reference such as `lla` (lat-long-alt) and `ne` (North-East).

Much of this domain theory will be reflected in the resulting complete safety case. For instance, the schemas give rise to the goals (and subgoals) of the safety case. The schemas are also used to generate verification conditions. The external functions in the code base are represented in AUTOCERT as function schemas. These are represented as nodes in the safety case. Finally, the verification conditions are proved by a subset of the axioms, and the proof is also represented as an evidence node in the safety case.

V. SAFETY ASSURANCE METHODOLOGY

Fig. 5 shows our overall safety assurance methodology as a data flow among the different processes and activities applicable during the development of the Swift UAS. Note that the figure mainly shows some of the key steps and data relevant for this paper. Additionally, neither the iterative and phased nature of the involved activities nor the feedback between the different processes have been shown.

We emphasize that safety analysis and argumentation (Fig. 5) are not post-development activities. Rather, they are performed in conjunction with system development and verification, so that safety considerations are addressed from the outset. In particular, safety analysis activities start from concept definition, and continue through requirements development, design, implementation, verification, and operation. The data from the later stages of development and operation serve to refine and validate earlier analyses.

In general, safety analysis drives safety case development. Safety analysis results are, therefore, at the core of the set of heterogeneous information available to create the system safety case. Thus, after performing early-stage safety analysis, we use the results, which include the identified high-level hazards, the broad mitigation mechanisms, and safety requirements, to create a (skeleton of a) high-level, preliminary safety case, by following the activities for argument development (Fig. 5). As development progresses, we repeat the safety analysis to identify lower-level hazards, and the design decisions required to mitigate those hazards. In turn, we use these data to refine the preliminary safety case into an interim safety case. During these steps of safety case refinement, additional constraints on the evidence required for substantiation become more clear, e.g., from the safety standards used, the development activities, and from the safety analysis of the system and its constituent elements. Additionally, during safety case refinement, we obtain some evidence from design and implementation activities. Substantiating evidence can be gathered from verification; and as additional evidence is obtained from operation, the assumptions made during the development can be validated (or invalidated), so as to update both the safety analysis, and the corresponding operational safety case.

A. System Safety Analysis

The system safety process that is being used in the ongoing development of the Swift UAS, is based on the framework of a safety risk management plan, such as [17] or [18]. The process includes safety considerations into system design at an early stage through hazard identification, risk analysis, and risk management (labeled as safety analysis in Fig. 5). Hazard identification and risk analysis are fundamental activities in safety engineering involving, in brief, (a) identifying those situations or conditions relevant to the system which, if left uncontrolled, have the potential to result in an undesirable loss event; and (b) characterizing the consequences, severity, and likelihood of such situations and conditions. Broadly, in risk management we use the results of risk analysis to prioritize and mitigate risks. We derive the system safety requirements during the early stages of concept development and requirements formulation. The lower-level safety requirements, including those related to software safety, are identified later when the system requirements are better understood, and lower-level details are being designed.

As input to the steps of safety analysis, we use a variety of data that give insight into potential hazards. During the early stages of system development, these data include, but are not limited to, the concept of operations, available design documentation for legacy systems (if applicable), and previously identified hazards (often captured in a preliminary hazard list). In general, additional heterogeneous sources include the following.

1) *Procedural, development, and safety standards* imposing design and safety constraints on the system.
2) *Procedures* (distinct from procedural standards) describing operations, e.g., for maintenance on range, before, during, and after flights, including maintenance, and the roles played by the flight team members.

3) *Mathematical theory*, e.g., the theory of aerodynamic stability and control are used to derive parameters, included in the control laws, that govern the safe operation of aircraft.
4) *Assumptions*, i.e., simplifying assumptions such as decoupled dynamics: linear independence of latitudinal and longitudinal models, are sometimes made in autopilot design, which break down when large angles of attack are considered; such assumptions have significant safety implications.
5) *Vehicle flight logs*, e.g., which show the absence of mishaps for qualification testing, and also the system behavior under specific flight conditions.
6) *Calibration experiments* such as those used to calibrate sensors; the results of such experiments eventually appear in the control system software.
7) *Hardware tests*, e.g., static load tests used for determining actuator sizes, tolerances, endurance, etc.
8) *Aircraft models*, e.g., geometric, and three-dimensional CAD models, computational fluid dynamics models, among others, are used for deriving control parameters, and for visualizing safety aspects of the avionics.
9) *Data-sheets* providing component parameters and specifications used in the aircraft.
10) *Simulations* which progressively evaluate the hardware, software and eventually 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.

We believe that the main value of characterizing such heterogeneous data is to manage the wider context of safety, e.g., to identify and manage hazards arising from system interactions. In particular, although goals-based argumentation is largely product focused, safety implications also arise from other related sources, e.g., process and procedural deviations during operation, incorrect or implicit assumptions, etc.

Once hazards have been identified, we define mitigation measures to reduce risk to acceptable levels. One specific outcome of the risk reduction and mitigation step is requirements on system safety. These requirements take several forms, including constraints on the design, guidelines, and procedures for maintenance, operation, etc. See Section VI.A for an example.

B. Safety Argumentation

The general idea underlying safety argumentation is to create a structured safety case to systematically justify safety claims, for instance by justifying that all identified hazards have been eliminated or mitigated, such that mishap risk has been reduced to an acceptable level. As shown in Fig. 5, safety argumentation comprises the activities of argument development (the main focus of this paper) and uncertainty assessment [19] (out of scope for this paper).

The main activities in argument development are *claims definition*, *evidence definition and identification*, *evidence selection*, *evidence linking*, and *argument assembly*, of which the first four are adapted from the *six-step method* for safety case construction [11], [20]. In particular, we consider the activity of argument assembly, which is where our approach deviates from existing methodologies [10], [11] for safety argumentation, including the six-step method. This activity reflects the notion of *assembling* the data produced from the remaining activities to create a safety case (in our example, fragments of argument structures for the Swift UAS) containing safety claims and the supporting evidence, linked through an explicit chain of reasoning. More specifically, it reflects the notion of assembling lower-level auto-generated argument fragments with higher-level manually constructed argument fragments. This combination of *bottom-up* and *top-down* argument construction is a novelty of our approach that further distinguishes it from other existing approaches.

The activity of argument assembly accounts for (i) the inclusion of *argument design criteria* such as maintainability, compliance with safety principles, reducing the cost of re-certification, modularity, composition of arguments, etc.; and (ii) automation.

Argument design criteria represent trade-offs made in the *argument architecture*, which affect the overall structure and organization of the elements of the argument. We envisage automation in argument assembly to include the assembly and generation of

1) argument fragments from fundamental elements of an argument, such as claims, evidence, and reasoning;
2) argument modules created using manual, automatic, and semi-automatic means [13]; and
3) heterogenous data in the overall safety argument.

Specifically in this paper, argument assembly involves, in part, the automatic creation of an argument structure from formal software verification (implemented as model transformations), and its automatic inclusion into a manually created higher-level argument structure. We describe the specifics of the relevant transformations in Section VI.C.

Safety argumentation, which is phased with system development, is applied starting at the level of the system in the same way as the system safety process, and then repeated at the software level. Consequently, the safety case produced itself evolves with system development. Thus, similar to [21], we can define a *preliminary*, *interim*, and *operational* safety case to reflect the inclusion of specific artifacts at different points in the system lifecycle. Alternatively, we can also define finer grained versions, e.g., at the different milestones defined in the plan for system certification.

The argument structure of the Swift UAS, which we present subsequently in this paper, embodies a fragment of an interim safety case.

C. Software Verification

As mentioned earlier, we use our software verification methodology (Fig. 6) to create the lower levels of the software safety argument. Fig. 6 shows some of the data (boxes) and verification activities (annotated arrows) involved, as well as the connections to the wider system safety process (dotted arrows).

TABLE I
EXCERPT FROM SWIFT UAS HAZARD ANALYSIS

ID	HAZARD DESCRIPTION **Operating Phase**: Descent **Sub-phase**: Approach	LIKELIHOOD	SEVERITY	RISK LEVEL	SAFETY REQUIREMENT
AVCS_012	Flight management system (FMS) Failure	Remote	Major	Medium	[FSP_AVCS_004] When FMS failure is detected, it is always the case that failsafe autopilot eventually takes control within a specified time duration
AVCS_013	AP Failure	Probable	Major	High	[FSP_AVCS_003] When AP failure is detected, it is always the case that failsafe autopilot eventually takes control within a specified time duration
AVCS_015	Autopilot controller module failure	Remote	Hazardous	High	[A1] Commands must be interpreted correctly [A2] No command shall make the autopilot execute an unsafe maneuver. [SAFR_AVCS_001] During descent, the Autopilot shall not produce control surface variable values which are outside their specified ranges.
FLTCS_01	The control surface positions are outside the range of their required values for a safe landing on approach	Remote	Hazardous	High	[SAFR_FLTCS_001] Control surface positions shall be within their specified ranges during descent.

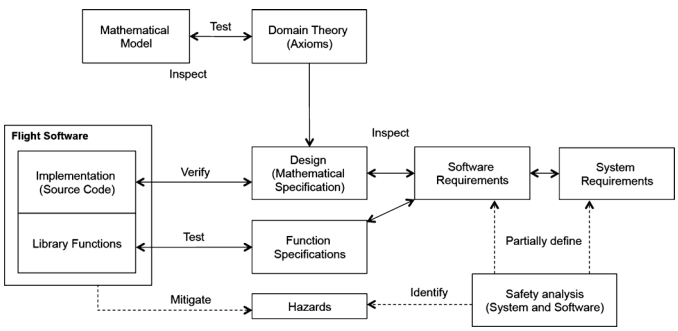

Fig. 6. Software verification methodology [13].

To verify the flight software in the Swift UAS, we formally verify 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 [22], whereas testing and verification using other tools is deferred to future work.

The specification formalizes software requirements which, in turn, we derive from system requirements during the safety analysis. A logical domain theory (i.e., a set of axioms and function specifications; also see Section IV) provides the context for formal verification. Axioms can be subjected to increasing levels of scrutiny, going from simply assuming their validity, to inspections, up to testing them against a computational model which, itself, is inspected [23].

VI. METHODOLOGY APPLICATION

A. Hazard Analysis

During hazard identification, Preliminary Hazard Analysis (PHA), and Functional Hazard Analysis (FHA) for the Swift UAS, we systematically identified and documented the known hazards, and brainstormed for new hazards, in close cooperation with the Swift UAS engineering team. We used documents related to the concept of operations, preliminary design, operating procedures, as well as other heterogeneous information (as identified in Section V.A). We also applied Failure Modes and Effects Analysis (FMEA) as part of bottom-up reasoning for hazard identification.

The hazard categories identified include environmental hazards (e.g., unexpected air traffic in the range), energy release hazards (e.g., thermal runaway of the onboard lithium polymer battery), failure hazards (e.g., subsystem failures from unreliable, unavailable or incorrectly constructed components), deviations from procedures (e.g., incorrect application of pre-flight checklists), operational hazards (e.g., unanticipated pitch down during landing), as well as interactions (e.g., miscommunication between operator and air traffic control).

Table I shows an excerpt of the hazard analysis, for the *descent* phase of the UA induced by the `LAND` command (Fig. 1). We show (a small subset of) some of the relevant failure hazards in the avionics software (specifically, the autopilot), as well as a hazard related to the flight control surfaces, which pose a safety risk during descent. Each of the failure hazards identified here may be considered as the aggregation of the various failure modes of the corresponding component, although we do not show the FMEA in this paper. Several of the columns in the actual analysis, such as the *effects* on the immediate sub-systems and the wider system, *potential causes*, *mitigation measures and corrective actions*, etc., have been hidden here, to make the table readable. The hazard risk levels are determined qualitatively as a combination of the hazard severity and likelihoods, by applying a risk classification matrix, e.g., as in [18]. The table also shows the consequent definition of (some of) the relevant safety requirements which, when correctly implemented, are expected to mitigate the hazards and reduce risk.

Some of the requirements shown are applicable at the system level, while others are relevant for the software. There are also relations between (some) system-level safety requirements

The autopilot is correct if
- SR–(1): The system is properly initialized
  - (a) aircraft state information is properly received from the sensors
  - (b) the current, previous, and next waypoints are properly defined
  - (c) the FMS object is properly initialized
  - (d) the AP object is properly initialized
  - (e) Input and Output variables are properly routed via the reflection systems scripts
  - (f) Parameter data is properly initialized.
- SR–(2): The FMS system is correct
  - (a) The FMS system and AP system are properly initialized by the autopilot
  - (b) The FMS system properly interprets all commands
  - (c) The FMS system properly transitions between commands and waypoints
  - (d) UpdateCommandTransitions is correct
    - i. The FMS system properly transitions into a COMMAND_LAND command
      - A. The FMS system properly transitions between phases when the command is COMMAND_LAND
      - B. the transition criteria for each phase is properly interpreted
      - C. the FMS system properly transitions between modes representing the different phases of the landing command
      - D. the FMS system properly sets the new waypoint for approach
    - ii. The FMS system properly transitions into and executes all other commands
  - (e) The FMS system properly sets FMS Lateral and Longitudinal modes for the corresponding command
    - i. setNewFMSWaypoint is correct
  - (f) The FMS system properly sets AP Lateral and Longitudinal modes
    - i. SetFMSLatMode is correct
    - ii. SetFMSLonMode is correct
  - (g) The FMS system properly updates state variables in the AP system
    - i. UpdateFMSLatMode is correct
    - ii. UpdateFMSLonMode is correct
- SR–(3): The AP system creates correct output for all aircraft control surfaces
  - (a) The autopilot correctly initializes the AP object
  - (b) Each PID loop
    - i. receives correct aircraft state information
    - ii. receives correct current and previous waypoint information
  - (c) PID controller objects are properly initialized
  - (d) PID controller updates are correct for each aircraft controller surface
    - i. PID controller updates are correct for the Aileron
    - ii. PID controller updates are correct for the Elevator
- SR–(4): The CGL mathematics libraries provide correct results
- SR–(5): The autopilot correctly computes the actuator outputs

Fig. 7. Breakdown of autopilot functionality, giving low-level functional requirements related to software safety requirements.

and software (safety) requirements, due to the contribution of software to the system hazard. For instance, in Table I, the software safety requirement for the autopilot to produce the correct values of the control surfaces during descent (SAFR_AVCS_001) can be derived from the system safety requirement for the control surfaces to be within their specified ranges during descent (SAFR_FLTCS_001). In fact, the former is also a functional requirement of the autopilot (SR-3(d) in Fig. 7, which lists an informal, high-level hierarchy of functional requirements for the autopilot module).

The autopilot functional requirements must be satisfied for the correct and safe operation of the autopilot. This decomposition, based on the structure of the code defining the autopilot module and its subsystems, was created by a software engineer familiar with the autopilot software. We used this decomposition as a guide when creating the autopilot software safety argument (described subsequently in this paper, in Section VII.C.3).

Roughly speaking, the correctness of the autopilot module is logically dependent on showing the correctness of the underlying subsystems, i.e., the FMS and AP, which the autopilot depends upon and initializes. The FMS interprets the list of commands that are supplied, and the transition between these commands must be correctly executed. The AP must then correctly compute the output for each aircraft control surface (based upon the new modes set by the FMS in the previous step). The correctness of the autopilot requires that each subsystem properly communicate any state transitions, and then operate appropriately on that resulting state.

B. Formal Verification

In the previous section, we derived several low-level requirements. Following our running example, we now concentrate on the aileron and elevator control variable calculations, and formalize and verify them against the software. Fig. 8 shows

```
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).
```

Fig. 8. Specification (excerpt): formal assumptions and requirements [13].

a specification fragment consisting of assumptions about the current aircraft state and flight plan, and requirements on signals to the control surfaces. Note that the formal specification (Fig. 8) only concerns a small part of the behavior covered by the larger informal specification (Fig. 7). More specifically, the requirements SR-3(d)-i and SR-3(d)-ii, in Fig. 7, are the requirements formalized in Fig. 8, while the informal assumptions are implicit. Both the informal assumptions and requirements, and their formalized equivalents will eventually appear as assumptions, and claims, respectively, in the assurance argument (Fig. 15, and Fig. 16).

`currACPos` is the current aircraft position in the *North-East* frame of reference, whereas `srcWpPos`, and `dstWpPos` are the current, and next waypoints, respectively, in the flight plan. The purpose behind the axioms, schemas, and domain theory in general is to ultimately prove properties of the code via verification conditions. Given these assumptions on the aircraft state, we must show that the code that implements the *descent* phase of the `LAND` command (Fig. 1) 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 the annotation schemas (Fig. 3).

AUTOCERT then applies a verification condition generator, which uses the annotations and function specifications to generate a set of VCs, logical conjectures which are sent to a suite of Automated Theorem Provers (ATPs) along with the axioms. Most VCs conjecture that the regulation of a variable within some bounds maintains a given property. In Fig. 8, for the requirement $output \rightarrow m\_aileron\_m1p1 :: desired(aileron)$, 51 verification conditions were generated. Of these 51, most conjectured that the regulation of a variable maintained a given property. The regulation involved the quadrant adjustments of radian values or the reversing of the sign based on the current aircraft state. All 51 were proved using a suite of 5 ATPs. For the second requirement $output \rightarrow m\_elevator\_m1p1 :: desired(elevator)$, 13 VCs were generated. In the same way as for the first requirement, the VCs correspond to the maintenance of a given property under regulation through quadrant adjustments or sign reversals. All 13 of the verification conditions were verified using the same suite of 5 provers.

In effect, AUTOCERT checks the implementation against a formal specification in several steps, ensuring the following points. (a) Individual code fragments correctly implement mathematical equations because, in general, there can be a large semantic gap between mathematics and implementation. Concepts can be fused together or separated throughout the code. (b) The sequence of implemented equations links together correctly to meet a higher-level requirement. (c) Sufficient assumptions are given, and are necessary (i.e., are actually used).

Specifications consist of low-level assumptions and requirements. An example of a low-level (informal) requirement might be that the code shall compute the rotation matrix. The corresponding formal requirement would state that the code must implement a particular mathematical concept. The formal verification of this requirement then traces that mathematical concept to a sequence of lines of code, and reveals its logical dependency on other requirements and assumptions. Such low-level requirements often need to be verified during reviews, and can require the tracing and decomposition details provided by the formal analysis.

Each branch of the formal verification and corresponding auto-generated argument shows the logical slice of the system dataflow that is relevant to meeting that requirement. The formal argument has two kinds of verification steps: that a mathematical concept is implemented, and logically depends on other mathematical concepts or assumptions; and a side condition, i.e., a verification condition (VC), holds. The former are interesting, but the latter less so, and could for most purposes be safely hidden.

In fact, there are two levels of *proof*. AUTOCERT is an inference tool that matches implementation in the code to mathematical concepts, thus allowing formal requirements to be decomposed, and to be traced to code. It is the upper-level proof (the tracing relations comprising decompositions and logical dependencies that establish that lower-level requirements entail higher-level requirements) that is represented graphically in the argument structure. However, the (lower-level) proofs of VCs using automated theorem provers (external to AUTOCERT) are considered evidence.

C. Transformation

In principle, there are two ways in which a formal method can be integrated with the construction of a safety case: (i) the output of AUTOCERT can be transformed into a safety case fragment, or (ii) safety case fragments can be transformed into formal specifications that are then input to AUTOCERT. We consider these two cases, in turn.

1) From Formal Proofs to Argument Structures: Recall that Fig. 2 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. AUTOCERT generates a document (in XML) with information describing the formal verification of requirements. The core of this document is the chain of information relating requirements back to assumptions. Each step in the document is described by (i) an annotation schema for the definition of a program variable, (ii) the associated VCs that must be shown for the correctness of that definition, and (iii) the variables on which that variable, in turn, logically depends.

The goals (and subgoals) of the argument structure have been derived from the applied annotation schemas. The subgoals correspond to the schema's logically dependent variables, and the VCs related to each goal. An argument for a VC is a proof, generated using a subset of the axioms, which forms the evidence connected to the VC goal. We include the prover used as context. We also create goals out of function specifications from external libraries used in the software and its verification. 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.

The transformation also creates assumption nodes that relate to code variables (e.g., `x` represents altitude), which are attached to goals. In principle, assumptions can be placed anywhere in scope of the goal corresponding to the formal requirement. One choice is to attach all the formal assumptions directly to the goal requirements, which can be done by repeating the assumptions at each goal which uses them or by using cross-references for repeated nodes. Another choice is to attach assumptions to the most recent common ancestor of all formalized requirements. Our algorithm attaches them at the corresponding goal requirements.

During the process of merging the manually created argument structure with the auto-generated ones, we replace distinct nodes of the former with the tree fragments generated from AUTOCERT. Specifically, we graft the top-level goals of the latter onto the appropriate lowest-level nodes of the former. We annotate these nodes with unique comments, `autocert:id`, to relate them to a tree in the automatically created file, meaning that the goal with tag `autocert:id` is to be solved with AUTOCERT.

2) From Argument Structures to Formal Specifications: Often, an argument structure fragment may be created before the software verification is completed. Here, we can annotate nodes with `autocert:id`, where `id` identifies formal statements that are to be extracted in 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 argument structure.

In the autopilot case, the nodes representing the aileron and elevator control verification would have text representing both an informal description of the system requirement being verified, and the corresponding formal requirement, resulting in the specification in Fig. 8.

VII. SWIFT UAS SAFETY ARGUMENT

A. Goal Structuring Notation

Fig. 9 shows the main elements of the Goal Structuring Notation (GSN) [11], which we use to document the Swift UAS safety case.

An argument structure in GSN contains a top-level *goal* stating the safety claim, e.g., a given system is acceptably safe. We develop goals into sub-goals using *strategies*, and this procedure is continued until there are elementary claims that can be connected to *solutions*, i.e., the available evidence. The structure also specifies the *assumptions* made, the *justifications*

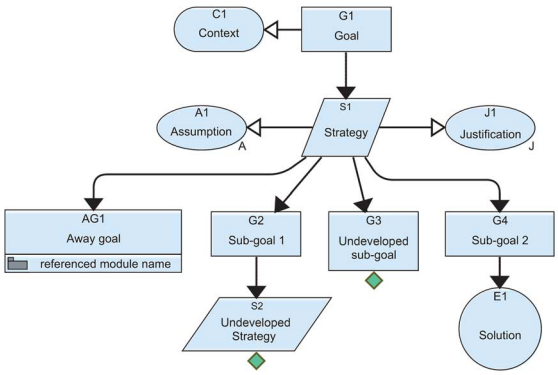

Fig. 9. Goal structuring notation (GSN) for safety case argument structures.

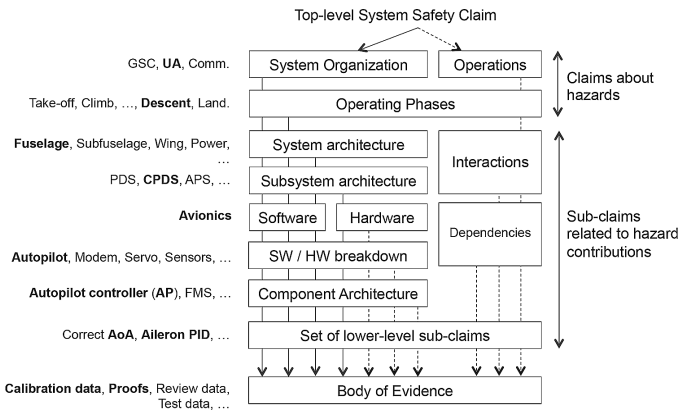

Fig. 10. Argument architecture for the Swift UAS.

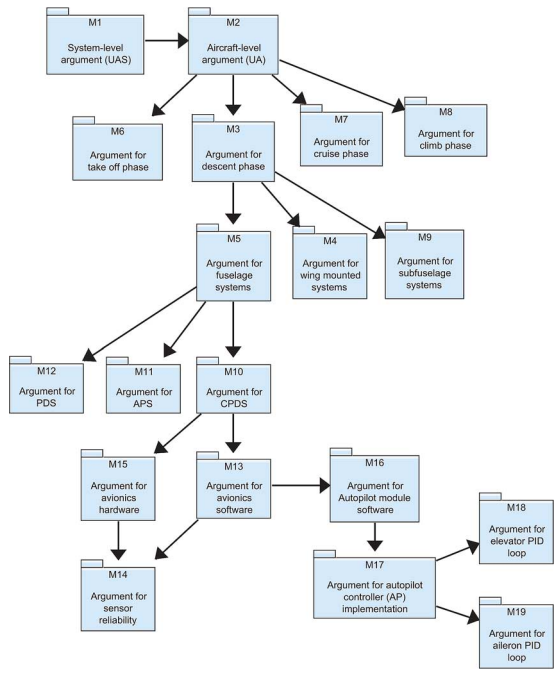

Fig. 11. Module diagram of the Swift UAS safety case, reflecting a part of the argument architecture.

if any, e.g., for the strategies used or the sub-claims created, as well as the *context* in which the claims, strategies, and solutions are valid. We link goals, strategies, and solutions using the *is-solved-by* relation (shown with the filled arrowhead), while context, assumption, and justification elements require an *in-context-of* relation (shown with the hollow arrowhead). Note that GSN nodes can also contain pointers to more detailed information, and the description of a GSN node summarizes these details.[2] For example, detailed definitions can be given externally, and linked to by a *context* node whose description could simply be an identifier.

GSN provides a graphical annotation ($\diamond$) for goals and strategies to indicate that they are *to be developed*, i.e., they are incomplete. GSN additionally contains notations for modularity, e.g., to reference elements in different modules using so-called *away* nodes. For instance, in Fig. 9, AG1 is an *away goal*, with referenced module name denoting the reference to the relevant module.

B. Argument Architecture

The argument architecture (Fig. 10) for the Swift UAS safety case describes the organization of the underlying argument to satisfy various attributes such as compliance, comprehensibility, validity, maintainability, etc. Our main concern was to assure that all hazards, and system contributions to hazards, identified in the safety analysis (Section V.A), have been acceptably managed.

We adopt a hazard-directed style of argumentation, where we develop the system safety claim by addressing hazards arising from the system organization, its interactions, and its operations. We develop the resulting claims first over the operating phases, subsequently over the hierarchical system architecture and its interactions, and eventually over lower-level components. The intent is to trace hazard mitigation through the safety argument, in part, to the behavior of system components. For instance, to assure software safety, the safety case includes explicit correctness arguments to demonstrate that software contributions to the identified system hazards are acceptable.

In this paper, we mainly present an end-to-end *slice* of the overall safety case (highlighted in **boldface** in Fig. 10, and shown in Fig. 11 as a modular organization), which corresponds to the system operation during descent (Section III.C), including the low-level computations performed in the autopilot (Section III.D). It traces the system safety claim to the identified hazards, the corresponding safety requirements (Section VI.A), and to the evidence from formal verification (Section VI.B).

The system safety claim, that the Swift UAS is acceptably 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, is made in a system-level argument. As shown, we link this claim first to an aircraft-level argument in the descent phase of operation, then through the aircraft system architecture to the avionics software, and eventually to a failure hazard of the autopilot module. We demonstrate the mitigation of this hazard through a correctness argument in which the supporting evidence comprise proofs of correctness, that the PID controllers for the aileron and elevator control surfaces produce the required values.

[2]Also note that that the node coloring shown is aesthetic and specific to our toolset. Presently, it does not provide semantics like the links do.

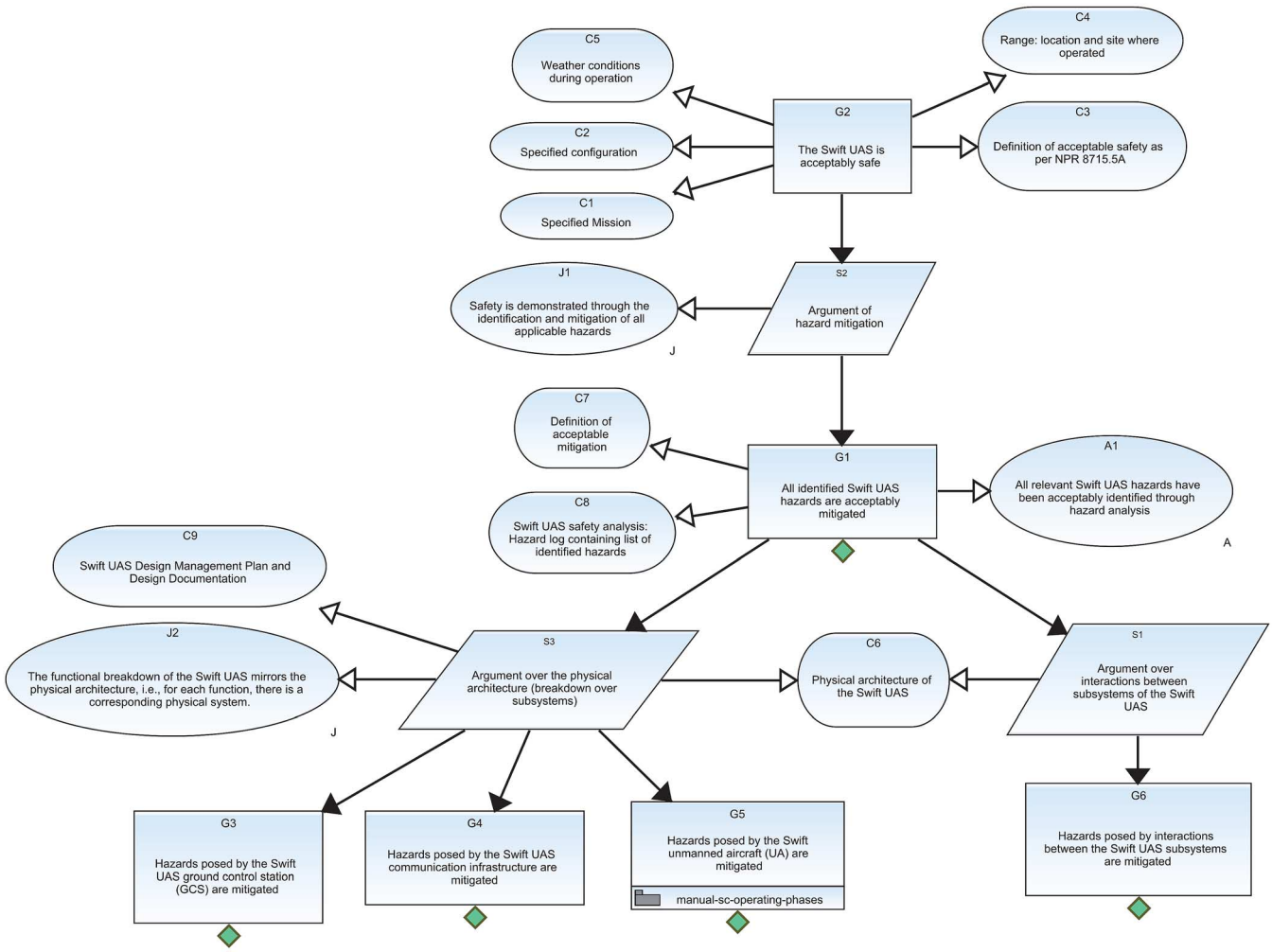

Fig. 12. System-level argument fragment for the Swift UAS.

C. Manually Created Safety Argument

The manually created argument fragment mainly concerns the airborne system, i.e., the UA. We construct this argument based on the hazard analysis (Section VI.A), and it is layered according to the argument architecture (Fig. 10).

In Fig. 11, the manually created argument fragment corresponds to the arguments contained in modules M1 through M17, whereas modules M18 and M19 contain the auto-generated fragments (described in Section VII.D). In this paper, we mainly present the system-level argument (module M1), and the argument for the software (modules M13, M16 and M17); more details on the overall argument are in [14].

1) Module M1. System-Level Argument: The system safety claim, i.e., that the Swift UAS is acceptably safe, is made in the context of a specified mission, a particular configuration, the location and site of operation, and the weather conditions during operation (Fig. 12). Acceptable safety is as defined in NASA procedural requirements for range safety, NPR 8715.5A [24], to which we also refer in the context of the safety claim. We develop this claim by argument over identified hazards, and subsequently over the physical architecture of the system and its interactions (since the argument shown is a slice, the argument over operational hazards is hidden in this view).

The result is claims about the system components, i.e., the GCS, the communication infrastructure, the airborne system (the UA), and the relevant interactions. As shown, we use an *away goal* AG1 for the claim of mitigating hazards posed by the UA, to indicate that it is developed in a different module, i.e., in the aircraft level argument (corresponding to module M2 in Fig. 11, and labeled as manual–sc–operating–phases in Fig. 12). Therein, we develop the claim of mitigating hazards posed by the UA, first by argument over the operating phases, and then in general by argument over hierarchical breakdown over the system architecture, i.e., the fuselage systems and the Common Payload Data System (CPDS), into a claim of mitigating (avionics) software failures during descent. (See [14] for more details).

2) Module M13. Avionics Software Argument: The argument structure for justifying the claim of mitigating avionics software failures during descent uses three strategies (Fig. 13): (i) S6, validity of the input to software, (ii) S7, satisfaction of software safety requirements, and (iii) S9, correctness over the (software) architectural breakdown. Note that these strategies represent one set of strategies that we chose for assurance. Other strategies, such as argument over the set of identified failures or failure modes, can also be used.

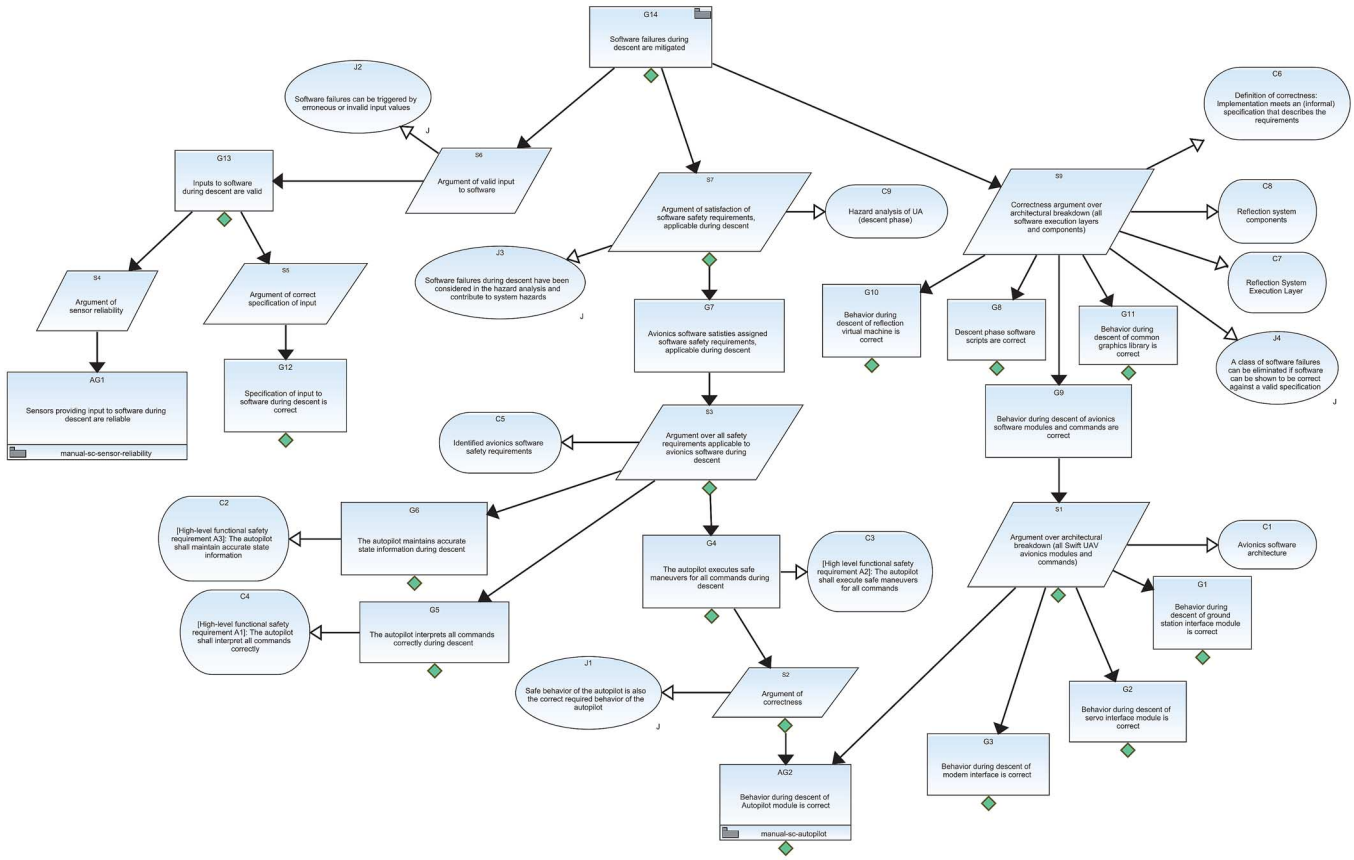

Fig. 13. Argument fragment for the avionics software in the Swift UAS.

Fig. 13 shows how the claim of mitigating avionics software failures during descent is traced to the identified high-level functional safety requirements, which appear as specific goals.

- G4: The autopilot executes safe maneuvers for all commands during descent.
- G5: The autopilot interprets all commands correctly during Descent.
- G6: The autopilot maintains accurate state information during descent.

Additionally, the argument traces the mitigation of software failures during descent to the reliability of the input sensors (shown as the away goal AG1, in Fig. 13).

Note that arguing correctness is not always required when making and justifying a safety claim. However, for the Swift UAS, the correctness of the avionics software is itself safety related, i.e., incorrect behavior is unsafe behavior. Because the claims given here are informal, and although we refer to correctness, this should be understood to be informal. We make this context explicit in the argument fragment, in part, through the definition of correctness of the software components (C6).

In the particular case of the autopilot, which is part of the avionics software, and also forms a part of the failsafe system for contingency management, its correct behavior (in reference to its informal specification given in Fig. 7) is required to assure safety. We show this requirement as the away goal AG2 in Fig. 13, referencing the module manual–sc–autopilot (corresponding to the module M16, in Fig. 11) that contains the autopilot software argument, which we describe next.

3) Module M16. Autopilot Software Argument: The main claim in the argument for the autopilot software is that its behavior during descent is correct (G14, in Fig. 14). We apply four strategies to develop this claim: (i) S1, validity of the specification; (ii) S2, correctness of the implementation; (iii) S3, satisfaction of higher-level requirements; and (iv) S13, validity of the mapping between state variables and aircraft data. We apply the strategies S1 and S2, over the constituent modules of the autopilot, i.e., its software hierarchy.

Fig. 14 also shows how the argument for the autopilot software explicitly captures logical dependencies between correct software behavior, and heterogeneous information. Specifically, to support the main claim in the argument, we need to show, in part, that the functionality contained in its modules is also correctly specified. Thus, for the claim that the computation of angle of attack is correctly specified (G15), we use the data from wind tunnel calibration experiments of the pitot probe (air data sensor), and reviews of the software specification conducted against the theoretical formula for angle of attack, as supporting evidence (nodes E3 and E4, respectively). Furthermore, to support the main claim (of the autopilot software argument), we also need to show, in part, that the autopilot controller (AP) implementation is correct (shown as the away goal AG1 in Fig. 14, referencing module manual – sc – ap – implementation, which corresponds to module M17 in Fig. 11).

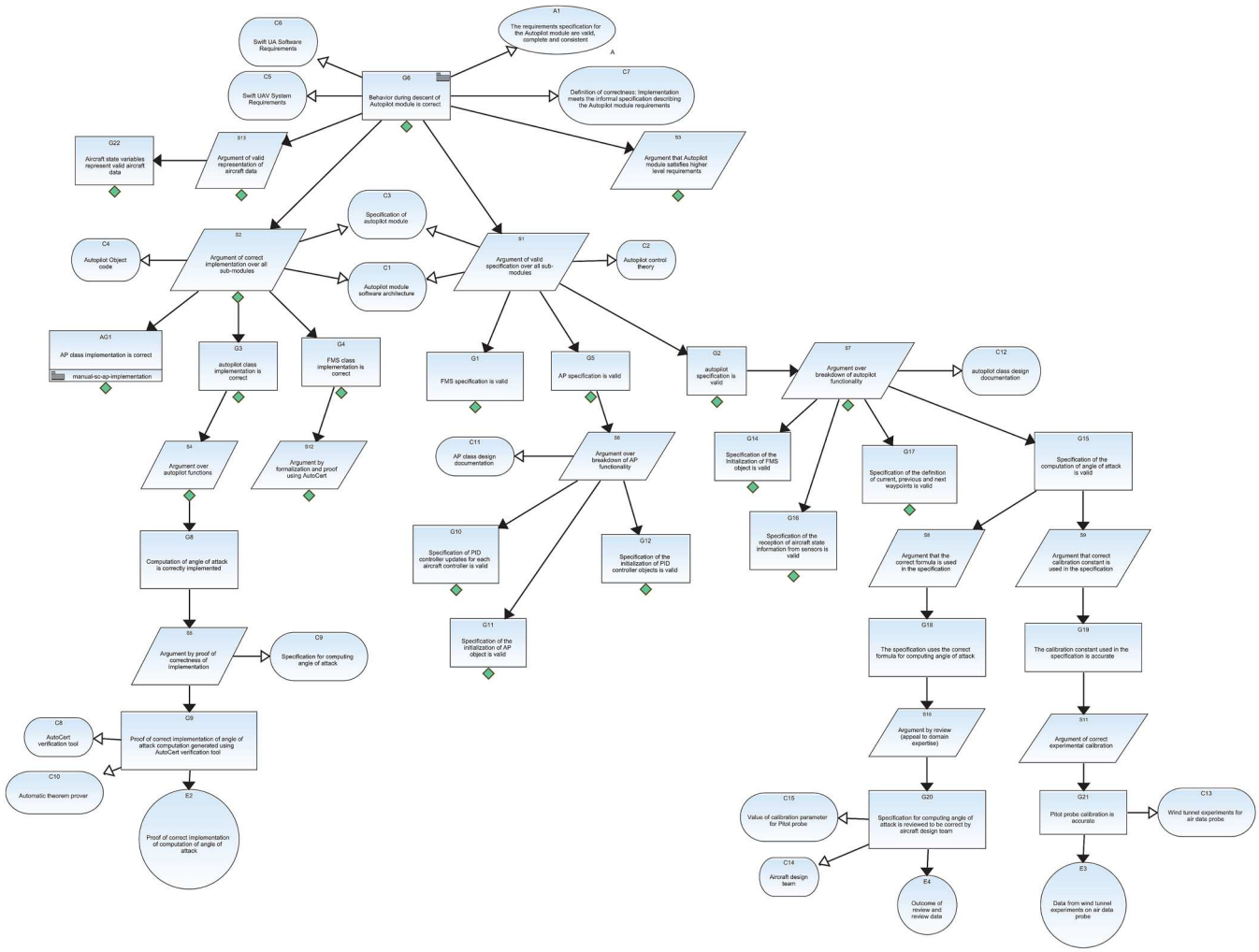

Fig. 14. Argument fragment for the Swift UAS autopilot module.

Here, it is worth noting that the argument structure mirrors, to an extent, the (informal) breakdown of autopilot functionality (Fig. 7). For example, the main requirement of autopilot correctness (Fig. 7) maps[3] to several corresponding claims in the safety case, one of which is the claim of autopilot correctness during descent (Goal G6, in Fig. 14). Subsequently, the claims about correct implementation (and valid specification) of the autopilot class (goal nodes G3, G2), the FMS class (goal nodes G4, G1), and the AP class (goal nodes AG1, G5) in Fig. 14, respectively, map to the informal requirements SR-(1), SR-(2), and SR-(3) in Fig. 7.

There are similar such claims for the correctness of the autopilot for other flight phases, but these have not been shown in the paper mainly because we focus on a slice of the overall argument, and on the descent phase. Furthermore, not all the requirements in Fig. 7 have been transferred to the argument structures because, in part, our main focus for the paper was on those requirements relevant for the slice we considered.

[3]Note that the wording of the requirements differs slightly from the wording of the claims in the safety case, e.g., in the latter we differentiate between the implementation and the specification, whereas in the former, as shown in Fig. 7, this distinction is not made.

4) Module M17. Autopilot Controller Argument: The main goal in the autopilot controller (AP) argument (Fig. 15) is that its implementation is correct. To demonstrate that this claim holds, our strategy is to show the correct implementation of its code blocks, one of which implements the PID controllers for the elevator and aileron control surfaces (shown in Fig. 15, as the away goals AG1 and AG2 respectively).

Once again, it is worth highlighting the correspondence between the claim in goal node G4, the low-level software safety requirement it represents (SAFR_AVCS_001 in Table I), and the system safety requirement (SAFR_FLTCS_001 in Table I) from which the software safety requirement was, in turn, derived. Additionally, the claim in goal node G4 in Fig. 15 corresponds to the informal requirement SR-3(d) in Fig. 7, while the sub-claims in the away goals AG1, and AG2, respectively, correspond to the sub-requirements SR-3(d)i.

Furthermore, because these requirements were formally specified (Fig. 8), we can create formalized claims for AG1 and AG2 (described next). After formalizing the claims, we auto-generate the supporting argument fragments (reflecting modules M18 and M19 in Fig. 11), and then auto-assemble them into the argument architecture. Now, we describe the auto-generated argument.

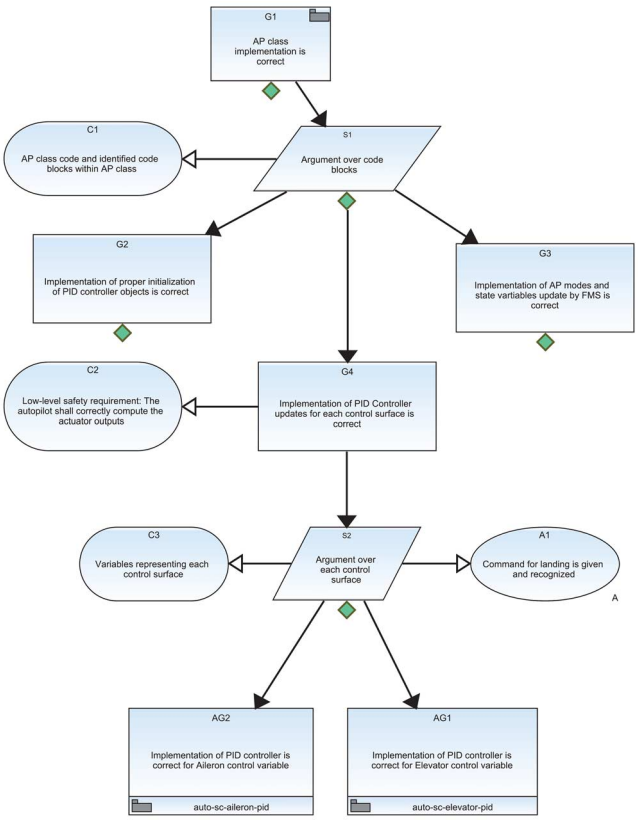

Fig. 15. Argument fragment for the autopilot controller containing claims about PID controllers for the UA control surfaces.

D. Automatically Generated Safety Argument

First, we formalize the claims in the leaf nodes of Fig. 15, i.e., that the implementations of the PID controller for the aileron and the elevator control variables are correct. The formalization step is the point where we relate variables and constants to physical artifacts, and justify this correspondence. Effectively, we relate our formal model to the world here.

Fig. 16 shows how this task has been accomplished. The main (informal) claim is that the implementation of the PID controller is correct for the aileron control variable (Goal G122 in Fig. 16, corresponding to away goal AG2 in Fig. 15). We apply the strategy of formalization (S19) in the context of the relevant domain theory, using the AUTOCERT specification language. Of the resultant sub-claims, one of them (Goal AC1) is the formalized statement of the informal claim, i.e., the formula produced by translating the informal requirement using the chosen formal language. Any formal argument relies on explicit assumptions about the artifact under verification, and in this case the assumptions relate to aircraft state variables. Assumptions on goals are typically about the system (e.g., the system operated correctly), whereas assumptions on strategies are typically about the assurance (e.g., the hazard is analysis complete). Here, the assumptions relate to code variables, and so refer to the system (e.g., x represents altitude), and are therefore attached to goals.

In principle, assumptions can be placed anywhere in scope of the goal corresponding to the formal requirement. One choice is to attach all the formal assumptions directly to the formalized claims. To do so, we could repeat the assumptions at each goal which uses them or by use cross-references for repeated nodes. Another choice is to attach assumptions to the most recent common ancestor of all formalized requirements. Our algorithm attaches them at the corresponding goal requirements, e.g., formalized assumption FA1 attached to formalized claim AC1. Although all the relevant assumptions (see Fig. 8) are to be attached, we have only shown one due to space constraints.

In addition to the reliance on explicit assumptions, a formal argument also relies on the validity of the formalization, that is, on the correspondence between the formal artifacts we reason with, and the physical artifacts they model, to engender confidence. In Fig. 16, we express this result as a supporting claim (G123) after applying the strategy of formalization (S19).

The argument structure used to justify the formalized claim shows the structure of the verification (only one step has been shown in the formal verification). To auto-generate the argument structure, we use the verification information produced by the AUTOCERT tool [22]. In particular, AUTOCERT decomposes the formal equivalent of the claim under consideration into a number of sub-claims and side conditions, i.e., VCs, via repeated property decomposition. Then, automated theorem provers can discharge these formulas. AUTOCERT assumes that low-level library functions meet their specifications, and does not verify their bodies. Therefore, evidence of this assumption has to be provided through an alternative means, e.g., by testing, or from an inspection.

The fragment outlines the sequence of intermediate computations in the code used to establish the relevant goal, which is typically a property on a variable. Some of these intermediate steps correspond to lower level goals.

As mentioned earlier, Fig. 16 only shows the initial steps of the auto-generated argument to support the (formal equivalent of the) claim on the aileron control variable, `output` → `m_aileron_m1p1`, whose property is stated in the goal AC1. This property is the condition that must be shown to hold for the argument to hold. The strategy used asserts that the correctness of the claim is shown by decomposition of the correctness property, where the notion of decomposition is that embodied within AUTOCERT. The context (AC6) clarifies that the decomposition is of the correctness property at line 542, which is in reference to the original source code.

Next, we see the VC that must be shown to achieve this main claim (AC18) is also a goal (albeit incomplete, as no proofs have been generated). The claim in AC28 represents a logically dependent variable of a schema, `m_rollError_rad`, and its property; though there may be multiple logically dependent variables in general, here it is the case that the aileron control variable, `output` → `m_aileron_m1p1`, is directly logically dependent only on the variable `m_rollError_rad`. This goal also represents the start of a recursive instance of the argument structure tree, which exhibits similar reasoning until it reaches the system assumptions or axioms.

As mentioned earlier, ATPs discharge the VCs, wherein the proof provides the evidence, and the prover provides context. Thus, the solution node AC16 indicates that a proof was successfully found (using the theorem prover *Vampire-0.6*), and the content of this node shows the path to the proof object rather than the proof itself. The argument structure also needs to rep-

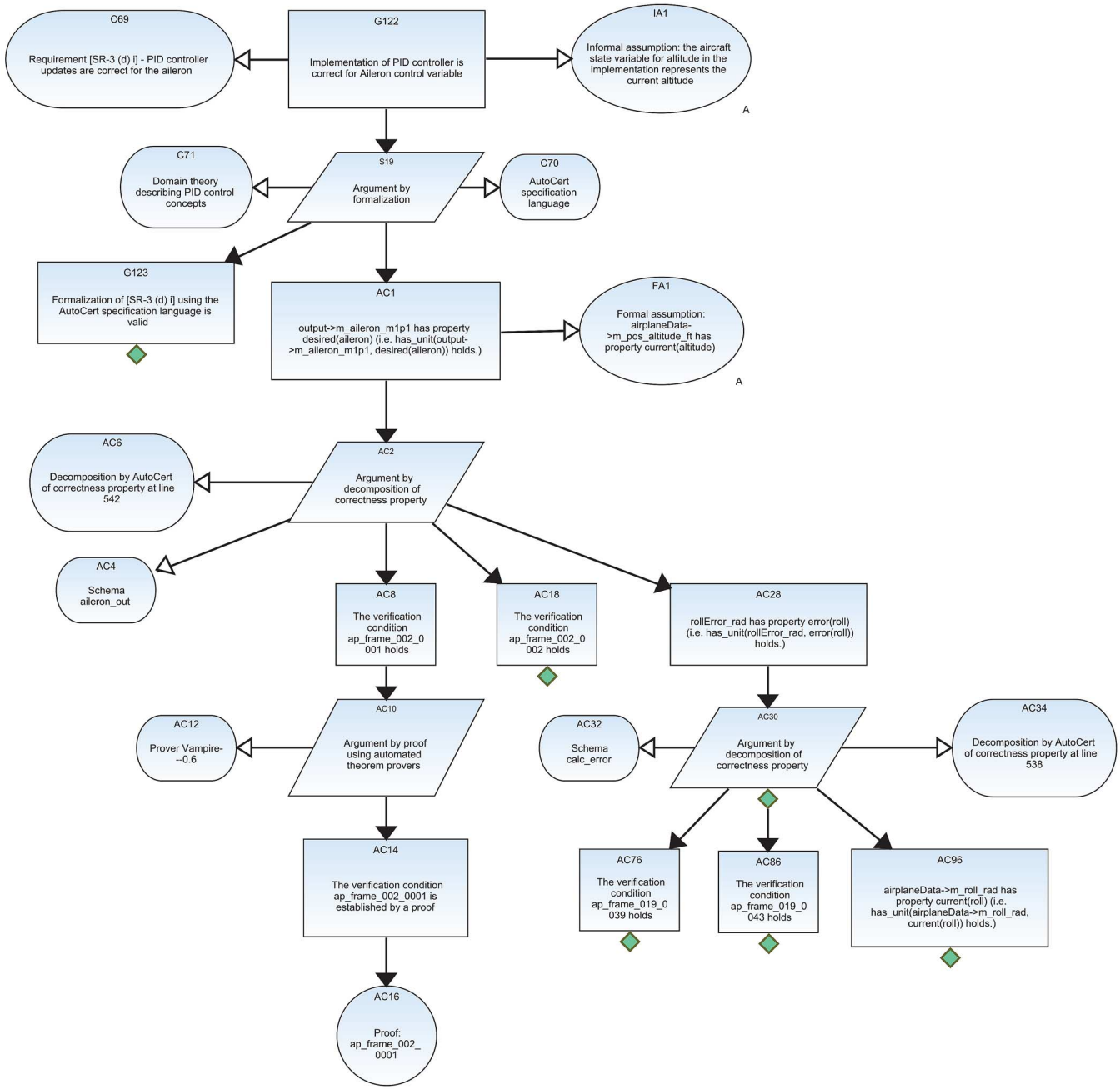

Fig. 16. A step in the auto-generated argument structure fragment for the aileron PID controller, after formalization of an informal claim.

resent any assumptions that have been made about library functions, and the methods, if any, that have been used for verification; this information is obtained from a separately specified file. Finally, the provers use various domain theories to discharge VCs; the full text of the list of domain theories includes such theories as arithmetic reasoning and transformation geometry.

Note that the properties being decomposed here can be traced to specific lines of code given in the context nodes, e.g., AC6, AC34, and so on. Additional context nodes (C71, AC4, AC32, etc.) also indicate the schemas that have been applied, i.e., the formulas that have been used. When the safety case is being evaluated, those formulas will need to be inspected by a domain expert for validity.

The argument could be restructured so that there is a single context node referencing the entire domain theory, which an expert would then need to inspect, rather than checking each individual formula at each decomposition step. However, it is important to know for traceability where the mathematical concepts, i.e., low-level requirements, have been implemented in the code. By making the decomposition structure explicit, as in Fig. 16, the automated fragment provides this traceability, although the entire fragment can be thought of (and formallyrepresented) as a single hierarchical item of evidence [25]. Although we could replace the argument underneath goal node AC1 with a logically equivalent decomposition, low-level traceability information is lost when the decomposition struc-

Goal: Implementation of PID controller is correct for aileron control variable `output->m_aileron_mlpl`. This goal requires us to formally prove that `ap` satisfies the following formal requirement:

```
has_unit(output->m_aileron_mlpl, desired(aileron))
```

The relevant external assumptions for the verification of this goal are:

1. `airplaneData->m_heading_rad` is a value representing a current heading
2. `airplaneData->m_pitch_rad` is a value representing a current pitch
3. `airplaneData->m_pos_altitude_ft` is a value representing a current altitude
4. `airplaneData->m_roll_rad` is a value representing a current roll
5. `airplaneData->m_pos_north_ft` is a value representing a position to north
6. `airplaneData->m_pos_east_ft` is a value representing a position to east
7. `airplaneData->m_airspeed_fps` is a value representing a current speed
8. `srcWpPos` is a value representing a position in the North East frame
9. `dstWpPos` is a value representing a position in the North East frame
10. `currACPos` is a value representing a position in the North East frame
11. `m_latmode` is equal to `'LATMODE_CROSSTRACK'`
12. `m_lonmode` is equal to `'LONMODE_ALTITUDECMD'`
13. `m_desiredAltitude_ft` is a value representing the desired altitude
14. `m_desiredpitch_rad` is a value representing the desired pitch
15. `m_prevWaypointEast_ft` is a value representing a position to east
16. `m_prevWaypointNorth_ft` is a value representing a position to north
17. `m_waypointEast_ft` is a value representing a position to east
18. `m_waypointNorth_ft` is a value representing a position to north
19. `lineB` is a value representing an intersection point
20. `lineM` is a value representing a slope

In order to certify this requirement, we begin with the property of the variable

- `output->m_aileron_mlpl`

that occurs in the formal requirement; the fact that this implies the requirement is shown by a single verification condition:

- `ap_frame_019_0051`

We then turn to the sequence of intermediate variables on which the safety of the initial variable depends:

- `headingError_rad`,
- `m_pidTargets->m_currentXTrackErr`
- `m_pidTargets->m_desiredheading_rad`
- `m_pidTargets->m_desiredroll_rad`
- `m_pidTargets->m_xtracksignal_deltaHeading`
- `rollError_rad`

and the following input variables from APUpdate

- `airplaneData->m_heading_rad`
- `airplaneData->m_roll_rad`
- `dstWpPos`
- `srcWpPos`
- `currACPos`

The variable `output->m_aileron_mlpl` has a single relevant occurrence in the requirement. The property is established at a single location, ap.cpp, line 542 by the aileron variable calculation. The correctness of the definition gives rise to 2 verification conditions.

- `ap_frame_002_0001` (i.e., establish the precondition at line 542 under the substitution originating from line 539)
- `ap_frame_002_0002` (i.e., establish the precondition at line 542 under the substitution originating from line 539)

It relies, in turn, on the property of the following variable:

- `rollError_rad`

Fig. 17. Fragment of a draft safety case narrative relating to the correct implementation of the aileron PID loop [14].

ture is not made explicit. The steps are similar for the elevator control variable.

Fig. 17 shows a fragment of an automatically created *safety case narrative*, corresponding to a step in the auto-generated fragment.[4] We also automatically create such an argument fragment (or its narrative) for the elevator control surface (refer to [14] for details). The narrative report is generated from the same verification information used to generate the (formally constructed part of) the argument. It can be seen as a textual rendering of the formal argument, which highlights certain key tracing information, including traces from (i) high level requirements to low level requirements, (ii) requirements to assumptions, (iii) requirements to code, (iv) requirements to concepts (mathematical concepts used to meet requirements), and (v) requirements to evidence (such as proofs). These reports are intended to serve as aids during code reviews.

E. Automatically Assembled Safety Case

Once the automatically generated safety case fragment has been created, it must be merged with the manually created fragment (described earlier in Section VII.C). To do this (as mentioned earlier in Section VI.C), we use unique identifiers, and associate the top-level goals in the auto-generated argument with the relevant incomplete goals from the manually created fragment, i.e., for each top level requirement from the AUTOCERT

[4]Creation of the narrative, in fact, is logically independent of whether the argument fragment is manually created or auto-generated; for more details, see [26], and [27]. Note also that the report lists several assumptions that were not used in the corresponding formal specifications.

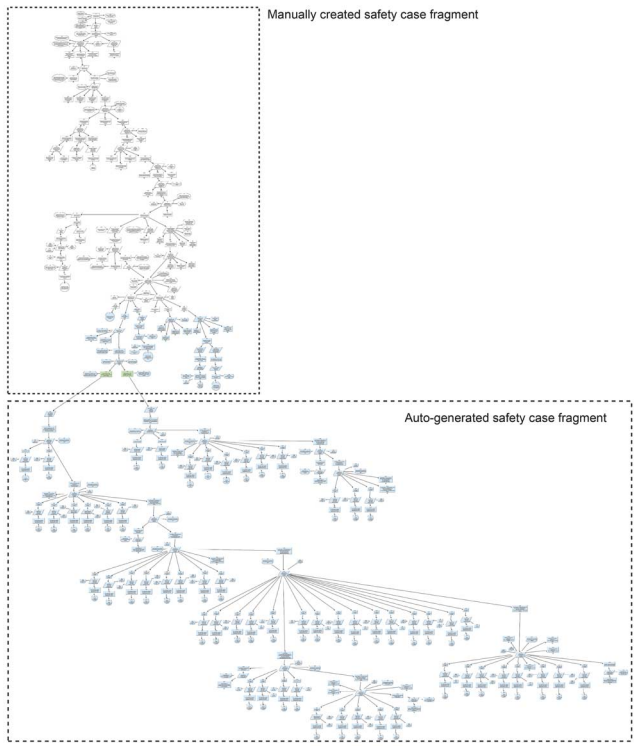

Fig. 18. Bird's eye view of an end-to-end slice of the overall system safety case for the Swift UAS.

verification, we match nodes in the auto-generated fragment with nodes in the manually created fragment via the identifier. Then, 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.

Fig. 18 shows a bird's eye view of the resulting end-to-end slice (an open box equivalent of the module diagram in Fig. 11), comprising the *manually created* fragment, and the *automatically generated* fragment, both of which were created, and then automatically assembled using our toolset AdvoCATE [27].

VIII. DISCUSSION

It is important to note that, because the Swift UAS is under development, its safety case (and the argument fragment that we have presented) is interim in the safety case lifecycle. To *fully* justify the top-level claim of system safety, we also require (repeated) evidence of safe flight and other evidence from operation (which is contingent on mission specific configurations and weather conditions). This evidence forms part of the *operational* safety case, which we have yet to create. We make the following observations about the use of our methodology and the development of safety arguments in general, following which we outline some avenues for further enhancing our work.

A. Observations

1) Heterogeneity: Safety cases are necessarily heterogeneous in nature. Even in arguments that address software concerns alone, heterogeneous evidence needs to be considered. For example, in Fig. 14, one leg of the argument fragment

assures correct computation of the aircraft angle-of-attack. The evidence includes, among other things, wind-tunnel testing of the air-data (pitot) probe, because a calibration constant relevant to the pitot probe appears as a parameter in the code. While software verification evidence for this function, as required by traditional certification processes, would mainly need to show that the code implements the specification, the assurance argument goes further: it highlights that the claim (of correct computation of angle of attack) also relies on the evidence that the sensor is appropriately calibrated.

2) Improving Comprehension: One of the primary motivations for creating a safety case is to communicate safety information to the relevant stakeholders. Hence, we believe that a safety case should not be viewed as a static, unchanging artifact. Rather, as a formal record of that which has been done (and continues to be done) to make a system safe, it is the means for continually monitoring and assuring safety throughout the lifecycle of a system. We also believe that a graphical argument structure is intuitively superior at communicating to a regulator, the claims, assumptions, justifications, and evidence that have been assimilated for assuring that a system is acceptably safe. The argument structure is an index into this compendium of information, providing both a global overview, and a way to navigate to supporting information and examine the details (with appropriate tool support). The argument structure can also be represented textually (cf. Fig. 17); however, we want a structured formalism which shows the decomposition down to low-level requirements, and eventually to evidence.

With the specific case of encoding the reasoning underlying a deductive verification as a GSN argument structure, as given in this paper, there are a couple of advantages in graphically showing the existing level of detail: (a) a unified notation, i.e., the GSN, is available to review both high-level system claims and lower-level software claims, assumptions, context and evidence; and (b) the autogenerated argument structure (e.g., Figs. 16 and 18) highlights the exact reasoning used and the traceability from requirements to code, giving a basis to gauge whether the output from the verification tool is trustworthy. Demonstrating traceability from high-level requirements to low-level requirements, and eventually to source code, is typically mandated by aviation safety assurance standards. Thus, this traceability information, which we capture in the auto-generated argument, is useful to identify those mathematical concepts that have been used, and where those concepts have been captured in the code.

Effectively, a graphical argument structure can explicitly highlight the logical and stochastic interdependence between system elements and software in the safety assurance case, so as to better explain the contribution of software to system safety concerns. Nevertheless, there is a need to hide some details and to manage the size of the argument. For instance, a regulator may not wish to see the very low-level details of some formal verification, such as the proof of a VC. In this case, there are at least two ways to modularize or abstract argument structures: (i) using GSN modules (Fig. 11), and (ii) automated hierarchical abstraction [25], which can substantially reduce the number of nodes that are shown in a graphical view. Hierarchy and modularization are orthogonal concerns to the work presented in this paper, and out of scope.

Automation can also assist in creating instance arguments from their corresponding abstract argument patterns [28]. Effectively, a safety analyst then deals with the abstract safety argument, which is smaller and more manageable. In fact, the style of decomposition used during the verification of low-level requirements (see Fig. 16) can be formally defined as the AUTOCERT *property decomposition pattern* [29]. We have also implemented an *architecture decomposition pattern*, which is to be used when the decomposition style is over sub-systems, wherein the successive steps of the decomposition emphasize *assume-guarantee contracts* between sub-systems. This decomposition could also, in principle, be replaced with a logical equivalent, but we believe the derived contracts are very useful in providing insight into the assurance of the system.

How a safety argument is initially structured is likely to play an appreciable role in its comprehensibility and complexity. For instance, in the Swift UAS safety case, one of the main strategies in the aircraft-level argument (module M2 in Fig. 11) is to argue that hazards across all operating phases have been mitigated. This argument is followed by the argument over the architectural breakdown of the airborne system (e.g., in module M5 in Fig. 11). This approach allows us to address, at the outset, those hazards whose risk is probabilistically dependent on the mission phase, e.g., failure of the nose wheel actuator may not be hazardous during the cruise phase, but it is a hazard during landing.

An alternative organization is to develop the relevant claim, first by argument over the architectural breakdown, and then over the relevant operating phases. We believe that this approach facilitates the creation of a safety argument which may be both easier to maintain and better modularized. In general, such choices are tied to argument design criteria (Section V.B), and need careful consideration early during the development of the safety argument.

3) Reducing Errors, and Improving Confidence: There are a variety of errors in safety case construction that could be eliminated by automation. For example, syntactic errors can be eliminated by simple syntax checking, and acyclicity in arguments can be checked by ensuring that there are no loops in the argument structure. The specific type of errors that are reduced here are logical fallacies in those parts of the argument that have been auto-generated from deductive verification. Logical fallacies in an argument can be avoided by verifying the argument, or by constructing it automatically from artifacts (which, of course, need their own appropriate verification). In this work, we present one approach to automatic construction of argument fragments. The approach also requires explicit specification of assumptions and formal requirements, and can reveal both missing assumptions, and unused assumptions.

There is a need for a quantitative assessment framework [30] to support decision making and argument assessment, e.g., whether an argument is valid and covers sufficient information, whether the evidence supplied is trustworthy, and the amount of confidence that can be justifiably placed in the claims made along with the supporting arguments. This work corresponds to the activity of uncertainty assessment, in our methodology.

There are a variety of ways to deal with sources of doubt in a safety case. Our previous work [19] identifies one way for-

ward using Bayesian probabilistic reasoning, for a quantitative consideration of uncertainty in argument structures. Confidence arguments provide another way to deal with some sources of doubt within the safety case, by creating an argument structure justifying the reduction of assurance deficits. Other ways to improve confidence in an assurance case also exist, such as eliminative induction, and the use of Dempster-Shafer theory.

Additionally, we have identified an initial set of metrics that we believe will support this need [14]. In brief, these include coverage measures for hazards, higher-level and lower-level safety requirements, measures of (internal) completeness, along with simple measures of argument structure size. In large safety cases, such metrics can conveniently summarize the state of the safety argument during system evolution. An implicit assumption underlying the coverage measures that we have defined is that the argument chains in the safety case are themselves valid. Manually created, inductive arguments can be systematically reviewed for identifying argument fallacies, as in [31], which is part of our ongoing work to improve confidence in the Swift UAS safety case. Assessing argument validity, and including this notion into the coverage measures, appears to be a promising mechanism also to support decision making.

B. Enhancements

Although we have only verified one small part of the system, we can potentially do much more. Although the AUTOCERT tool is aimed at one specific kind of analysis, we intend to combine the results of other kinds of formal verification. AUTOCERT provides a proof that source code complies with a mathematical specification. As part of its analysis, AUTOCERT reverse engineers the code, sifting through potentially overlapping fragments to create links from the code to high-level functional descriptions of concepts used in requirements. The functional descriptions are specified by *annotation schemas*, and AUTOCERT works by inferring annotations at instances of these patterns. It then generates the chain of reasoning which allows the requirements to be concluded from the assumptions, where each link in that chain corresponds to a particular implementation pattern. It thus provides a decomposition of the argument which lends itself naturally to inclusion in a safety case. However, not all mathematical properties are best specified in such a compositional dataflow style, and we plan to actively investigate integrating results from other tools.

As our work progresses and matures, we anticipate developing recommendations for a safety case methodology that is aligned with NASA standards and procedures. Now, we list some specific lines of work that we believe are worth pursuing.

1) Additional Formal Verification: The low-level claims that we verified were based on functional and physical unit correctness. There are other properties that should be checked in the avionics software, such as runtime safety and physics-based bounds. One possible way to do this is to combine results and reasoning from different (verification) tools, as has been illustrated for the AUTOCERT tool in this work.

2) Additional Automation and Assembly: Our hazard analysis has been conducted manually. However, there is potential for automation, especially when considering hazard identification guided from definitions of the system boundaries, as well as from its functional and physical decompositions, i.e., we hypothesize that it is possible to automate some part of the hazard analysis, in particular via the systematic enumeration of the combinations of system components or their interactions at the defined system boundaries. We believe that this will ameliorate the integration of safety cases into existing processes.

3) Inclusion of Formal Verification Knowledge: There are many different ways of converting formal verification knowledge into safety case fragments; the structure of the safety case is driven by both the safety methodology, and the verification methodology.

Examples of high-level choices include whether to decompose over all requirements, versus all scenarios, or all code branches. Rather than hard-coding these design choices in the transformation, they could be represented declaratively using templates. This approach would give us control over the structure of the generated safety case, and let us more easily investigate and compare different structures.

There are also choices in how to layout the generated safety case. We have chosen to duplicate shared VCs, but they could be shared as a directed acyclic graph.

4) Alignment With NASA Standards, Requirements, and Guidelines: NASA has numerous safety-relevant procedural requirements, standards, and guidelines at both the software and system levels. It will be important to develop a safety case methodology which is aligned with these. We also plan to continue our work on uncertainty assessment in argument structures [19], both with respect to its theoretical basis and its application to our target system, so as to be compatible with NASA's efforts in Probabilistic Risk Assessment (PRA) [32].

5) Safety Case Manipulation: The inclusion of automatically generated fragments, and all relevant sources of information, will lead to increasingly large safety cases. Existing mechanisms for managing large argument structures are limited, and we believe they should be amenable to manipulation in various automated ways. The Query-View-Transformation (QVT) standard for model manipulation could be applied here.

We also believe that introducing hierarchy into safety cases, i.e., *hicases* [25], is a promising approach that could reduce the burden of review. There is also a natural fit with automatically generated cases, as hierarchical structuring can also be generated automatically. Indeed, there are additional forms of meta-information that can also be generated to aid review.

IX. CONCLUSION

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 was to show how heterogeneous safety aspects, especially those arising from formal, and non-formal reasoning, can be communicated in a unified way to support certification activities, rather than to claim safety improvement.

Several aspects of our work distinguish it from the existing literature and our previous work.

1. The safety argument that we have created (Fig. 18) provides a level of detail that goes significantly beyond the state of the practice. Typically, argument structures leave several details implicit or informal, rarely going down to the level of software implementations. Making safety-relevant data and its connections to requirements explicit is highly worthwhile because a safety case serves primarily as a form of communication. We believe This explicit connection is also useful, as a form of book-keeping used to track and trace the data relevant for system safety assurance. Indeed, feedback from the Swift UAS engineers indicated that they viewed the assurance argument as an information log providing a transparent record that safety concerns have received sufficient consideration [42].
2. We have demonstrated the feasibility of automated assembly of manually developed safety case fragments with those generated automatically. However, we believe much more can be done to increase the degree of rigor and formality. Thus far, safety cases have not generally combined manually developed and auto-generated fragments. When automation is applied, it tends to be used to provide evidence as a monolithic black box, rather than as a full-fledged argument fragment that can be separately and rigorously examined.

   Although there exists skepticism about the value of creating such detailed arguments (particularly the criticisms that a proof need not be further examined in the form of an argument structure, and that such structures are excessively detailed to be amenable to review), we believe that it can be addressed in a straightforward manner with the appropriate abstractions. From the feedback to this work given by the Swift UAS engineers, we have already identified the need for abstraction mechanisms to manage the complexity of argument structures [42]. More recently, we have defined hierarchical structures, *hicases*, which can be applied to abstract argument fragments precisely to address the concern of hiding details. Hicases can be applied orthogonally to the modular abstraction shown in Fig. 11. See [25] for more details.
3. We have combined traditional safety analysis techniques with formal methods; 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, and shown how it can be explicitly reflected in the created argument fragments. Furthermore, we view formal and non-formal sources not as opposites but as complementary, and equally relevant.
5. We have highlighted how software is considered in the system context with explicit justification for the constituent parameters and specifications, when viewed from a safety perspective.

We have implemented our approach for argument development in AdvoCATE, our toolset for assurance case automation [27]. The core of the system is a graphical safety case editor, integrated with a set of model-based transformations that provide functionality for manually creating argument structures, translating and merging pre-existing structures from other formats, and for incorporating automatically generated content from external formal verification tools, as described in this paper. The tool also provides additional automation capabilities, such as auto-generation of textual narratives and tabular representations, computing metrics, and the creation of to-do lists.

We believe that the work presented here is a promising step towards increased safety assurance, particularly in UAS. The need to manage and reconcile diverse information in both the system and software safety cases becomes apparent from the perspective of not only safety, but also compliance to airworthiness requirements for operating a UAS [43], where the overarching goal is to show that a level of safety equivalent to that of manned operations exists. Safety analysis is imperative to determine the required regulations, and whether existing regulations are sufficient or how they ought to be augmented. 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.

ACKNOWLEDGMENT

We thank Corey Ippolito and Mark Sumich for providing the necessary domain information. We also thank the anonymous reviewers for their insights, which helped to improve this paper.

REFERENCES

[1] S-18, Aircraft And System Development And Safety Assessment Committee, ARP 4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, Dec. 1996, Society of Automotive Engineers (SAE).

[2] Software Considerations in Airborne Systems and Equipment Certification, RTCA SC-205 and EUROCAE WG-71, DO-178C/ED-12C, Dec. 2011.

[3] F. Redmill, "Safety integrity levels—Theory and problems, lessons in system safety," in *Proc. 18th Safety-Critical Systems Symp.*, 2000, Springer-Verlag.

[4] I. Dodd and I. Habli, "Safety certification of airborne software: An empirical study," *Rel. Eng. Syst. Safety*, vol. 98, no. 1, pp. 7–23, 2012.

[5] *Road Vehicles-Functional Safety*, ISO 26262, International Organization for Standardization (ISO), Nov. 2011.

[6] Draft Guidance, United States Food and Drug Administration, FDA, Guidance for Industry and FDA Staff—Total Product Life Cycle: Infusion Pump—Premarket Notification, Apr. 2010.

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

[8] R. Bloomfield and P. Bishop, "Safety and assurance cases: Past, present and possible future—An Adelard perspective," in *Proc. 18th Safety-Critical Systems Symp.*, Feb. 2010.

[9] EUROCONTROL—European Organisation for the Safety of Air Navigation, Preliminary Safety Case for Enhanced Traffic Situational Awareness During Flight Operations, PSC ATSA-AIRB, Dec. 2012 [Online]. Available: http://www.eurocontrol.int/articles/cascade-documents

[10] P. Bishop and R. Bloomfield, "A methodology for safety case development," in *Industrial Perspectives of Safety-Critical Systems: Proc. 6th Safety-critical Systems Symp.*, F. Redmill and T. Anderson, Eds., Feb. 1998, Springer.

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

[12] EUROCONTROL—European Organisation for the Safety of Air Navigation, Preliminary Safety Case for ADS-B Airport Surface Surveillance Application, PSC ADS-B-APT, Nov. 2011 [Online]. Available: http://www.eurocontrol.int/articles/cascade-documents

[13] E. Denney, G. Pai, and J. Pohl, "Heterogeneous aviation safety cases: Integrating the formal and the non-formal," in *Proc. 17th IEEE Int. Conf. Engineering of Complex Computer Systems (ICECCS)*, Paris, France, Jul. 2012, pp. 199–208.

[14] E. Denney, G. Pai, and J. Pohl, Automating the Generation of Heterogeneous Aviation Safety Cases, NASA Ames Research Center, Tech. Rep. NASA/CR-2011-215983, Aug. 2011.

[15] E. Denney and B. Fischer, "Generating customized verifiers for automatically generated code," in *Proc. Conf. Generative Programming and Component Engineering (GPCE'08)*, Nashville, TN, USA, Oct. 2008, pp. 77–87, ACM Press.

[16] N. Basir, E. Denney, and B. Fischer, "Building heterogeneous safety cases for automatically generated code," in *Proc. Infotech@Aerospace*, St. Louis, MO, USA, 2011.

[17] Office of Safety and Mission Assurance, NASA General Safety Program Requirements, NPR 8715.3C, NASA, Mar. 2008.

[18] U.S. Department of Transportation, Federal Aviation Administration, *System Safety Handbook,* FAA, Dec. 2000.

[19] E. Denney, G. Pai, and I. Habli, "Towards measurement of confidence in safety cases," in *Proc. 5th Int. Symp. Empirical Software Engineering and Measurement*, Sep. 2011, pp. 380–383.

[20] T. Kelly, "Arguing safety: A systematic approach to managing safety cases," Ph.D. dissertation, Univ. York, York, U.K., 1998.

[21] T. Kelly, "A systematic approach to safety case management," in *Proc. Soc. Autom. Eng. (SAE) World Congr.*, Mar. 2004.

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

[23] K. Y. Ahn and E. Denney, "A framework for testing first-order logic axioms in program verification," *Softw. Qual. J.*, pp. 1–42, Nov. 2011.

[24] Office of Safety and Mission Assurance, NPR 8715.5A, Range Flight Safety Program, NASA, Sep. 2010.

[25] E. Denney, G. Pai, and I. Whiteside, "Hierarchical safety cases," in *Proc. 5th NASA Formal Methods Symp.*, G. Brat, N. Rungta, and A. Venet, Eds., May 2013, vol. 7871, pp. 478–483, ser. LNCS, Springer-Verlag.

[26] E. Denney and B. Fischer, "A verification-driven approach to traceability and documentation for auto-generated mathematical software," in *Proc. Automated Software Engineering (ASE'09)*, 2009.

[27] E. Denney, G. Pai, and J. Pohl, "AdvoCATE: An assurance case automation toolset," in *Proc. SAFECOMP 2012 Workshops*, F. Ortmeier and P. Daniel, Eds., Sep. 2012, vol. 7613, ser. LNCS, Springer-Verlag.

[28] E. Denney and G. Pai, "A formal basis for safety case patterns," in *Proc. Computer Safety, Reliability and Security (SAFECOMP 2013)*, F. Bitsch, J. Guiochet, and M. Kaniche, Eds., 2013, vol. 8153, pp. 21–32, ser. LNCS.

[29] E. Denney and G. Pai, "Evidence arguments for using formal methods in software certification," in *Proc. 2013 IEEE Int. Symp. Software Reliability Engineering Workshops (ISSREW)*, Nov. 2013, pp. 375–380.

[30] A. Wassyng, T. Maibaum, M. Lawford, and H. Bherer, "Software certification: Is there a case against safety cases?," in *Foundations of Computer Software, Modeling, Development and Verification of Adaptive Systems*, ser. LNCS. New York, NY, USA: Springer-Verlag, 2011, vol. 6662, pp. 206–227.

[31] W. Greenwell, J. Knight, C. M. Holloway, and J. Pease, "A taxonomy of fallacies in system safety arguments," in *Proc. Int. System Safety Conf.*, 2006.

[32] M. Stamatelatos *et al.*, "Probabilistic risk assessment," NASA Office of Safety and Mission Assurance, Procedures and Guide for NASA Managers and Practitioners 1.1, Aug. 2002.

[33] E. Denney, C. Ippolito, R. Lee, and G. Pai, "An integrated safety and systems engineering methodology for small unmanned aircraft systems," in *Proc. Infotech@Aerospace*, Garden Grove, CA, USA, 2012, no. AIAA 2012–2572.

[34] N. Basir, E. Denney, and B. Fischer, "Deriving safety cases for hierarchical structure in model-based development," in *Proc. 29th Int. Conf. Computer Safety, Reliability and Security (SafeComp'10)*, Vienna, Austria, 2010.

[35] E. Denney and G. Pai, "A lightweight methodology for safety case assembly," in *Proc. 31st Int. Conf. Computer Safety, Reliability and Security (SAFECOMP 2012)*, F. Ortmeier and P. Daniel, Eds., Sep. 2012, vol. 7612, pp. 1–12, ser. LNCS, Springer-Verlag.

[36] P. Bishop and R. Bloomfield, "A methodology for safety case development," in *Industrial Perspectives of Safety-Critical Systems: Proc. 6th Safety-critical Systems Symp.*, F. Redmill and T. Anderson, Eds., 1998, Springer.

[37] R. Weaver, "The safety of software—Constructing and assuring arguments," Ph.D. dissertation, Dept. Comput. Sci., Univ. York, York, U.K., 2003.

[38] E. Lee, I. Lee, and O. Sokolsky, "Assurance cases in model-driven development of the pacemaker software," in *Proc. 4th Int. Symp. Leveraging Application of Formal Methods, Verification and Validation (ISoLA)*, Oct. 2010, vol. 6416, pp. 343–356, ser. Lecture Notes in Computer Science (LNCS).

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

[40] 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. Softw. Eng.*, vol. 33, no. 5, pp. 347–365, May 2007.

[41] R. Bloomfield, B. Littlewood, and D. Wright, "Confidence: Its roles in dependability cases for risk assessment," in *Proc. 37th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN)*, 2007.

[42] E. Denney, I. Habli, and G. Pai, "Perspectives on software safety case development for unmanned aircraft," in *Proc. 42nd Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN 2012)*, Boston, MA, USA, Jun. 2012, pp. 1–8.

[43] Federal Aviation Administration, Unmanned Aircraft Systems (UAS) Operational Approval, Jan. 2013, National Policy N 8900.207, U.S. Department of Transportation.

Ewen Denney (M'09) earned a B.Sc. (Hons) in computing science and mathematics from the University of Glasgow (1993), an M.Sc. with Distinction in computer science from Imperial College (1994), and a Ph.D. degree in computer science from the University of Edinburgh (1999).

He is a Senior Computer Scientist at SGT Inc., NASA Ames Research Center, with the Robust Software Engineering group, in the Intelligent Systems Division (Code TI), where he has worked on automated code generation and safety certification in the aerospace domain, developing AI-based systems for the automated generation of code for scientific computation, and the certification of autocode.

Dr. Denney has served on numerous program committees and scientific advisory boards. He has chaired and co-chaired several conferences, including Software Certificate Management (2005), the inaugural NASA Formal Methods Symposium (2009), Proof Carrying Code and Software Certification (2009), Generative Programming and Component Engineering (2011), Assurance Cases for Software-intensive Systems (2013), and Automated Software Engineering (2013). He is the author of more than 60 publications on formal methods and safety assurance. He is a member of the IEEE Computer Society, the AIAA, and the ACM.

Ganesh Pai (S'97–M'07) holds a B.E. degree in electronics engineering (2000) from the University of Bombay, an M.S. degree in electrical engineering (2002) from the University of Virginia, and a Ph.D. degree in computer engineering (2007), also from the University of Virginia.

He is a Research Scientist at SGT Inc., NASA Ames Research Center, with the Robust Software Engineering group in the Intelligent Systems Division (Code TI), where he works on safety assurance of flight-critical systems and software. Prior to his current role, from May 2007 to March 2011, he was a Senior Engineer with the Fraunhofer Institute for Experimental Software Engineering (IESE), Germany. His research interests lie in the broad areas of systems and software engineering with a focus on dependability and safety.

Dr. Pai has served on the program committees of a series of workshops on Software Engineering for Embedded Systems, the NASA Formal Methods Symposium (2013), as co-chair of the workshop on Assurance Cases for Software-intensive Systems (2013), and on the organizing committee of Automated Software Engineering (2013). He is a member of the IEEE Computer Society, and the AIAA.