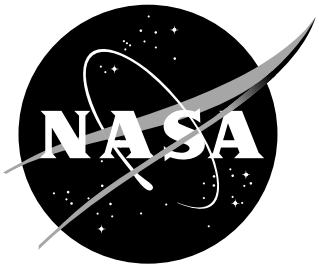
NASA/CR–2011–215983



# Automating the Generation of Heterogeneous Aviation Safety Cases

*Ewen W. Denney*
*SGT, Inc.*
*Ames Research Center, Moffett Field, California*

*Ganesh J. Pai*
*SGT, Inc.*
*Ames Research Center, Moffett Field, California*

*Josef M. Pohl*
*SGT, Inc.*
*Ames Research Center, Moffett Field, California*

August 2011

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

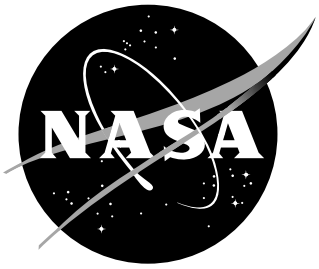
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at ***http://www.sti.nasa.gov***
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA STI Help Desk at 443-757-5803
- Phone the NASA STI Help Desk at 443-757-5802
- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076–1320

NASA/CR–2011–215983



# Automating the Generation of Heterogeneous Aviation Safety Cases

*Ewen W. Denney*
*SGT, Inc.*
*Ames Research Center, Moffett Field, California*

*Ganesh J. Pai*
*SGT, Inc.*
*Ames Research Center, Moffett Field, California*

*Josef M. Pohl*
*SGT, Inc.*
*Ames Research Center, Moffett Field, California*

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

August 2011

## Acknowledgments

This work has been funded by NASA contract NNA10DE83C. We thank Corey Ippolito and Mark Sumich for providing the necessary background information, Joe Wlad for his advice on FAA certification, Ibrahim Habli for discussions on safety cases, and Tom Pressburger for giving comments on earlier drafts. Any errors in this report are those of the authors.



Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

# Abstract

A *safety case* is a structured argument, supported by a body of evidence, which provides a convincing and valid justification that a system is acceptably safe for a given application in a given operating environment. This report describes the development of a fragment of a preliminary safety case for the Swift Unmanned Aircraft System. The construction of the safety case fragment consists of two parts: a manually constructed system-level case, and an automatically constructed lower-level case, generated from formal proof of safety-relevant correctness properties. We provide a detailed discussion of the safety considerations for the target system, emphasizing the heterogeneity of sources of safety-relevant information, and use a hazard analysis to derive safety requirements, including formal requirements. We evaluate the safety case using three classes of metrics for measuring degrees of coverage, automation, and understandability. We then present our preliminary conclusions and make suggestions for future work.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Context

The development of a safety case has been a common practice[1] for the certification of safety-critical systems in the nuclear, defense and rail domains [9]. Recently, the requirement for a safety case has been considered in emerging international standards [31], and national guidelines [22]. The move towards safety cases presents a departure from highly prescriptive safety standards in which certification, particularly of software-intensive systems, is often obtained by compliance with predefined objectives and processes. For example, a safety case is required for compliance with the new automotive functional safety standard ISO 26262 [31]. It is also a recommended practice in the US-FDA draft guidance on the production of infusion pump systems [22].

Indeed, the development and acceptance of a safety case is a key element of safety regulation in many safety-critical sectors. However, safety cases are typically constructed manually, since most tools only provide basic drawing support such as "boxes and arrows". This is time consuming and expensive, especially when we are dealing with large amounts of artifacts and iterative software development. There is a need, therefore, to integrate automated formal analysis techniques with commercial safety case tools to automatically construct safety cases.

New approaches to the development of aviation software, such as model-based development and automated code generation, offer significant potential to eliminate coding errors and reduce development costs and times. Nevertheless, despite these advances significant barriers remain to raising the level of assurance of software-intensive systems, which are particularly acute for the mathematical and safety-critical software typical in the aviation domain.

### Objectives

The principal objective of this research is to "enable improved comprehension by many participants in safety assurance" (See NRA, Amendment 7, Appendix B.6, Section 3.1.3), by "integrating formal and informal reasoning", and "ensuring that system level safety requirements allocated to software may be traced to the actual code" (Section 3.1.4.4). The research presented supports approach 3.1.4.4.

The main outcome of this research will be a safety assurance approach and tool for the automated generation of safety cases from formal verification and additional, informal, artifacts. In addition, this research supports the investigation of the types of evidence, such as domain-specific knowledge from mathematics, physics, and engineering, needed to support argument-based assurance of NextGen flight-critical systems (Approach 3.1.4.3), particularly (c), "developing argument assurance methods ... for complex software-intensive ... systems". An outcome will be an openly available framework for the specification of safety-related artifacts.

The ultimate goal of this research is a safety assurance approach and tool for the automated generation of safety cases from formal verification and additional informal artifacts. This will include an openly available framework for the specification of such safety related artifacts.

### Application of Interest

Our approach is applied to the Swift Unmanned Aircraft System (UAS).We picked the Swift UAS primarily due to the ease of accessing the system and its developers, afforded by its development locally at NASA Ames. In addition, there are practical and strategic advantages for conducting research in the construction and application of a goals-based approach for safety argumentation to a UAS:

(a) There is significant interest within the FAA on how operational approval may be provided to UAS operating in the national airspace; in the identified interim guidance [13], safety cases have been considered as an "an alternate method of compliance" to the requirements for operational approval. Consequently, research supporting the creation of safety cases for UAS is a step towards potentially addressing the FAA requirements for operating UAS.

(b) To determine the regulations needed for the safe operation of UAS (or whether existing regulations are sufficient and/or how they ought to be augmented), safety analysis is imperative. For instance [29] identifies

[1] Particularly in the UK.

several hazards in UAS and their implications for regulation, while [11] and [12] identify the properties for an effective framework for airworthiness certification for UAS as well as the challenges therein. Research on the application of the state of the art in safety assurance, i.e., safety cases, thus affords the development of a framework for assuring safety in tandem with the identification of UAS-relevant hazards.

## 1.2 Specific Research Problem

In order to address the challenges of complexity and diversity, formal approaches that encompass not just software, but also domain-specific knowledge from mathematics, physics, and engineering are required. Moreover, since it is humans who design, build, verify, and ultimately sign off on software, analysis tools need to support their activities by explaining their reasoning in a comprehensible and usable way, and not acting simply as omniscient "black-boxes".

Specifically, approaches to product-oriented certification should be developed so that software can be certified by virtue of explicit evidence, rather than through following a standard development process; safety cases are a good example of this. The key challenge therefore is to develop tools and techniques which support the automated creation of evidence-based arguments in complex domains.

In general, however, constructing safety cases is difficult and the research proposed here needs to address a variety of open problems:

**Size** Safety cases reasoning about software details will quickly grow very large; in particular, they may grow super-linearly in the size of the underlying software due to the increased number of requirements the software needs to satisfy. We need to develop querying and abstraction mechanisms such as views to keep the cognitive load on the users tractable.

**Heterogeneity of tools and artifacts** The safety cases need to reconcile different artifacts such as unit tests, simulation runs, or formal proofs from different classes of tools which provide different levels of assurance, in order to allow a comprehensive safety assessment.

**Informal and domain-specific artifacts** The heterogeneity problem is compounded by the prevalence and importance of informal (i.e., non-software) and domain-specific artifacts such as standards, engineering tables, or textbooks.

**Tracing** The safety cases need to trace all these artifacts to the code. We need to develop trace abstraction mechanisms to keep the cognitive load on the users tractable.

**Integration of domain-specific and project-specific information** The core of the safety cases can be constructed automatically, but we need to develop methods to integrate domain-specific and project-specific information into this core; in particular, we need to provide user-accessible "hooks".

**Implicit assumptions** Safety cases are designed to make implicit assumptions explicit; however, normally this is a manual process that underpins the construction of a safety case. We need to develop methods to make explicit the assumptions behind formal artifacts such as safety proofs.

**Iterative development** Traditionally, safety cases are static documents that are developed after software development has finished. This reduces the benefits of their application as it fails to uncover potential risks already during software development, and becomes entirely inadequate for iterative development styles. While we can easily re-generate the safety cases, we need to develop methods to identify and highlight changes in the different versions, and to maintain their consistency with any manually specified domain-specific and project-specific information, resp. to identify and highlight inconsistencies.

**Confidence** Serious concerns exist about current safety case practices [27], highlighting the need for methods to assess that sufficient confidence can be placed in safety cases. [28] proposes an assurance approach in which a safety case comprises two complementary arguments: the safety argument documents the reasoning supporting the claims concerning the safety of the system, while an interlinked, *qualitative* confidence argument documents the reasoning as to why the confidence in that safety argument is sufficient.

Others have also recognized the need to consider uncertainties in the safety argument, albeit from the perspective of quantification, e.g., in quantifying the epistemic uncertainty in dependability arguments when assessing the confidence in claims about the probability of failure [8]; in evaluating the confidence placed in safety arguments where claims address the achievement of a desired safety integrity level [10], and the quantification of confidence in diverse argument legs to examine whether diversity in arguments improves overall confidence in a safety claim [44].

## 1.3 Relevant Terminology

The terminology relevant to the discussion in this report is first given. Except where otherwise stated, the terminology here is taken from [57].

1. *Hazard*: In practice, several notions exist for the concept of a *hazard*. For example, as per MIL-STD-882D [57], a hazard is any real or potential condition that can cause a mishap, i.e., injury, illness, or death to personnel; damage to or loss of a system, equipment or property; or damage to the environment. The FAA defines a hazard as a "condition, event, or circumstance that could lead to or contribute to an unplanned or undesired event" [20].

   In the literature [43], a hazard is defined as "the state or a set of conditions of a system that, together with other conditions in the environment of the system, will lead inevitably to an accident (loss event/ mishap)". For our purposes, both definitions are relevant, i.e., it is important to consider not only conditions in the target system (Section 2) but also situations in the environment in which it operates, during hazard identification. Our rationale is based on Reason's model of accident causation [51], i.e., that mishaps occur largely as a consequence of the interaction between one or more unmitigated hazards.

2. *Risk*: Risk (of a mishap) is expressed in terms of potential mishap severity and the probability of its occurrence. This also corresponds to the notion of risk given in [53].

3. *Safety*: Safety is given as "freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment".

4. *System Safety*: System safety is given as "the application of engineering and management principles, criteria, and techniques to achieve acceptable mishap risk, within the constraints of operational effectiveness and suitability, time, and cost, throughout all phases of the system life cycle."

The distinction between safety and system safety is noteworthy; the latter is a process and decision tool, such that the former can be achieved within the context of an agreed upon definition of acceptable risk. Note also that *System Safety* is not the same as "safety of the system", where the latter refers to the notion that the system[2] is safe.

## 1.4 Safety Assurance Methodology

The main philosophy of our safety assurance approach[3] is that the system safety process drives the safety argumentation process, as shown in Figure 1.

The system safety process as shown in the figure is derived from the framework of a safety risk management plan such as the one recommended in the Department of Defense (DoD) standard practice for system safety (MIL-STD-882) [57], or the Federal Aviation Administration (FAA) system safety handbook[4] [58].

This plan includes safety considerations into system design at an early stage, through the identification of hazards, risk analysis and risk management. In brief, hazard identification and risk analysis involves respectively

[2]Here, the proverbial *system* refers not just to the Swift UAS, but the Swift UAS *and* its operating environment.

[3]Note that our approach, as shown in Figure 1, only shows an initial conceptualization of our safety assurance methodology. We intend to refine this methodology, eventually, to show concrete links to the system development process. Both the system safety process and the system development process influence the safety argumentation process, and vice-versa.

[4]The FAA System Safety Handbook builds on the recommendations from MIL-STD-882D; consequently there are several points of overlap.

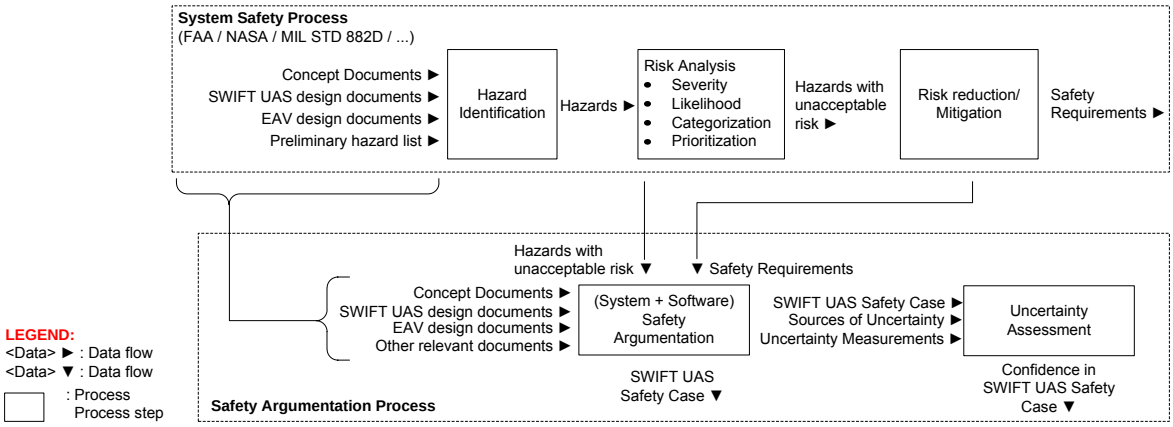



Figure 1: Our safety assurance methodology where the system safety process drives the safety argumentation process.

(i) determining those situations or conditions relevant to the system which, if left uncontrolled, have the potential to cause an undesirable event, and (ii) characterizing the consequences, severity and likelihood of such situations/ conditions. These principal steps are similar to the recommendations for probabilistic risk assessment (PRA), as given within the PRA procedures guide for NASA managers and practitioners [53]. PRA requires explicit quantification of likelihoods and consequences and can coexist with qualitative methods for risk analysis. Risk management broadly uses the outcomes of risk analysis to prioritize and mitigate risks.

In Figure 1, these are exactly the process steps Hazard Identification, and Risk Analysis. As input to these processes, we use the concept documentation for the target system - the Swift UAS, available design documentation[5], and a preliminary list of identified hazards. The outcome of hazard identification and risk analysis is used in defining mitigation measures for those hazards considered as having unacceptable risk. The outcome of this step (Risk reduction/mitigation) are, in part, requirements on system safety.[6]

The outcome of the system safety process, in effect, triggers the safety argumentation process, as shown in Figure 1. In general, the idea is to use a structured safety argument, i.e., a safety case, to systematically build a case that all identified hazards have been eliminated, or mitigated, such that mishap risks have been reduced to acceptable levels, i.e., the system is safe. The safety argumentation process is applied starting at the level of the system in the same way as the system safety process, and it is repeated at the software level. The main steps in creating a software (and system) safety case is to (a) create safety claims, e.g., indicating the mitigation of relevant hazards (b) link the evidence that supports the claims via a structured argument.

Our software verification methodology is used to create part of the software safety argument, in particular, the lower levels, and it indicates connections to the wider system safety process (Figure 2). The methodology we follow for verifying the flight software is to formally verify the implementation against a mathematical specification and to test low-level library functions against their specifications. In this report we will concentrate on formal verification using AUTOCERT, and defer testing, and verification using other tools to future work. The mathematical specification is given in a program logic, and corresponds to software requirements which, in turn, are derived from system requirements during the safety analysis. The formal verification takes place in the context of a logical domain theory (i.e., a set of axioms). Axioms can either be assumed to be correct, or they can be inspected, or they can be tested against a computational model which, itself, is inspected.

Once a safety argument has been created, we assess the trustworthiness of this argument by characterizing the

[5]Typically, when a new system is being developed, design documentation is not available at the early stages and mainly concepts are to be evaluated. In our case, we are applying our methodology to a system, the Swift UAS, which had already commenced development and reuses design information from its predecessor system, the Exploration Aerial Vehicle (EAV). Therefore available design documentation is also included in our hazard analysis.

[6]These requirements take several forms, including constraints on the design, guidelines and/or procedures for maintenance, operation, etc.

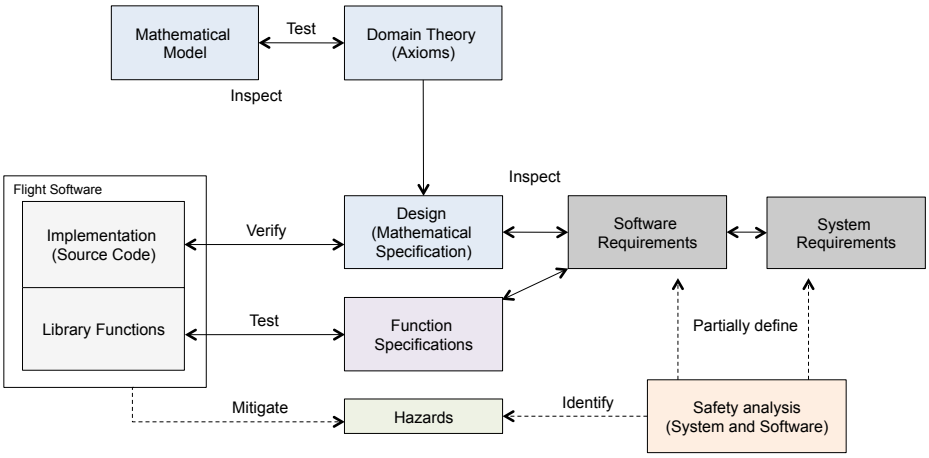



Figure 2: Software verification methodology

sources of uncertainty in the argument, and attempting to quantify the confidence (uncertainty) in the argument, via uncertainty assessment. The goal, here, is to provide as objective a basis as reasonably possible, for decision making, i.e., whether to accept or reject a safety argument.

## 1.5 Safety Cases and the Goal Structuring Notation

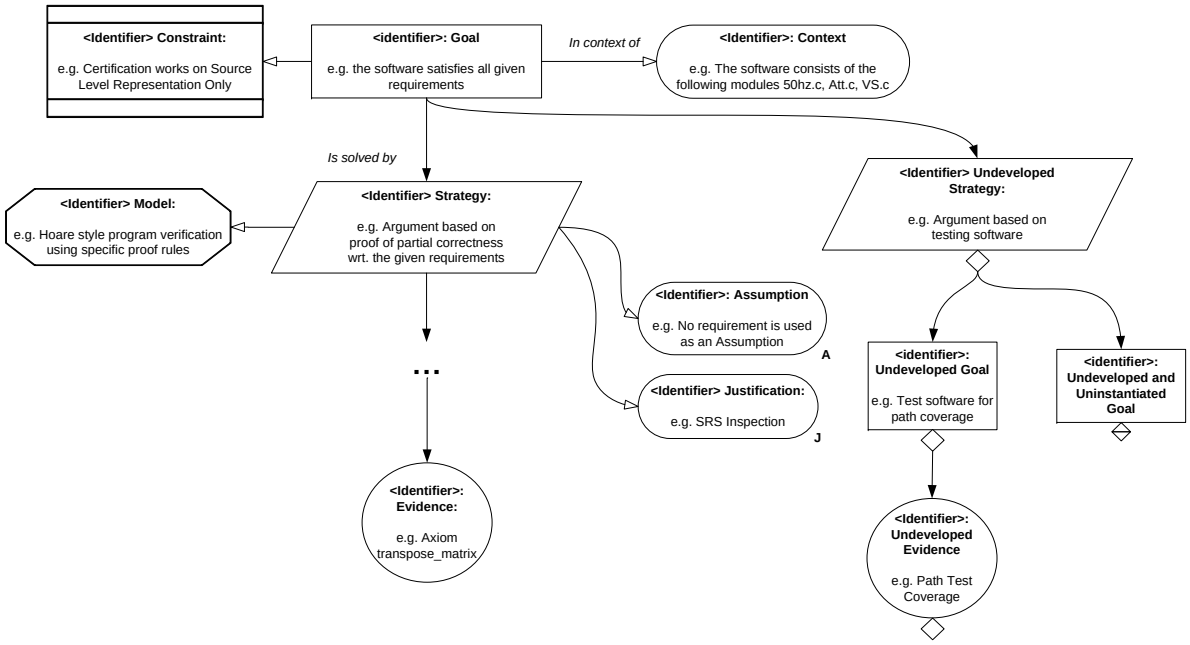



Figure 3: Syntax of the GSN notation

A safety case [55] 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." Thus, it provides an explicit means for justifying the safety of a system through a reasoned argument and supporting evidence. A safety case, thus, comprises three main elements (i) a set of claims (ii) evidence supporting the

claims, and (iii) an comprehensive, comprehensible, and structured argument linking the evidence to the claims.

Safety cases are documented in a variety of ways, including text and graphical notations. For the safety case fragments given in this report, we use the Goal Structuring Notation (GSN) [25] for documentation. The elements of the GSN are shown in Figure 3. Each element represents a specific type of information that is contained in the safety case. For example, a safety claim, i.e., a goal, is shown using a rectangle. The strategy used to decompose this claim into sub-claims is represented using a parallelogram, while sub-claims are again represented using rectangles. Assumptions, justifications and context information are documented in the GSN using rounded rectangles and, respectively, they convey the assumptions made, e.g., in stating a claim or using a strategy, the justifications, e.g., for using a particular strategy, and the context of relevance, e.g., when making a claim. Evidence is represented using a circle.

Whenever these basic elements[7] are either *undeveloped*, *uninstantiated*, or both, a diamond shape, a triangle shape or a diamond shape with a horizontal line, are respectively appended to the relevant element shape as shown in Figure 3. Undeveloped elements refer to elements which have been identified but not completely developed, i.e., it is known to be incomplete. Uninstantiated elements refer to those elements which have not yet been identified but are known or hypothesized to exist. Elements which are both undeveloped and uninstantiated serve as placeholders for possible elements which can be added into the safety case.

## 1.6 Organization

Thus far, we have described the research context (Section 1.1), and the problems which motivate the work presented in this report (Section 1.2). We have presented a high-level overview of our approach to address these research problems (Section 1.4), whilst providing the terminological (Section 1.3) and notational (Section 1.5) bases for documenting the outcomes of our work. In the subsequent sections of this report, we will describe our approach in greater detail and illustrate it by application to a target system. Specifically, the rest of this report is organized as follows:

- In Section 2, we present the target system, the Swift UAS, to which our methodology is applied. In this section, we also describe the specifics of the flight software architecture (Section 2.2), which contains the flight management system (FMS) and a controller for a specific flight control surface (Section 2.3), to which we apply our safety assurance methodology.
- The safety considerations for the target system is presented in brief in Section 3, where we first discuss the regulatory framework (Section 3.2) that constrains the development of the Swift UAS. Then, we present the contingency management measures which have been considered by the team developing the Swift UAS for ensuring the safe operation of the system (Section 3.3), and finally the system requirements relevant for safety (Section 3.4).
- Subsequently, Section 4 discusses the heterogenous information that is to be considered when analyzing the safety of the Swift UAS, and in creating a safety argument for the same.
- Thereafter, we present our safety analysis of the target system in greater detail (Section 5). In particular, Section 5.1 documents the preliminary hazard analysis while Section 5.2, presents a small sample of the resulting safety requirements. Of the requirements identified in this section, we mainly consider one high-level requirement as the base-year target safety requirement and two additional low-level requirements which are derived from the former.
- Section 6 documents an end-to-end slice of the Swift UAS safety case which we have developed after the application of our methodology. In particular, a manually-created fragment describing the safety argument starting from the system level and including the Swift UAS software is given in Section 6.1. Then we document the semi-automatically created safety argument fragment for the aileron control of the Swift UAS in Section 6.2.
- Then, in Section 7, we describe the approach we employ to enable the (semi-)automatic generation of the safety argument using our software verification methodology (Figure 2). In particular, the domain theory required for the mathematical specification of the software requirements is given in Section 7.1, followed by the specifics of the generation and transformation steps (Section 7.3). Section 7.4 discusses how we

[7] Note that some of the syntactical elements have been recently updated in the GSN, e.g., the notation for the "Model" has been eliminated.

create the safety case arguments from the formal specifications created in the application of our software verification methodology.
- We evaluate our overall work using four objective evaluation measures (Section 8). In particular, we define and apply measures for coverage (Section 8.1), degree of automation (Section 8.2), the understandability of (Section 8.3), and the uncertainty (confidence) in (Section 8.4) the safety argument respectively.
- We conclude the report with a subjective assessment and a discussion of the limitations of our work (Section 9), and with avenues for further research (Section 10).
- Appendices A, B, C, D, and E, present respectively (a) traceability from the identified safety requirements to the safety analysis and argument elements (b) parameters and variables in the Swift UAS considered in our safety assurance methodology (c) an aircraft design flow, as we have surmised from discussion with the Swift UAS developers (d) the sources of regulatory requirements which set the context for the application and adoption of this work in a wider and practical context, and (e) an overview of the software certification concerns as per the state of the practice for safety considerations in aviation software.

**Traceability to Contractual Requirements**

Deliverable 8, *Final Base Year Report*, states that we should *Evaluate the research conducted during the Base Year* and present:

- An evaluation of the limitations of the Base Year safety case generation method (Section 9).
- Measures of
  - the coverage of the generated safety case of the base-year target safety requirements (Section 8.1).
  - the degree of automation achieved in generating the safety case (Section 8.2).
  - the understandability of the generated safety case (Section 8.3).
- A discussion of the relative importance of formal and informal evidence within the generated safety case (Section 9).
- Recommendations for future work, and an analysis of how well the work proposed for the option years satisfies those recommendations (Section 10).

# 2 Target System: The Swift UAS

## 2.1 Description

Our target system is the Swift *Unmanned Aircraft System* from NASA Ames. In general, an Unmanned Aircraft System (UAS)[8] consists of some Ground Station Controllers (GSC), one or more unmanned aircraft, a control link for communications, and possibly other support equipment. In our case, the UAS comprises a single UAV (the Electric Swift UAS) plus two ground stations (a primary and a secondary) and control links (2.4 GHz for direct commands from the pilot[9] and 900 MHz for telemetry and commands from the GSC[10]). We will follow convention and generally refer to the aircraft itself as an Unmanned Aerial Vehicle (UAV).

### 2.1.1 Operation of the Airborne System

The UAV can be controlled on the ground by a pilot, or fly autonomously by following a pre-programmed or uploaded nominal flight plan (i.e., a mission). This consists of a sequence of commands (which will determine a set of waypoints), from takeoff to landing, although a pilot might takeoff, land, or intercept at any time. The off-nominal plan[11] describes the actions of the Contingency Management System (CMS), the failsafe trajectory, the procedures to be followed on the ground, and so on.

[8] UAS is the official FAA term, i.e., not UAV.

[9] They previously used 72 MHz for pilot control but this was more prone to interference.

[10] If the avionics is installed: there are two configurations — with and without the autopilot; for the initial pilot shakedown tests, the UAV must be flown without the autopilot.

[11] Unlike the nominal plan, this is not defined as a single entity.

Typically, there is a team of several individuals who operate the UAS in a semi-autonomous manner, that is, in both *pilot in control* (PIC) and *computer in control* (CIC) modes. The GSC is operated by a Ground Station Operator (GSO) who calls out important state information (e.g., the true airspeed) to the pilot. There may also be a secondary (or research) pilot, who controls the aircraft in the *secondary pilot in control* (SIC) mode. Commands can be uploaded by the GSO from the GSC to the UAV. The pilot can control the UAV via a transmitter with joystick and trim tab. A change of control is always instigated by the primary pilot via the safety switch. Figures 4 and 5 show the communication between the ground station pilots and the UAV in configurations with and without avionics, respectively. Note that the control bit feedback loop is implemented in software and informs the autopilot to disengage.

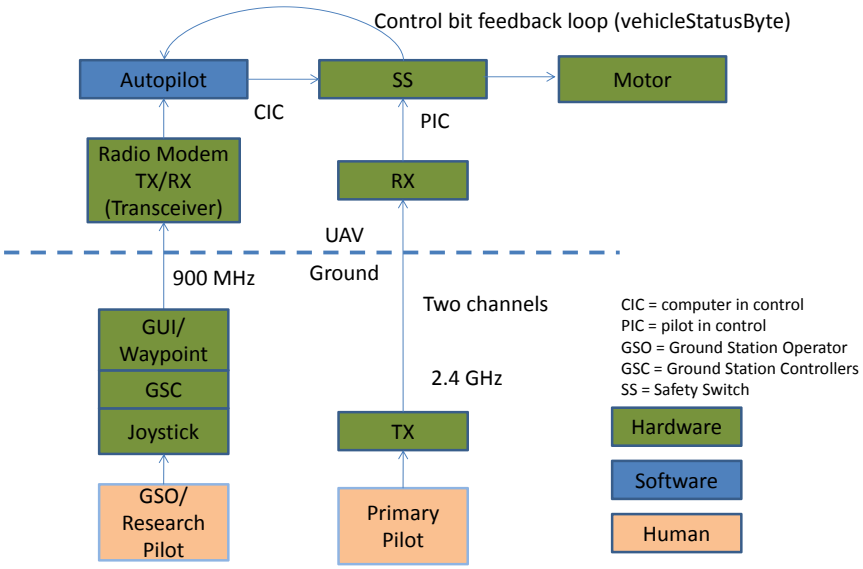



Figure 4: Ground station and UAV communication configuration with avionics

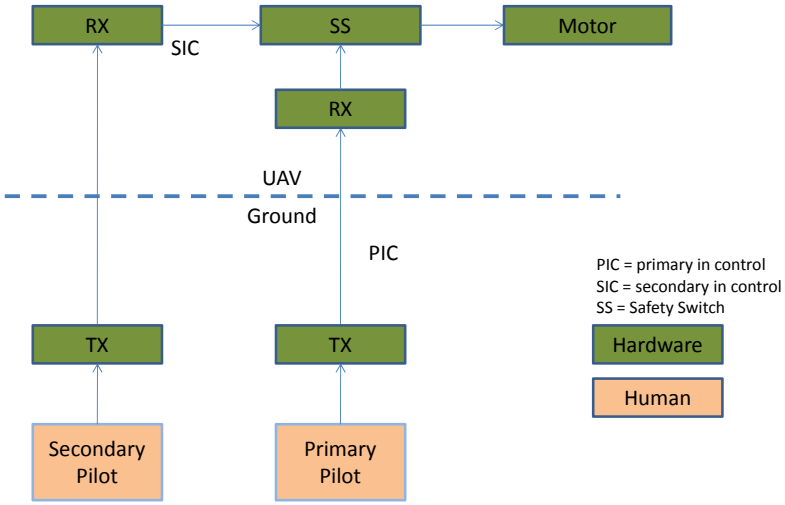



Figure 5: Ground station and UAV communication configuration without avionics

### 2.1.2 System Parameters

The UAV has an autopilot which takes inputs from sensors (including uploaded commands), and sends them to the Flight Management System (FMS), which updates the state and sends outputs to the actuators. The actuators move the control surfaces.

A downlink sends the following state information back to the GSC:

- Indicated airspeed (IAS), $V = (u, v, w)$, from the Pitot tube; they measure the magnitude and not the vector components, used as the true airspeed
- Absolute velocity (GPS; the altitude $w$ has lower precision because of the GPS), in feet per second with north and east components
- Position (GPS), in feet from origin with north and east components
- Acceleration (only the $y$-component is needed; this is relevant to the sideslip angle)
- Angle of attack (alpha), sideslip angle (beta)
- Orientation in Euler angles; relative to the Earth's magnetic field, in the LVLH frame (local vertical local horizontal)
- Angular velocity, $\omega = (p, q, r)$, in body axes – taken straight from the gyros
- Pitch, Euler angle, in radians
- Roll, Euler angle, in radians
- Heading, true heading, in radians
- Vertical airspeed, in feet per second
- Battery voltage levels (filtered using a low-pass filter)
- Commands the vehicle is receiving from the pilot
- Health status from some units; e.g., the Athena INS/GPS unit
- Any payload sensors

Angle of attack (alpha) and side slip (beta) are measuerd and collected but are unused in the FMS or the autopilot. They are measured using a 5-hole + 2 pitot tube. The 5 are for alpha, beta, and airspeed. The two are both static holes. One is the static hole for airspeed. The other, inside the aircraft, is the pressure transducer to measure altitude. All of this data is collected on a GS111M, which contains an Inertial Measurement Unit, an Aircraft Data Computer, and a GPS.

## 2.2 Flight Software Architecture

We now describe the overall architecture and relationship of components within the flight software (FSW). It has a layered architecture with several loosely-coupled *modules* implemented on the *Reflection Virtual Machine* (RVM). Section 2.2.1 describes the relationship between the execution layers. Next, Section 2.2.2 describes the high-level relationship between the virtual machine and the other components that make up a vehicle's execution platform. The internal data-flow within the *Autopilot* module is given in Section 2.2.3. Finally, an example configuration of modules and scripts for one particular mission is provided in Section 2.2.4.

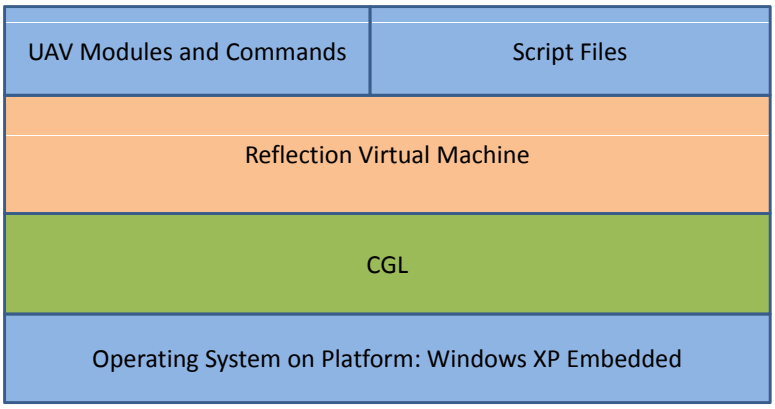



Figure 6: Execution layers of the Reflection system on a UAV.

### 2.2.1 Execution Layers of the Reflection Framework

The *Reflection* system refers to a multi-component, event-driven, configurable software package [36]. It is used to provide support for ground and flight control systems on autonomous platforms such as UAVs, as well as simulations of these platforms. It has a complex architecture with many different components. The architecture consists of several layers of execution. When the Reflection system is loaded on a UAV each component has its place in one of these layers. Figure 6 shows how these layers relate to each other in a bottom up fashion, each layer running on top of the one below. The base layer is the Windows XP Embedded operating system running on the hardware on the UAV. On top of this is the CGL, followed by the RVM.

The separation between the layers is not as simple as the diagram implies, but the Reflection system is defined on top of CGL. The modules that interface with the hardware, such as the sensors and telemetry/modem systems, and that define the virtual systems, such as the autopilot and failsafe modules of the UAV, are loaded on top of the RVM. Finally, scripts are used to dynamically define connections between modules and which modules are loaded into the virtual machine. The software running on the GSC consists of a similar setup. What follows is a description of these components and their relationship to one another.

### 2.2.2 Modules, Scripts, and the Reflection Virtual Machine

The Reflection system is a framework for facilitating communication between different and distinct modules. We see this relationship of the virtual machine and other subsystems in Figure 6. Each module represents a subsystem in a UAV system. Each is an interface with a physical piece of hardware or is itself a virtual system. For instance there are modules representing the autopilot, the modem interface, data-stores, and sensor and servo interfaces. Figure 7 illustrates the relationship of these modules with the other components.

The Reflection system allows for modules to be defined which can be used in a plug-and-play fashion. Different scenarios, missions, and simulations can be run with differing sets of modules and different parameters defined in those modules. Whereas the Reflection virtual machine provides the structure to facilitate communication, the modules and their data provide the content.

Modules are independent compilable sets of classes (defined in C++) which represent or model some aspect of a UAV. They are a loosely-coupled package of classes which internally define much of their functionality and data-flow within the module. Functionality that is not defined internally is derived through calls to the CGL physics and math libraries or from standard C++ classes. These packages can be compiled into a shared library (or DLL on win32 systems). Through hooks in the module code, the DLLs can be made to interact as interoperable components in the RVM.

The Reflection Virtual Machine is made up of two static components. The first is the Reflection framework. This is the code that provides event listeners and data routes between modules. The *CGL* is an independent but related "Common Graphics Library". It contains classes and code for a number of common calculations, trans-

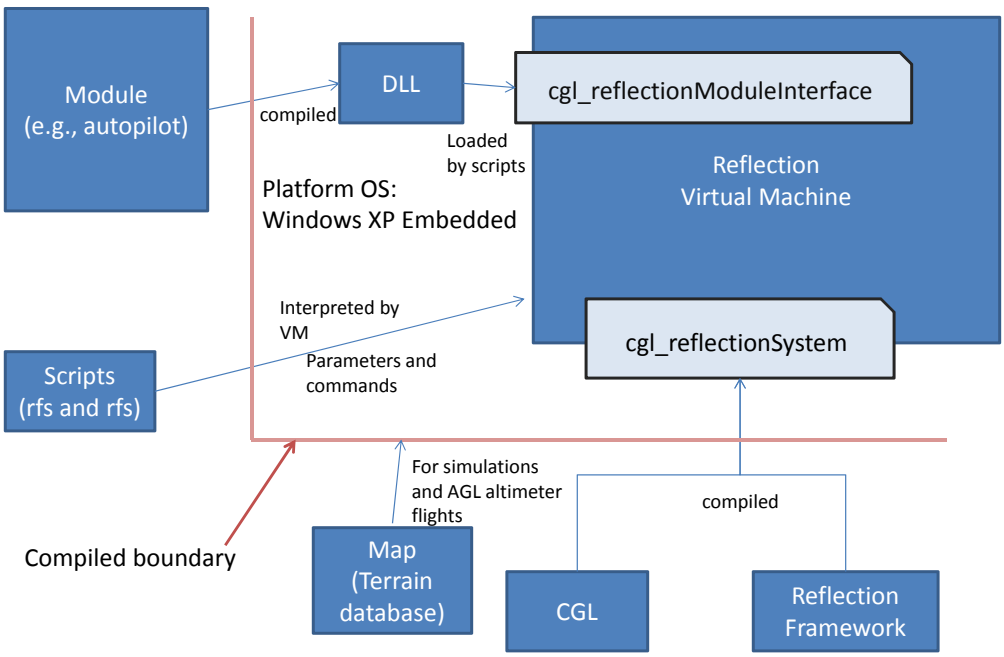



Figure 7: Relationship between modules, scripts, and the Reflection Virtual Machine.

formations, and operations. It supports both the Reflection framework as well as code defined in the modules. In support of the former, CGL defines operations for timing events and loading DLLs, all of which are available to the Reflection system. In support of the modules and module development we find resources more commonly found in a graphics library such as a physics engine and mathematics library. These have specialized libraries for doing navigation and aeronautical domain type calculations (again amongst many others).

The CGL and Reflection framework are brought together through the `cgl_reflectionSystem` class. Compiled together and running on the UAV system (the OS, such as Windows XP Embedded), these two frameworks define the Reflection Virtual Machine. The DLLs created from modules are loaded into the virtual machine through the `cgl_reflectionModuleInterface`. This interface dynamically loads the modules and is invoked by a definition command in a script. For any mission there are many modules that need to be loaded into the system. Once loaded each module exposes certain parameters and functions to the virtual machine. These exposed pieces are used by the virtual machine to define communication routes and create the flow of data through the system.

The data routing is user defined through a set of mission specific scripts. The scripts are written in a scripting language defined in the Reflection framework. The scripting language allows a user to define what objects (as DLLs) should be loaded into the virtual machine and what the relationship is between those objects. A relationship can be defined between the output parameter (or variable) of one module and the input variable of another, in effect creating an explicit data flow within the system. Each module defines what parameters and functions are exposed and the script writer is free to use those as needed. The scripting language also has utilities to define what data the mission should use. In the instance of a UAV mission a script can define the initial state of

the module and the sequence of commands (such as takeoff, fly to a waypoint, and land) the aircraft should follow. It further defines the information passed between the ground station and the UAV and the methods by which the ground station and the UAV interact, allowing for new commands to be inserted into the sequence or for complete control to be taken over by the ground system. A script can have an extension `.rfs` (Reflection script) meaning roughly that it defines a configuration of certain modules or state. A script can also be a static, reusable, function in which case it has the extension `.rff` (Reflection function). Scripts sit outside of the compiled boundary but are interpreted by the Reflection system.

The map data or terrain database can be loaded directly by the modules or through the scripts to provide geographic information to the ground-station, the UAV, or in a simulation. If the UAV is flying in a mode which uses an Above Ground Level altitude calculation the terrain data maps provide essential ground information.

### 2.2.3 Autopilot Module

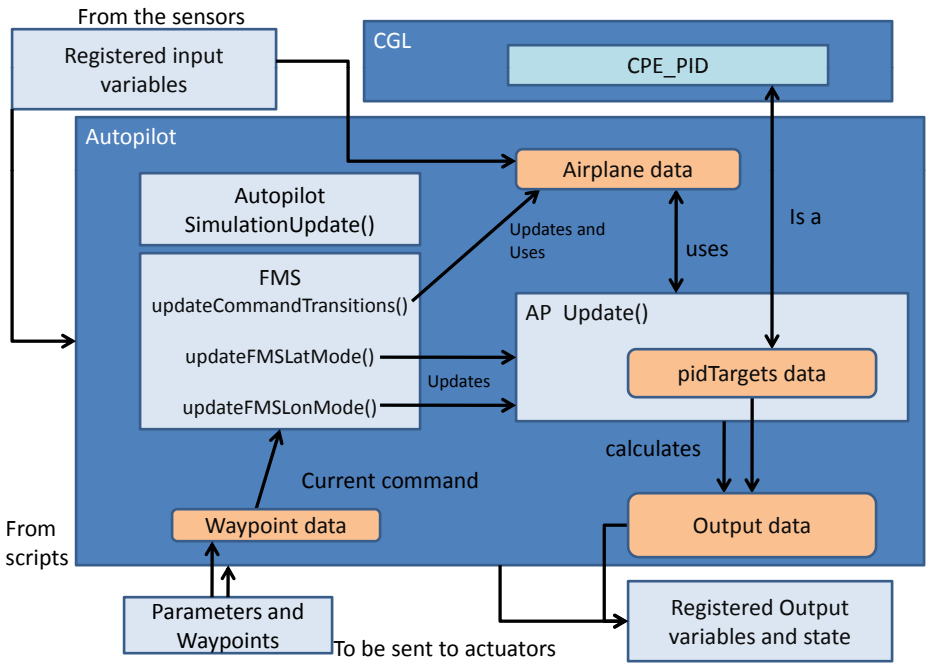



Figure 8: Updates and data flow in the autopilot module.

Figure 8 describes the internal behavior and data flow in the *Autopilot* module. This module is inserted (loaded) into the virtual machine and has relationships with a number of other modules. The Reflection Virtual Machine is able to emulate and facilitate the communication loops between sensors, controllers, and actuators. This figure represents the controller section which takes in data from the sensors and outputs data to the actuators.

As noted above, modules can register parameters and variables with the "outside" world through the scripts. The registered input and output boxes represent this set of data blocks. Data comes in through the registered input variables (as do commands for the system to evaluate). Data and state information is sent out of the system through a similar set of parameters and variables. For instance the sensor data is routed in to an airplane data structure which is mapped through the `autopilot` object defined in the scripts. The data is stored there for use by the controller and the flight management system. These subsystems are described further below. The registered input variables can also be used to engage or disengage any part of the autopilot or allow for the pilot to send commands directly to the UAV via a "stick" on the ground and the UAS's telemetry system. Further calls will be sent via the Reflection Virtual Machine to one of the registered functions that the autopilot exposes. This will cause that specific function to be evaluated. This is how the Reflection Event System interacts with the autopilot.

The *Autopilot* code consists of three major classes. The first is the `autopilot` class itself. This is the code that interfaces with the Reflection system and makes internal calls to the other two classes. The first three calls, when the autopilot is engaged, are to the *Flight Management System (FMS)* implemented in the `FMS` class. The next call is to the controller (*AP*) implemented in the `AP` class which actually sets the output variables for the actuators which will be communicated through the virtual machine.

The *Autopilot* module runs in a loop. Each loop iteration is driven by the virtual machine event system. At each loop, the timing of which is a configurable parameter via a script, the `autopilot` class initiates the subsequent calls to the related systems (`FMS` and then `AP`). At a macro level the *Autopilot* drives the *FMS* to check the status of the current command and then tells the *AP* system to react to any changes.

The `FMS` class updates the command status of the autopilot. It evaluates where the aircraft is with respect to the completion of the current command or reaching the current waypoint. Further it determines if it needs to transition to the next command, if in fact there is a next command. The *FMS* system also updates the lateral and longitudinal mode settings. This is general state information which allows the autopilot controller (the `AP` class) to determine what calculations need to be done in support of the aircraft meeting its goal (namely the completion of the current command). The list of commands that the UAV will follow is stored in the *FMS* system. Commands are represented by the class `Waypoint`. `Waypoint` contains information pertinent to the heading, location, speed that the aircraft needs to attain.

The update commands in the `FMS` class set state data that is accessible by the *AP* system. This data is a combination of PID controller information and mode information. The determination of what data should be provided from the *FMS* forward to the *AP* system is calculated based on defined parameters, the current command and modes being evaluated in the *FMS*, and the data store in the `airplaneData` structure. The `airplaneData` struct defines the current state of the aircraft: position North and East of the origin; roll angle; airspeed; vertical speed; pitch angle; heading; acceleration; and flight path angle. The PID controllers are represented by a class within the CGL library physics engine, `CPE_PID`. The PID variables represent a standard feedback-loop controller that takes sensor data and calculates some gain value to pass on to the actuators. They are stored as a block in the structure `pidTargets`. They will be updated again in the `AP` class.

The `AP` class iterates through different modes of operation: longitudinal, lateral, and speed. Each mode has a specific set of aircraft surfaces and controls that it can effect. Each of these modes is independent of the others. Depending on what command is currently under evaluation (via the *FMS*) and what phase that command is in, different calculations are made (different sections of code are run). For instance if the command in the *FMS* is currently `COMMAND_LAND`, different code would be run in the *AP* system depending on what phase (approach, glide, flare, taxistop) the UAV was currently undertaking. `COMMAND_LAND` is the only command which has multiple phases. The `AP` class updates the output variables. This data is then used by the Reflection system to drive the actuators and any other uses it may serve. This sequence is the repeated.

### 2.2.4 Mission Configurations

Figure 9 shows the components required for a particular mission, in this case Flight 070801 SAAV of the EAV aircraft using the avionics configuration and an autopilot. The Autopilot module defines the autopilot flight management system and controller. The gs11m is the sensor interface which routes data from the sensors into the autopilot. It also routes data to the ground station via the iavmodem interface and to the pcsfilter which filters position and altitude data for navigation. The ap_rcap module is an experimental rate controlled autopilot. The ap_failsafe module takes data from the modem and the autopilot and routes it to the sv203interface. The sv203interface is the connection to the actuators. The datastore module is a logging utility storing data from the sensors, commands, and outputs.

The components are combined together on the target platform, namely the virtual machine running on the EAV. Each module is compiled into a DLL. Each DLL represents a physical or virtual entity necessary for the operations of the mission. For any given mission there are any number of possible scripts, which define the overall makeup of the system and its data flow. The scripts define the relationships between the modules and tell the virtual machine which components are active in the system. In particular, the scripts define the frequency of the loop event calls to the modules and what functions within the modules need to be called. For completeness we also indicate the connection between the modules and the CGL libraries and physics engines, though this is

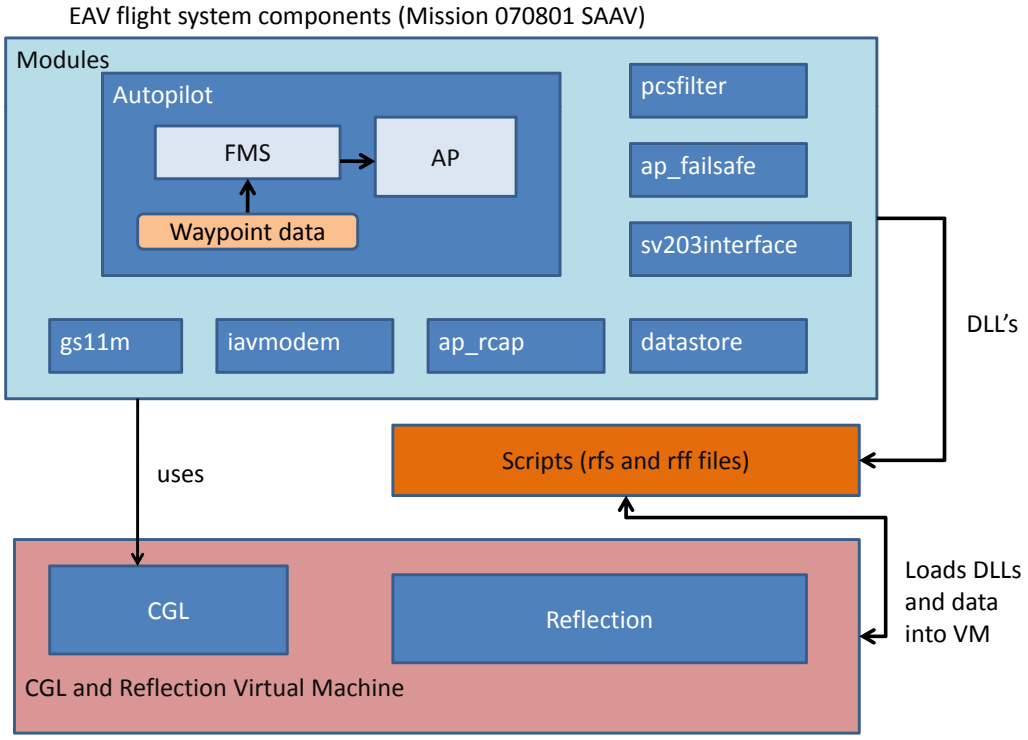



Figure 9: Example configuration of EAV flight system components: Mission 070801 SAAV.

unchanged between missions.

## 2.3 Flight Management System and Controller

We describe the calculation and dependency structure of the output value for aileron control[12] under the landing command in the autopilot module. We also briefly describe the calculation and dependency for the output value for the elevator control. First, we describe the background of the autopilot module and put the aileron calculations and the landing command in context.

### 2.3.1 Control Modes

The aileron calculation under a landing command is only one possible route through the code. The computation of the aileron output is done in every command. Sometimes the computation is the same as we describe here. Other times different modes are considered which will cause different cases to be evaluated along the computational sequence. However, at each iteration of the call to the autopilot subsystem, the sequence of function calls is the same.

The `Autopilot` class is the interface between the Reflection Virtual Machine and the additional classes that compose the rest of the autopilot module, `FMS` and `AP`. The `Autopilot` class drives the calls to both the `FMS` class and the `AP` class. The calculation of the aileron output is done completely within the context of the `AP` class under the landing command. In other cases there are state updates made in the `FMS` class that affect the aileron output.

The `FMS` class has the responsibility of transitioning between commands and then setting `FMSLATMODE` and `FMSLONMODE` based on what command is currently under evaluation. The `FMSLATMODE` and `FMSLONMODE`

[12]The precise configuration of control surfaces depends on the aircraft. The code we have looked at here is specific to the EAV, but the general principles are common to all autopilots.

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

Figure 10: Commands, FMS Lateral modes and FMS Longitudinal modes relevant to the EAV aircraft. The heading is the prefix to each element in that column.

in turn will cause a specific APLATMODE and APLONMODE to be set, respectively. There is a correspondence between these modes. A specific FMSLATMODE determines a specific APLATMODE. The case is similar for FMSLONMODE and APLONMODE (and APSPDMODE, the speed mode which relates to the throttle control, but we do not consider that here). In Figure 10 we see a listing of all the commands and modes, both lateral and longitudinal, in the FMS system. The landing command only invokes a few of the lateral and longitudinal commands. Only the lateral mode TRACKTOWAYPOINT will be considered with respect to the aileron output control calculation.

The AP system is responsible for generating the values which are routed to the actuators that control the aircraft control surfaces. Once a specific set of AP modes have been set, via the FMS class, the AP.Update command can be called. The Update command in the AP class is invoked from the function SimulationUpdate in the Autopilot class. Update is called after all the mode and command transition functions in the FMS class have finished. The Update command operates by progressing, linearly, through a series of PID "loops". These are feedback loops, not programming language control structures.

Each PID loop is a switch statement (case statement) evaluating the current APLATMODE, APLONMODE, or APSPDMODE (speed mode), depending on what the PID loop is controlling: lateral, longitudinal, or throttle, respectively. As such, each PID loop will affect either a lateral, longitudinal, or speed control surface. Each PID loop will result in a value which will be output, or used in a calculation of the eventual output, to the actuator of a single control surface of an aircraft. The specific calculations for lateral surfaces, given the current command in question, was set up in the FMS class calculations. In other words a specific APLATMODE was set in FMS that will now, in the AP system, be the case considered within each of the PID loops. Similarly longitudinal modes invoke a specific APLONMODE. A PID loop that effects a lateral surface will only consider APLATMODEs. Again, there is a similar situation for longitudinal surfaces and APLONMODEs.

The PID loops we will consider here will all affect the output to the aileron actuator. Hence they are all lateral PID loops. This is based on the code for the EAV autopilot system. Specifically we will look at the calculation of the aileron PID loop and the PID loops upon which it depends, namely the roll and heading PID loop. There are no direct outputs (to the actuators or other modules) from the roll and heading PID loop. They do not directly affect a control surface. However, the results from these PID loops will be used in the calculation of the aileron output. The other control surfaces are elevator, rudder, and throttle[13]. Each is considered under different PID loops. Each PID loop has a switch statement evaluating the corresponding modes. Figure 11 gives a visual representation of the landing phases and its transitions. Each of the phases is marked by a set of transition criteria. The transition criteria are defined in the code but make use of system parameters which are set via the script files.

[13]There is additionally a flap control surface, however in the EAV system this is unused and initial values assigned to it are simply passed through the code without modification.

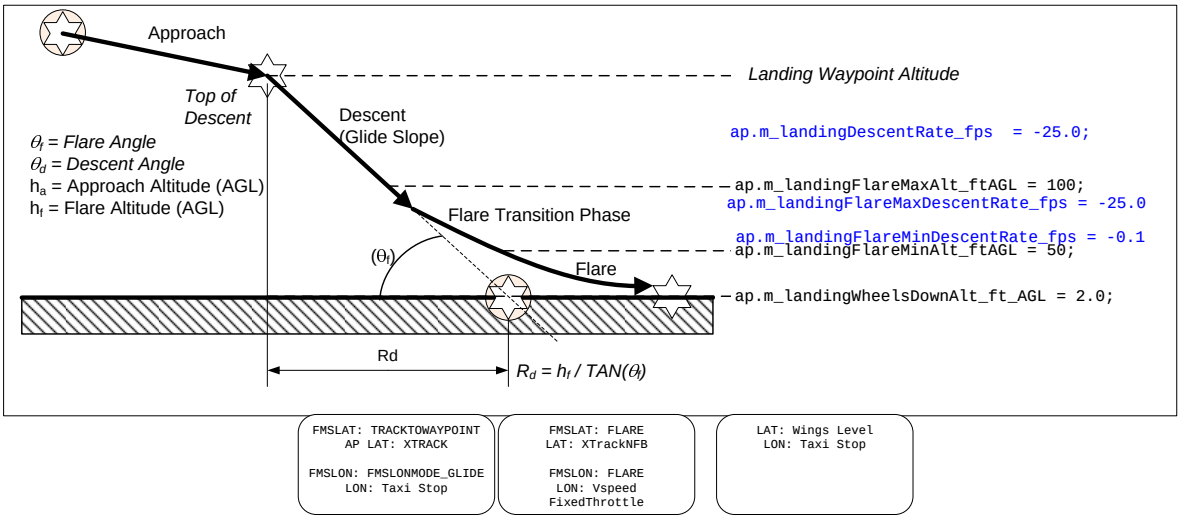



Figure 11: The phases invoked during the COMMAND_LAND command (from [33]).

### 2.3.2 Aileron High-level Control Sequence

We describe one particular sequence of calculations through the controller, which is executed under specific mode and command conditions. This particular calculation is done in three of the four phases of a landing command, namely Approach, Descent (or Glide), and Flare, but not Taxistop. These phases all fall under the landing command, and are represented in the code as transitions between modes. There is no variable or global structure to indicate that, for instance, we are in the Descent phase.

In order to initiate this calculation the following modes and command must be set:

- Command = `COMMAND_LAND` and
- `FMSLATMODE` = `TRACKTOWAYPOINT` and
- `APLATMODE` = `LATMODE_CROSSTRACK` or
  `APLATMODE` = `LATMODE_CROSSTRACK_NOFLYBACK`

There are no preconditions, other than the setting of the `APLATMODE`, derived from the FMS system for aileron calculations.

The variable $m\_aileron\_m1p1$ holds the value that will be routed to the aileron actuator through the Reflection framework. The result stored in that variable is dependent on the calculations of roll angle, heading, and crosstrack and depends as well on a number of state and position variables (waypoints and current position). The calculation of the value stored in $m\_aileron\_m1p1$ traces eventually back in to the `FMS` class and the `Autopilot` class.

The dependencies of variables are seen in Figure 12.

The following is an outline of the steps necessary for the calculation of the aileron control variable within the `AP` class.

- Calculate the desired heading from the source, destination, and current locations and adjusting this value by the cross track calculation.
- Calculate the desired roll angle from the desired and current heading.
- Calculate the value sent to the aileron actuator from the desired and current roll angle.

Each is dependent on the previous. They correspond to the heading, roll, and aileron PID loops, respectively.

The calculations in the heading PID loop result in the computation of $m\_pidTarget.m\_desiredHeading\_rad$, which has the following properties:

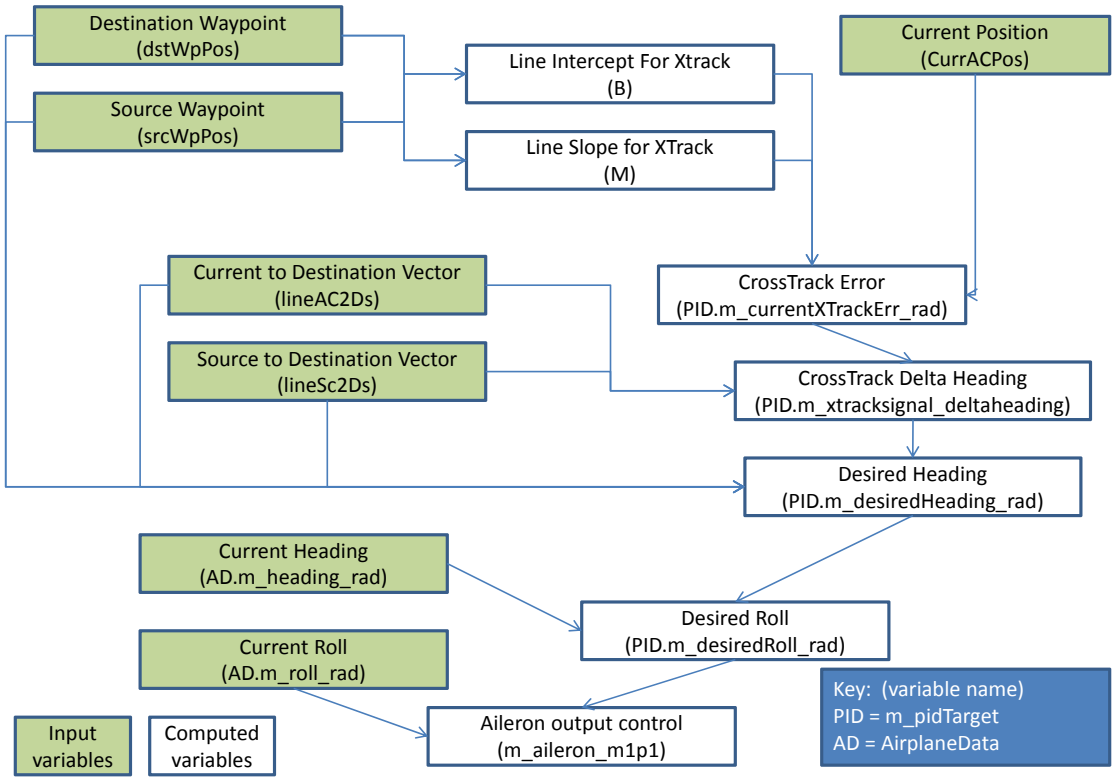



Figure 12: Computational dependencies in the calculation of aileron control Variable $m\_aileron\_m1p1$.

- It is the heading angle between the source and destination waypoints adjusted by the delta heading resulting from the crosstrack error.
- The crosstrack error is the result of a PID update command (explained below): the sum of the P, I, and D components based on the error derived from the distance the aircraft currently is to the line on which it is supposed to be.

The result of the roll PID loop is stored in $m\_pidTarget.m\_desiredRoll\_rad$.

- It, too, is the result of a PID update command based on the error derived from the difference in the aircraft's current heading and the desired heading.

The aileron PID loop output is stored in $m\_aileron\_m1p1$.

- It is a result of a PID update command based on the error which is the difference between the aircraft's current roll and the desired roll.

**PID Update**

The PID Update calculations, in general, have the following result form:

$$result = PGain * Error + IGain * IState + DGain * (lastError - Error)$$

where

- PGain, IGain, DGain, are defined values specific to the actual PID controller.

and

$$IState = \begin{cases} iMax & if\ IState + Error > iMax \\ iMin & if\ IState + Error < iMin \\ IState + Error & otherwise \end{cases}$$

The initial value of $IState$ and the values of $iMax$ and $iMin$ are defined parameters (different for each PID controller). $lastError$ is the previous $Error$ value passed as a parameter to the calculation and is stored as part of the state for the next calculation.

### 2.3.3 Mathematical Calculations for the Aileron

We describe the calculation of each of the preceding variables working back from output (the aileron control variable) to input (the aircraft state and the flight plan). We show the calculation of each variable and then describe the dependencies of that variable. The order is as follows:

- $m\_aileron\_m1p1$
- $m\_pidTargets.m\_desiredRoll\_rad$
- Geometric calculations based on position
- $m\_pidTargets.m\_desiredHeading\_rad$
- $m\_pidTargets.m\_xtracksignal\_deltaheading$

Each of these is now described.

**Variable** $m\_aileron\_m1p1$

The variable $m\_aileron\_m1p1$ is calculated in the following fashion.

$$\begin{aligned} m\_aileron\_m1p1 \quad = \quad & (RollErr2Aileron.mPGain) * Error \\ & +(RollErr2Aileron.mIGain) * (RollErr2Aileron.m\_IState) \\ & +(RollErr2Aileron.mDGain) * (lastError - Error) \end{aligned}$$

where

$$Error = airplanedata.m\_roll\_rad - m\_pidTargets.m\_desiredRoll\_rad$$

$$m\_IState = \begin{cases} m\_iMax & (m\_Istate + Error) > mIMax \\ m\_iMin & (m\_Istate + Error) > mIMin \\ (mIstate + Error) & Otherwise \end{cases}$$

$Error$ becomes $lastError$ in the next iteration.

The variable $m\_aileron\_m1p1$ is regulated to restrict the value to be between -1 and 1 via the function `REGULATE_CAP`.

**Variable** $m\_pidTargets.m\_desiredRoll\_rad$

The variable $m\_aileron\_m1p1$ initially depends on the calculation of the member variable $m\_pidTargets.m\_desiredRoll\_rad$ ($m\_desiredRoll\_rad$ for short) which is calculated as follows:

$$\begin{aligned} m\_desiredRoll\_rad \quad = \quad & (HeadingErr2Roll.mPGain) * Error \\ & +(HeadingErr2Roll.mIGain) * (HeadingErr2Roll.mIState) \\ & +(HeadingErr2Roll.mDGain) * (lastError - Error) \end{aligned}$$

where

$$Error = airplanedata.m\_heading\_rad - m\_pidTargets.m\_desiredHeading\_rad$$

and the other variables are calculated as above.

The variable $m\_pidTargets.m\_desiredRoll\_rad$ is regulated to be between the value of plus or minus $m\_bankLimit\_rad$, a defined parameter.

**Geometric Calculations**

The variable $m\_desiredHeading\_rad$ is dependent on a number of other variables. In particular a number of arrays and vectors indicating positions of interest.

$$\begin{aligned} currACPos[2] &= \{airplaneData.m\_pos\_north\_ft, airplaneData.m\_pos\_east\_ft\} \\ dstWpPos[2] &= \{m\_waypointNorth\_ft, m\_waypointEast\_ft\} \\ srcWpPos[2] &= \{m\_prevwaypointNorth\_ft, m\_prevwaypointEast\_ft\} \\ lineAC2Ds &= \{dstWpPos[0] - currACPos[0], dstWpPos[1] - currACPos[1]\} \\ lineSc2Ds &= \{dstWpPos[0] - srcWpPos[0], dstWpPos[1] - srcWpPos[1]\} \end{aligned}$$

where the content of the arrays and vectors above is

- $airplaneData.m\_pos\_north\_ft$ and $airplaneData.m\_pos\_east\_ft$ represent the aircraft's current position.
- $m\_waypointNorth\_ft$ and $m\_waypointEast\_ft$ are the values of the destination waypoint.
- $m\_prevwaypointNorth\_ft$ and $m\_prevwaypointEast\_ft$ are the values of the source waypoint.

The vector $lineSc2Ds$ is normalized and used to differentiate the source and destination waypoints. This is done to avoid the possibility of two subsequent waypoints having the exact same location. If this were the case the result would be a vector with 0 magnitude and possibly cause a division by zero condition.

$$\_Sc2Ds\_ = cglVec\_Normalizev(Sc2Ds, Sc2Ds)$$

$cglVec\_Normalizev$ is a vector normalization function that takes an array of reals as opposed to a vector object. If the source and destination are essentially the same point then the source position is adjusted and $lineSc2Ds$ is recalculated.

$$\begin{aligned} &if\ (fabs(\_Sc2Ds\_ < .00000001)) \\ &\qquad srcWpPos[0] = m\_prevWaypointNorth\_ft + 1.0 \\ &\qquad lineSc2Ds = \{dstWpPos[0] - srcWpPos[0], \\ &\qquad\qquad dstWpPos[1] - srcWpPos[1]\} \\ &\qquad \_Sc2Ds\_ = cglVec\_Normalizev(Sc2Ds, Sc2Ds) \end{aligned}$$

Finally a couple of calculations are made on these values to be used in the calculation of $m\_xtracksignal\_deltaheading$ and $m\_desiredHeading\_rad$:

$$\begin{aligned} acrossb &= cglVec\_Cross(lineAC2Ds, lineSc2Ds) \\ adotb &= cglVec\_Dot(lineAC2Ds, lineSc2Ds) \\ angle &= acos(cglVec\_Dot(lineAC2Ds, lineSc2Ds)) \end{aligned}$$

**Variable** $m\_pidTargets.m\_desiredHeading\_rad$

The variable $m\_pidTargets.m\_desiredRoll\_rad$ depends on the calculation of member variable $m\_pidTargets.m\_desiredHeading\_rad$ ($m\_desiredHeading\_rad$ for short).

The variable $m\_desiredHeading\_rad$ is calculated as follows:

$$m\_desiredHeading\_rad = \begin{cases} cglGeom\_calculateHeadingAngle\_rad(dstWpPos, srcWpPos) & fabs(angle) > PI/2.0 \wedge \\ \quad +m\_pidTargets.m\_xtracksignal\_deltaheading & m\_latmode! = \\ & CROSSTRACK \\ & \_NOFLYBACK \\ cglGeom\_calculateHeadingAngle\_rad(srcWpPos, dstWpPos) & otherwise \\ \quad +m\_pidTargets.m\_xtracksignal\_deltaheading & \end{cases}$$

**Variable** $m\_pidTargets.m\_xtracksignal\_deltaheading$

Figure 13 contains a visual representation of a track to waypoint lateral mode and the crosstrack heading adjustment. Track to waypoint adjusts the heading of the aircraft to fly the line defined between the previous and current waypoint. If the aircraft has passed the current waypoint the heading is reversed. The crosstrack heading adjustment is done in the case when the aircraft is not exactly on the heading defined by the previous and current waypoint. The crosstrack error is a calculation done to direct the aircraft towards its next waypoint by correcting its heading.

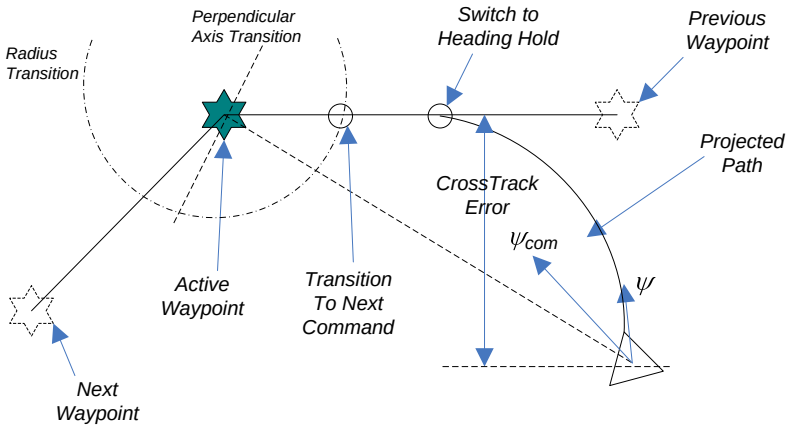



Figure 13: Depiction of tracking to a current waypoint and the crosstrack component of the heading calculation (from [36]).

The variable $m\_xtracksignal\_deltaheading$ is calculated in the following manner.

$$\begin{aligned} m\_xtracksignal\_deltaheading \quad = \quad & fabs((CrossTrackErr2Heading.mPGain) * Error \\ & +(CrossTrackErr2Heading.mIGain)* \\ & \quad (CrossTrackErr2Heading.m\_IState) \\ & +(CrossTrackErr2Heading.mDGain)* \\ & \quad (lastError - Error)) \end{aligned}$$

where the $Gain$ variables are as above, $CrossTrackErr2Heading$ is a pid controller variable, and the $Error = m\_currentXTrackErr$ which is calculated as follows:

$$m\_currentXTrackErr = cglGeom\_CalculateDistanceToLine(currACPos, lineM, lineB)$$

where $lineM$ and $lineB$ are pass-by-reference doubles computed in the following function:

$$cglGeom\_CalculateLineSlopeIntercept(\&lineM, \&lineB, srcWpPos, dstWpPos)$$

The variable $m\_xtracksignal\_deltaheading$ is regulated to be within between plus or minus $g\_XTrackMaxCorrectionAngle\_rad$, a defined parameter. The variable $m\_xtracksignal\_deltaheading$ is also adjusted based on the calculated values of $acrossb$ and $adotb$.

$$m\_xtracksignal\_deltaHeading = \{ \quad -m\_xtracksignal\_deltaHeading \quad if\ acrossb > 0$$

$$m\_xtracksignal\_deltaHeading = \begin{cases} -m\_xtracksignal\_deltaHeading & \begin{array}{l} if\ adotb < 0 \\ m\_latmode! = \\ CROSSTRACK \\ \_NOFLYBACK \end{array} \end{cases}$$

The variable $m\_desiredHeading\_rad$ is regulated plus or minus $\pi$.

### 2.3.4 Elevator High-level Control Sequence

The elevator output value directly controls the elevator actuators. It is calculated from the state of the aircraft and conditions set in the current (i.e. the destination) waypoint. The sequence of calculations is done in two steps. The first involves calculation of the altitude error and hence the pitch of the aircraft. The second involves using the pitch to calculate the adjustment to the elevator control surface.

The altitude error is calculated by subtracting from the current altitude, derived from the aircraft state, the desired altitude at the destination waypoint.

$$altitudeError\_ft = airplaneData.m\_pos\_altitude\_ft - m\_desiredAltitude\_ft$$

The value $m\_pidTargets \rightarrow m\_desiredAltitude\_ft$ is set each time a new waypoint is loaded and represents the value that must be obtained upon reaching that waypoint.

The altitude error value is then used to calculate the error in the current pitch of the aircraft. This is done via a PID controller. The specific PID controller, $m\_pid\_AltitudeErr2Pitch$ takes $altitudeError\_ft$ as input. This is assigned to the state variable $m\_desiredpitch\_rad$.

$$\_desiredpitch\_rad = m\_pid\_AltitudeErr2Pitch \rightarrow Update(altitudeError\_ft)$$

The aircraft state additionally stores the current pitch of the aircraft. This is used in conjunction with the desired pitch to calculate the error in the pitch.

$$pitchError\_rad = airplaneData.m\_pitch\_rad - m\_pidTargets \rightarrow m\_desiredPitch\_rad$$

The pitch error is then used to calculate, using another PID controller, the elevator control output value.

$$m\_elevator\_m1p1 = m\_pid\_PitchErr2Elevator \rightarrow Update(pitchError\_rad)$$

# 3 Safety Considerations for the Target System

## 3.1 Preliminaries

We consider the safety of two UASs — the EAV and the Swift. The Swift is still being developed (the EAV[14] team has applied for a Certificate of Authorization (COA) from the FAA and originally expected to get this in early 2011; however, this is on hold until they can provide an airworthiness statement). The Swift has recently been modified[15] by the manufacturer[16], MLB, and the safety requirements will likely change (e.g., they had originally planned to use a parachute as CMS, but following feedback from the AFSRB they will probably drop this).

The EAV (a Cessna) has flown before, both at Crow's Landing and Moffett Field. The two UAVs have much in common, in particular:

- The Reflection architecture [36]
- Avionics, including most of the autopilot
- Development methodology: simulation and testing configurations, for both software and hardware, hardware testing and component sizing methods
- Underlying theory
- Regulatory framework

[14] Note that *EAV* refers to both the name of the Code TI research group, and one specific UAS.

[15] The original aircraft was not designed to be a UAV. Also, the PI wanted double redundancy, and split control surfaces for distributed control.

[16] Or rather the predecessor company, Brightstar.

They differ in the specifics of the range safety analysis, and the actual parameters and configuration. Therefore we will concentrate for now on those common parts of the Swift and EAV.

- There are some safety issues which are specifically related to the Swift, however. The EAV control surfaces are the aileron, elevator, rudder, and throttle, whereas the Swift has 8 distributed control surfaces which will provide the functionality of elevons (combined elevators and ailerons) and flaps. They will also carry out an experiment using a morphing wing.
- The Swift is electrically powered, with large and explosive battery packs (which requires temperature sensors to be installed). Since it is a heavier aircraft it has high kinetic energy ($mv^2/2$). Also, it is very high performance, with a high glide ratio and lift/drag ratio. This means that if unpowered, in the worst case, it will glide very far.
- The EAV (a Cessna) has a complex stiff structure, but the Swift is more flexible, is thin, and is made of a uniform material, so has vibration modes (which is not the case for the EAV); however, it is anticipated that *flutter* – interaction between the airstream and flexible (structural) modes – will not be an issue for the Swift. However, close to $V_{NE}$[17] there might be structural failure (buckling of structure, and fatigue).

At this point there are several aspects of the UAS which we will not consider, such as the ground control software, the communication link, operating procedures (at least not formally), and wider NAS system integration issues.[18]

## 3.2 Regulatory Framework

The EAV group fly UAVs, first at the auxiliary site at Crow's Landing for qualification testing, and then at Moffett Field. Although the airspace at Moffett Field is controlled by NASA, they still need to apply to the FAA for a COA.

In fact, any aircraft which flies in the NAS, including Moffett Field, must comply with FAA regulations. However, since NASA can certify aircraft airworthiness and pilot qualifications, there is some latitude in certain areas. Moreover, since at this time there are no plans to fly in the wider NAS, and since the exact status of FAA regulations for UASs is under review, we do not consider FAA regulations to be directly applicable. [19]

The approval procedure is to first present to the Airworthiness and Flight Safety Review Board (AFSRB, Code Q), then get an airworthiness statement (stating that there is reasonable belief that the aircraft and its systems will not fall to pieces). They then apply for an FAA COA. The COA does not require a safety or risk analysis. Next, they present to the Flight Readiness Review Board (FRRB, Code JO), and then get a flight release statement.

Therefore, although the overarching safety requirements stem from NASA NPR 7900.3C [47], these trace down to the relevant requirements which apply at NASA Ames, specifically APR 1740.1 [48] (airworthiness) which, in turn, is reflected in operational requirements at the divisional level (Code JO), namely JO-3 (Flight Operations Manual Moffett Federal Airfield) [1].

Safety requirements can therefore be derived from:

- APR 1740.1
- JO-3
- COA guidance documents
- feedback from the review panels: AFSRB and FRRB
- range safety analysis

[17] $V_{NE}$ is the *Never Exceed Speed*. In general, there are many V-speeds which have a bearing on safety, but not all are relevant for small aircraft, and many are left to the pilot's discretion.

[18] Safety is ultimately a hierarchical concept, and we hope to explore these wider issues in future.

[19] Nevertheless, although not directly applicable to this target system, we do want to consider FAA regulations in our project.

All of these ultimately satisfy NPR 7900.3B.

In common with other NASA projects, procedural requirements such as NPR 7150.2A (software engineering) and NPR 7123.1A (system engineering) are also mandated. These NPRs reference corresponding NASA standards, such as NASA-STD-8719.13 (NASA Software Safety Standard). The NASA regulatory requirements are summarized in Appendix D.

## 3.3 Contingency Management

A true Flight Termination System (FTS) must be highly reliable[20]. Very few termination systems would qualify with this level of reliability. Instead, the EAV team uses a Contingency Management System (CMS), which is less reliable, but the use of two independent CMSs is almost as good. All that is required is to keep the aircraft on range, and crash "inside the box".

They use three CMSs which are usually applied in the following order of precedence:

1. If there is a failure of the primary pilot system, then go to the secondary system, which is redundant and on a different channel
2. If that fails, engage the onboard autopilot
3. If that fails, force a spiral descent to impact

Before each flight the team determines the actual order of the CMSs, based on the particular experiment they are carrying out.

## 3.4 System Requirements Relevant for Safety

We now discuss the high-level requirements which the EAV team have identified as being relevant for safety of the Swift UAS, but first note several aspects of safety relevance:

(i) The team has not carried out a comprehensive hazard analysis or exhaustive safety analysis, since most components are off-the-shelf[21]. Therefore they do "airworthiness-light" and rely more on *range safety* (and so downplay software). The boards are satisfied for the COA if it can be shown that there is a low probability of a crash out of range.

(ii) Another reason that the team is not detailed or comprehensive in the safety analysis is simply because this is a low to mid-TRL research project, which does not justify that level of detail. Also, since the Swift UAS builds on a heritage of other UASs, such as the EAV and XSCAV, there are requirements and implicit knowledge which are not always documented.

It could be argued, therefore, that the safety case need only consider the range safety, since as the "last line of defense", it is this that ultimately ensures the safety of individuals on the ground.

However, we adopt a wider notion of safety and consider safety aspects of the vehicle as well, including the software (which acknowledging its auxiliary status), even though the failsafe mechanisms are usually implemented in hardware (which is simpler and therefore considered more trustworthy than software). Although the autopilot itself is auxiliary, flight testing beyond line of sight would rely on the autopilot.

(iii) The failsafe mechanism is programmed into the receiver, which can be set to perform a simple action on failure, e.g., on fail set low throttle and maintain a constant state for spiral descent. It can be tuned on the ground. The autopilot does this in software, but that uses feedback to close the loop on the altitude to maintain a constant roll angle and set a low throttle.

The following high-level requirements are relevant[22] to safety:

**R8** Hinge moments must be calculated to show margin of safety.

**R10** Actuator response must be at least $4\times$ fastest response mode, desire to be $10\times$ fastest mode

[20] See `http://kscsma.ksc.nasa.gov/Range_Safety/NASALinks.html` for definitions.

[21] Risk analysis has been conducted for the EAV but not for the Swift UAS.

[22] In the absence of a hazard analysis, we do not claim these are comprehensive.

**R12** Actuators must not interfere or collide with existing structure.

**R24** Allow direct control of the aircraft.

- **R25** Direct pilot control system must meet flight critical functions requirements
- **R26** Critical flight system must be dually redundant
- **R27** Must provide two separate, independent, redundant channels to the pilot
- **R28** Pilot must be able to switch between pilot in control (PIC) or computer in control (CIC)
- **R29** Direct pilot link must provide control for entire flight experiment (range of around 1 mile)

**R39** Support fuselage functions.

**R44** Power the direct pilot link.

**R51** Provide independent CMS for flight termination.

- **R53** CMS[23] must terminate the flight in a timely manner.
- **R54** CMS must not let the aircraft drift outside the range safety area.
- **R56** CMS must operate on a frequency separate from the rest of the system.

The requirements stated above only implicitly indicate the hazard(s) which are mitigated or eliminated. To validate the relevance of these system requirements to safety, we intend to apply our safety methodology such that the hazards which are addressed by the respective requirements are explicitly identified and documented. However, this is not the primary scope of the safety analysis presented in this report and the specific aspect of tracing the safety relevant system requirements to the relevant hazards is left as an aspect of future work.

Note that:

(i) R8 requires a margin of safety of 1.5, that is, the aircraft must be overdesigned to handle 1.5 the performance envelope.

(ii) R39 is to make sure it can carry the load; they rely on the manufacturer to carry out the appropriate static load tests.

(iii) R10 is to ensure that the elevators are faster than the oscillations — a property of the eigenvalues (which are obtained by system ID), and characteristics of modes (frequency, amplitude, and damping).

(iv) R54 ensures range safety while turning during a spiral descent.

(v) R8 and R10 involve actuator sizings.

In the subsequent sections in this report, we will present safety requirements derived from the application of our safety methodology to the target system.

# 4 Heterogeneity in Safety Information

In this section, we describe the external non-formal sources of information involved when considering safety of the Swift UAS. *Non-formal*[24] here refers to those sources of safety-relevant information which have not been derived using formal methods. We consider external sources that provide *assumptions*, *evidence* and *context*. Some of these are used to derive (i.e., justify) values, deflection angles, and throw ranges (servo ranges of motion) that appear as parameters in the software (e.g., as PID gains).

**Procedural, Safety and Development Standards** NPR 7900.3B, APR 1740.1, AFSRB certificate (the aircraft must not weigh more that 150 kg or fly faster than 170 kts.). See Appendix D.

[23] The actual requirement refers to a parachute.

[24] We prefer this to "informal" and its implications of lower standards, since rigorous engineering and mathematical analysis are still used.

**Procedures** As distinct from procedural standards, the UAS team's procedures describe what they do on the range, before, during, and after flights, including maintenance, as well as the roles played by the members of the flight team.

Flight-day procedures: responsibilities of flight manager, primary and secondary pilots, ground station operator, systems safety officer and safety observer; who knows what, who does what when; check-in, pre-flight briefing, airfield systems set-up, pre-flight safety procedures and checklists, then the actual flight tests (nominal flight plan), limited by range and other constraints, with refueling if necessary, followed by shutdown procedures/checklists.

There are specific procedures/checklists for multiple flights.

Before each flight, they must rotate the vehicle to calibrate the heading because of hard iron effects in the magnetometer. This is documented in the procedure manuals.

They prefer the pilot to bring her down "for safety" since it is not clear what the autopilot would do if there was a sudden gust of wind. However, the landing is usually semi-automated with the autopilot making the final ascent, lining up with the runway, and the pilot then taking over before the flare maneuver.

**Mathematical derivations/theory** The mathematical theory of aerodynamic stability and control is used to derive certain parameters that govern the safe operation of aircraft. These numbers are incorporated into the control law gains.

It is important to understand the assumptions on which these derivations rely. Some assumptions are used for simplification.

We can distinguish the derivations themselves from the theorems which justify these derivations. Evidence for the latter can be given by linking to actual formal proofs of these results, or by citing references to papers and textbooks.

An example of a simplifying assumption is that most autopilots are designed with decoupled dynamics, that is, linear independence of latitudinal and longitudinal models. This works if you assume small angles (i.e., with big changes this assumption breaks down and the controller will not be stable).

For fixed wing aircraft, standard simplifying assumptions such as these are valid. Moreover, the pilots have never noticed any non-linearities during flights at Crow's Landing which would call these into question.

Modes and poles are derived to determine how the aircraft oscillates. We know that the Swift is governed by standard equations[25], so we can solve the appropriate linear dynamic system

$$\dot{x}(t) = A \cdot x(t) + B \cdot u(t)$$

where $x(t)$ is the state vector at time $t$, $A$ is the system matrix, $B$ is the control matrix, $u$ is the input (control) vector, to give

$$x(t) = A_1 \cdot e^{\lambda_1 t} + A_2 \cdot e^{\lambda_2 t} + ...$$

where the $A_i$ are the eigenvectors and the $\lambda_i$ are the eigenvalues. We assume we can ignore the smaller terms.

Note that this LTI is linearized for one specific trim condition, and it is assumed that this it is good for a range of flight.

The eigenvalue solutions come in complex conjugate pairs $\lambda = n \pm i\omega$, and each corresponds to a natural mode. If the eigenvalues cross the imaginary axis then the aircraft will be unstable.

Standard theory is also used to convert between various representation formats, such as Euler and wind angles. Another example is the conversion from normalized coordinates (-1, +1) to deflection angles of actuators. However, the angles themselves do not matter. Rather, it is their derivatives, in particular the partial derivative $\partial\theta_{ACT}/\partial u_{NC}$, i.e., the rate of change of the actual angle of the actuator with respect to the input signal $u$ in normalized coordinates.

[25] Although not a traditional aircraft, it has been established that flying wings are characterized by the same theory.

**Range Safety Calculations** The expected casualty rate, $E_C$, is calculated by the RSO, and is based on:

- probability of impact
- protection factor
- lethal area (approximately: wingspan + 2ft)
- population density in persons per nautical square mile

Typically a maximum population density is back-calculated from an $E_C$ of $1 \times 10^{-6}$ per flight hour.

**Expert opinion** One question in any safety case is what level of detail to present. This relates, in part, to the question of who is reading the safety case; what is obvious and reasonable to a Subject Matter Expert (SME), may be quite unclear and in need of justification to a non-expert.

Since a safety case is arguably primarily a means of communication we believe that revealing areas where expert judgment is required is important, so we aim to make explicit opinions and their rationale as well as any assumptions on which they rest.

Such decisions are rarely documented, and might not even be clear to other team members, even if they appear reasonable.

Examples of expert judgment might be that a parameter is within safe bounds, that an error is within an acceptable range, or that one subsystem is similar to another (and therefore has equivalent safety-related properties). *Coverage* is also often a matter of judgment, e.g., that a component has been sufficiently tested, or that sufficiently many flight maneuvers or simulation scenarios of the appropriate kinds have been considered.

Also, correctness of testing scenarios is often a nebulous "know it when you see it" concept.

**Vehicle Flight logs** These have two purposes: logging flight hours to show the absence of mishaps for qualification testing, and carrying out specific flight maneuvers for system identification.

Some logs are recorded on paper, some are electronic. Some are structured (e.g., using sheets from the army), others not. Paper logs are recorded in the Flight Log Book in the EAV Lab.

During flight tests the data-feed from sensors is stored and then used for system identification.

Flight test maneuvers are designed to excite certain modes so they can do system ID. A flight test involves taking an aircraft into a specific flight condition (e.g., a specified altitude, angle of attack, and Mach number), trimming the controls, and then providing a particular input sequence. Common maneuvers include the doublet, 2-1-1, and 3-2-1-1 [46]. These techniques are standard for fixed-wing aircraft

The 2-1-1 is known to generally be better (in the sense of exciting more modes) than the doublet. The EAV team performs a 2-1-1 on the aileron, elevator, and rudder, and a doublet on the throttle. They carry out a few of each, and iterate until they have enough data-points.

Typically there are predetermined manual inputs. They are trying to find the trim condition for a non-linear system.

They assume Gaussian white noise, with zero mean.

**Calibration experiments** The Pitot tube is currently the only sensor calibrated by the EAV team [35], since it is the only one they assembled themselves. They use the same calibration data for the EAV and the Swift. Other components are calibrated by the manufacturers. Those data are reflected directly in the code.

However, determining deflection angles is a kind of calibration too, and the magnetometer is calibrated before each flight.

**Hardware Tests** Static load tests make sure hinges (flange pieces) and actuators can handle loads. There are ground tests (endurance, vibration/thermal) using various configurations and compliance criteria.

The actuators must be sized correctly to make sure they can withstand the loads. This uses both manufacturer data and their own calculations.

**Aircraft models** There are several interrelated aircraft models which are used for deriving parameters (e.g., airfoil stability derivatives such as $C_D$), and for visualizing safety aspects of the avionics.

First, a geometric model of the aircraft is obtained using a robotic arm. This has a certain estimated error. These errors are not propagated forward into the control model (which is determined directly during system ID), but do propagate into the structural and CFD models.

The geometric model is used to derive structural models—CAD diagrams in SolidWorks which document the control surfaces, etc. of the aircraft. These diagrams were sent to MLB to describe the modifications they had to make.

The geometric model is also used to create CFD models, which are used to carry out airfoil analysis using the vortex lattice technique. X-Foil is a 2D airfoil analysis program, which works together with LinAir. The input is a geometric model of the airfoil, and the output consists of parameters like the lift coefficient. The STAR software is used to do Navier-Stokes analysis on the CFD model. The results of the X-Foil and Navier-Stokes analyses are stored in spreadsheets.

The CFD model itself is hard to verify, where "verified" means that it has been experimentally checked in a wind tunnel. However, the LinAir/X-Foil analysis is used to validate the more expensive CFD. Also, simulation verification checks parameters by putting them in a sim to see how the vehicle performs and to get pilot feedback (i.e., Simulation Config. 1). See Appendix C for more details.

Avionics System Diagrams can be used to demonstrate pilot preemption, that is, that we can always go to PIC. This is implemented in hardware (though there is an equivalent in the software). The pilot always has direct control of the MUX A and MUX B channels.

**Manufacturer Data-sheets** Manufacturers of components provide datasheets that give important parameters, such as hinge moments. If the hinge moments are inaccurate, you would exceed the motor limits.

This data is not directly cross-checked but is implicitly validated via ground tests, in particular the static load tests.

(Heald, 1933) is an old paper with data from wind tunnel experiments and an equation that can be used to cross-check with a CFD analysis, utilized to confirm what the hinge moments, $C_{he}$, should be.

**Simulations** The system is tested in the lab using a sequence of configurations, progressively going from full simulation (of hardware and associated software) to the actual system which is flown. In each configuration, the UAS is tested with a set of scenarios — a specific set of inputs to the simulator, e.g., high winds, start at 400ft, then have the engines cut out.

**Config 1: Workstation Simulation Testing** Everything is on the desktop and all components are simulated. The actual controlled software is used though.

**Config 2: Hardware in the Loop Testing** But there is still no UAV hardware here (just the Ground Stations - the "pelican cases" communicate with models)

Communication is by wireless; actuators are simulated. Since there are now 3 computers, they need communication components.

**Config 3: Iron Bird Simulation Testing** Move from desktop to flight computer. We connect to the actual actuators, but the sensors are simulated (simulation components of sensors run on the flight computer).

**Config 4: Flight Testing** No simulation - real sensor in the loop. Same set-up on ground as in the air (so it is considered a "flight test"). Of course, there is only so much you can test without actually flying.

**Software model** Models of sensors, actuators, commands, FMS etc.

**Formal verification framework** We consider assumptions about and inputs to the formal verification to also be external sources of information.

1. justification of the formalization (the model)

2. justification of the verification framework itself (i.e.,AUTOCERT)
3. testing: library functions, domain theory (the axioms)

[6, 7] discuss three levels of justification for the AUTOCERT formal framework.

# 5 Safety Analysis of the Target System

## 5.1 Preliminary Hazard Analysis

To construct the safety case outline for the Swift UAS using our safety assurance methodology (Section 1.4), we begin with hazard analysis.

A variety of techniques exist for hazard identification and analysis. Commonly used techniques include hazards and operability (HAZOP) analysis [49], preliminary hazard analysis (PHA) [58], fault-tree analysis (FTA) [18, 19], event-tree analysis (ETA), failure modes and effects analysis (FMEA) [56], and failure modes effects and criticality analysis (FMECA) [56]. The details of how to utilize each of these in hazard identification and analysis is not in the scope of this document; the interested reader may refer to the references above as well as to the bibliographies contained therein.

In this section, we are mainly concerned with hazard identification and preliminary hazard analysis (PHA) of the target system. This step serves as the foundation for identifying argument structures to assure that certain risks have been managed. In particular, hazard analysis assists in creating fragments of the overall safety argument (discussed in Section 6) that follows from risk management activities.

### 5.1.1 Hazard Identification

In this section, we present a fragment of the ongoing hazard analysis for the Swift UAS. We intend to apply PHA along with FMEA and FTA[26] for hazard identification and analysis. The motivation for choosing these techniques is their systematic process in identifying hazards and in qualifying (or quantifying) their consequences, severity and likelihood (i.e., risk). In practice, PHA is usually applied during the early stages of systems engineering, i.e., during concept definition and design, creating the loop between engineering activities and system safety. On the other hand, FMEA is typically applied when system details are sufficiently well known. This allows reasoning about the causes and consequences of failures in system components, and in identifying mitigatory measures. FTA allows a top-down reasoning from failure hazards to identify failing components in the system that contribute to those hazards.

In addition, PHA forms one step in a chain of successively and continuously refined analyses and yields a detailed evaluation of the safety risks for a given design (or a set thereof). Among the main outcomes from PHA are:

1. Failure modes and relevant hazards.
2. Initiating and *pivot* events in an event chain leading to mishap(s).
3. A basis for risk categorization according to the acceptability of risk.
4. Evidence that there is compliance with the safety regulations and standards.
5. A preliminary set of (system) safety requirements and inputs for design specifications.

Thus, it facilitates the early identification of the appropriate mitigation measures so that system design includes safety aspects.

This is not a comprehensive list of outcomes from PHA, but it represents some key outcomes of interest for the scope of this document. PHA also forms the basis for subsequent and deeper hazard analyses such as requirements hazard analysis (RHA), system and subsystem hazard analysis (SSA & SSHA respectively).

The input to PHA is a preliminary hazard list (PHL), and/ or the outcomes from concept hazard analysis (CHA). The former may be obtained from several sources including, but not limited to, design and concept documentation, known and relevant hazards, and brainstorming by the relevant stakeholders about other potential

[26]For the work presented in this report, we have mainly conducted PHA and FMEA, while FTA has been left for future work.

mishap scenarios. The latter is hazard analysis performed at the concept[27] level and is used to identify major hazards from previous generations of the system or from similar systems.

The *preliminary hazard list* (PHL) used here is based on the risk analysis performed for a previous UAS system generation, namely, the Exploration Aerial Vehicle (EAV) [37]. The main justification for reuse of the list of hazards is the commonality between the two systems, beyond the fact that both are unmanned aerial vehicles. The PHL is shown in Table 1, and includes hazards identified on the aircraft and in the ground system.

Table 1: Preliminary hazard list

| **Subsystem** | **Hazards** |
|---|---|
| Aircraft (UAV) | Cable failure<br>Connector failure<br>Software/ firmware/ Avionics CPU errors<br>Loss of communication<br>Mechanical fastener failure<br>Stuck servo<br>Component failure<br>Overheating of critical flight system |
| Ground System | Avionics ground system failure<br>Ground pilot transceiver failure |

In addition to these, additional hazards have been identified (by the authors) via brainstorming and through examination of the Swift UAS concept documentation. A fragment of these additional hazards are given next.

The additional hazards identified in Table 2 refer primarily to failure hazards in the airborne system. These list of hazards were presented to the subject matter expert[28] (SME) who confirmed their validity beyond those identified in the PHL (Table 1). The next step is to extend this hazard list with hazards from the ground system, the communication infrastructure and the operating environment. This is performed through discussion with the SME and the range safety officer, and facilitates reasoning about completeness of the hazard list.

### 5.1.2 Risk Analysis

Subsequent to hazard identification, we perform hazard risk analysis where the hazards are first individually characterized by their potential consequences, consequence severity, likelihood and acceptability. This is followed by mishap risk analysis, i.e., reasoning about combinations of identified hazards. For example, the hazards listed in Table 2 largely refer to failure conditions in the airborne system which, by themselves, may not necessarily result in a mishap. For example, a faulty sensor presents low mishap risk when the Swift UAS is in the landing phase after completing its mission, whereas corrosion damage in the power system or battery packs presents a mishap risk of its own accord, e.g., having the potential to explode. Consequently, hazards and their combination are to be analyzed in the context of:

1. The UAS operating phases, i.e., Take-off, Climb, Cruise, Survey, Return Cruise, Descent and Landing
2. Assumptions about the operating environment and changes in the operational situation.

Additionally, it is also necessary to reason about hazards that may potentially exist from the intended operation of the system; in particular interactions the intended operation in a potentially unpredictable environment. This requires more careful consideration of the design choices.

A snapshot of the risk analysis is shown in Table 3. It shows a subset of hazards which have been identified for a given operating phase and sub-phase; in this case Descent and Approach respectively. It also illustrates the approach towards risk analysis, i.e., characterizing risk based on the likelihood of hazard occurrence and its

[27] In this context, "concept" refers to the concept of operations (ConOps).
[28] Corey Ippolito, Code TI, NASA Ames.

| NO. | SYSTEM | SUB SYSTEM | COMPONENT / LOCATION | STATE / SITUATION | IS HAZARD? | RATIONALE |
|---|---|---|---|---|---|---|
| **1** | **AIRCRAFT** | | | | | |
| 1.1 | | **Actuation** | | Failure | Yes | Actuation failure may result in an aircraft which cannot be predictably manoeuvred |
| 1.1.1 | | *Control surface actuators* | | Failure | Yes | Control surface actuator failures can result in an unmanoeuvrable aircraft |
| 1.1.1.1 | | | Winglet actuator (L & R) | Failure | Yes | |
| 1.1.1.2 | | | Elevon actuator (L & R) | Failure | Yes | Failure of elevon actuator results in failure to control elevators and ailerons |
| 1.1.1.3 | | | Flap actuator (L & R) | Failure | Yes | Failure of flap actuator results in failure to control flaps |
| 1.1.2 | | *Steering actuators* | | Failure | Yes | Streering actuator failure results in an aircraft that cannot be steered on the ground introducing a potential for runway incursion / excursion |
| 1.1.2.1 | | | Front wheel steering actuator | Failure | Yes | |
| | | | | | | |
| 1.2 | | **Propulsion** | | Failure | Yes | Propulsion failure results in loss of lift and/ or thrust |
| 1.2.1 | | *Electric motor system* | | Failure | Yes | |
| 1.2.1.1 | | | Motor controller | Failure | Yes | |
| 1.2.1.2 | | | DC motor | Failure | Yes | |
| | | | | | | |
| 1.3 | | **Avionics** | | Failure | Yes | Avionics failure results in loss of control |
| 1.3.1 | | *Avionics hardware* | | Failure | Yes | Avionics hardware failure may result in loss of control |
| 1.3.1.1 | | | Flight sensors | Failure | Yes | Sensor failures will result in incorrect computation of control parameters |
| 1.3.1.1.1 | | | IMU/GPS (Rockwell Collins Athena 111m) | Failure | Yes | |
| 1.3.1.1.2 | | | DGPS (Novatel OEM4-G2) | Failure | Yes | |
| 1.3.1.1.3 | | | Air data probe | Failure | Yes | |
| 1.3.1.1.4 | | | GPS antenna | Failure | Yes | |
| 1.3.1.1.5 | | | DGPS antenna | Failure | Yes | |

Table 2: Fragment of additional hazard list

Table 3: Preliminary hazard analysis fragment

| ID | HAZARD / SCENARIO DESCRIPTION | POTENTIAL CAUSES | EFFECT ON SYSTEM / CONSEQUENCES | LIKELIHOOD | SEVERITY | RISK CATEGORY | MITIGATION MEASURES | CORRECTIVE ACTION | SAFETY REQUIREMENT |
|---|---|---|---|---|---|---|---|---|---|
| **OPERATING PHASE** | Descent (DE) | | | | | | | | |
| **SUB-PHASE** | Approach (APP) | | | | | | | | |
| | **ACTUATION** | | | | | | | | |
| PHA_DE.APP_ACT_001 | Winglet actuator failure | (1) Incorrect installation<br>(2) Interference<br>(3) Insufficient maintenance | (1) Loss of control of flight surface controlled by winglet actuator.<br>(2) Aircraft stall | Remote | Hazardous | 2B | Preflight inspection | Correct installation | [R12] Actuators must not interfere or collide with existing structure<br>... |
| PHA_DE.APP_ACT_002 | Elevon actuator failure | (1) Incorrect installation<br>(2) Interference<br>(3) Insufficient maintenance | (1) Loss of control of flight surface controlled by Elevon actuator (Elevator + Aileron)<br>(2) Aircraft stall. | Remote | Hazardous | 2B | Preflight inspection | Correct installation | [R12] Actuators must not interfere or collide with existing structure<br>... |
| PHA_DE.APP_ACT_003 | Flap actuator failure | (1) Incorrect installation<br>(2) Interference<br>(3) Insufficient maintenance | (1) Loss of control of flaps<br>(2) Aircraft stall | Remote | Hazardous | 2B | Preflight inspection | Correct installation | [R12] Actuators must not interfere or collide with existing structure<br>... |
| PHA_DE.APP_ACT_004 | Front wheel steering actuator failure | (1) Incorrect installation<br>(2) Interference<br>(3) Insufficient maintenance | No known consequence during approach sub-phase | Remote | Hazardous | 2B | Preflight inspection | Correct installation | [R12] Actuators must not interfere or collide with existing structure<br>... |
| | **AVIONICS HARDWARE: SENSORS** | | | | | | | | |
| PHA_DE.APP_AVCS_001 | IMU/GPS (Rockwell Collins Athenia 111m) Failure | | (1) Loss of GPS signal<br>(2) Incorrect waypoint and heading data supplied to autopilot<br>(3) Drift outside range safety area | Extremely Remote | Major | 3C | | | |
| PHA_DE.APP_AVCS_002 | DGPS (Novatel OEM4-G2) Failure | | | Extremely Remote | Major | 3C | | | |
| PHA_DE.APP_AVCS_003 | Air data probe Failure | (1) Incorrect installation | (1) Incorrect airdata supplied to autopilot<br>(2) Overspeed<br>(3) Aircraft stall<br>(4) Loss of flight | Remote | Hazardous | 2B | (1) Failsafe autopilot forces controlled spiral to ground | | [FSP_AVCS_002] It is always the case that whenever airdata probe failure is detected, failsafe autopilot eventually forces a controlled spiral to ground |
| PHA_DE.APP_AVCS_004 | GPS antenna Failure | (1) Incorrect installation | | Probable | Major | 3A | (2) Ground station pilot controller overrides autopilot | | |
| PHA_DE.APP_AVCS_005 | DGPS antenna Failure | (1) Incorrect installation | (1) Loss of GPS signal<br>(2) Incorrect waypoint and heading data supplied to autopilot<br>(3) Drift outside range safety area | Probable | Major | 3A | | | |
| PHA_DE.APP_AVCS_006 | 900MHz Omni antenna Failure | (1) Incorrect installation | (1) Loss of communication with ground station<br>(2) Drift outside range safety area<br>(3) Loss of mission | Probable | Major | 3A | Failsafe autopilot forces controlled descent | | [FSP_AVCS_001] It is always the case that whenever loss of communication with ground is detected,failsafe autopilot eventually takes control within a specified time duration. |
| | **AVIONICS SOFTWARE: AUTOPILOT** | | | | | | | | |
| PHA_DE.APP_AVCS_012 | Flight management system (FMS) Failure | | | Remote | Major | 3B | | Verify that specification is consistent with theory | [FSP_AVCS_004] When FMS failure is detected, it is always the case that failsafe autopilot eventually takes control within a specified time duration |
| PHA_DE.APP_AVCS_013 | AP Failure | 1. Deadlocks<br>2. Timing errors<br>3. Memory corruption<br>4. Incorrect specification<br>5. Incorrect implementation<br>6. Inaccurate / incorrect assumptions<br>7. Wrong interpretation of theory | (1) Incorrect computation of control surface signals<br>(2) Incorrect actuator signals supplied<br>(3) Loss of control over flight surface<br>(4) Loss of mission<br>(5) Loss of flight<br>(6) Incorrect computation of waypoints and headings | Probable | Major | 3A | (1) Ground station pilot controller overrides autopilot | Verify correct autopilot implementation | [FSP_AVCS_003] When AP failure is detected, it is always the case that failsafe autopilot eventually takes control within a specified time duration |
| PHA_DE.APP_AVCS_014 | Waypoint data Failure | | | Remote | Hazardous | 2B | (2) Failsafe autopilot intervenes when failure of autopilot detected | (1) Verify legality of issued commands<br>(2) Guarantee correct interpretation of commands | [A1] Commands must be interpreted correctly<br>[A2] No command shall make the autopilot execute an unsafe maneuver. |
| PHA_DE.APP_AVCS_015 | Autopilot module | | | Remote | Hazardous | 2B | | (1) Verify legality of issued commands<br>(2) Guarantee correct interpretation of commands | [A1] Commands must be interpreted correctly<br>[A2] No command shall make the autopilot execute an unsafe maneuver. |

severity. The risk analysis relies on risk categories (Figure 14), which effectively combine likelihood and severity to give a qualitative gauge of risk. On the other hand, if the likelihood and severity can be precisely quantified (such as using exact probabilities or distributions, or cost of the loss event), it is possible to perform *quantitative* risk analysis (as is done during PRA).

| | **SEVERITY** | CATASTROPHIC | HAZARDOUS | MAJOR | MINOR | NO SAFETY EFFECT |
|---|---|---|---|---|---|---|
| **Likelihood** | Index | *1* | *2* | *3* | *4* | *5* |
| PROBABLE | *A* | 1A | 2A | 3A | 4A | 5A |
| REMOTE | *B* | 1B | 2B | 3B | 4B | 5B |
| EXTREMELY REMOTE | *C* | 1C | 2C | 3C | 4C | 5C |
| EXTREMELY IMPROBABLE | *D* | 1D | 2D | 3D | 4D | 5D |

Figure 14: Risk categories reflecting a combination of the likelihood of hazard occurrence and severity [58]. Other categorizations also exist based on the safety guideline/standard used, e.g., a $5\times5$ table with five categories of likelihood of occurrence and different severity categories, as in [21].

The risk category column provides a basis for decision making, e.g., hazards with a risk category 1A, 1B, 1C, 2A, 2B and 3A may represent risks which are unacceptable, and therefore require specific measures to reduce it to acceptable levels. On the other hand a hazard with a risk category 3D may be considered as acceptable, and therefore may not be dealt with, in the same way as those in the previous categories. We revisit these notions in Section 6 when we consider how risk analysis will drive the creation of the safety case outline of the Swift UAS. As seen in table 3, the analysis facilitates the identification of functional safety requirements, which when correctly implemented will eliminate or mitigate the identified hazards. For example, the failure hazard of the `AP` submodule of the Swift UAS autopilot (PHA_DE.APP_AVCS_013), is adjudged to present unacceptable risk (category 3A). To mitigate this risk, a functional safety requirement is created which states:

**[FSP_AVCS_003]**: When AP failure is detected, the failsafe autopilot shall always eventually take control within a specified time duration.

The failsafe autopilot represents a mechanism for contingency management or flight termination and forms part of the Swift UAS contingency management system (CMS).

This analysis is ongoing therefore table 3 is partially filled. The gaps in the table reflect lack of domain knowledge on the part of the authors and subsequent work includes closing these gaps in discussion with the designer and range safety officer of the Swift UAS. Additionally, the columns Likelihood and Severity were filled by the authors and do not reflect the actual risk of these hazards in the Swift UAS. Indeed, subsequent work also involves populating the columns with values which are representative of the knowledge of the domain experts, and the available data.

Note also that in Table 3, only a small subset of software hazards and their corresponding causes have been listed. Elsewhere[29] latent software defects have been identified as potential additional causes for autopilot software failure hazards. As future work, we anticipate updating the hazard analysis to include these additional causes, the resulting mitigation measures and the corresponding safety requirements. Some of these safety requirements pertain to the existing practices of process-based software safety assurance activities (Appendix E).

Hazard identification also provides input for identifying failure modes. This can subsequently be used in an FMEA. As an example, a fragment of the identified failure modes is shown in Table 4.

The hazard analysis illustrated in this section provides the basis for the safety case outline of the Swift UAS. Additionally, it provides the rationale for deriving functional safety requirements. Indeed, the generic safety case outline for the Swift UAS, iteratively argues that hazards which present the potential for mishaps are eliminated or

[29] In communication with FAA DER, Joe Wlad. See also Appendix E.

Table 4: Failure modes and effects analysis (FMEA) fragment

| HAZARD / SCENARIO DESCRIPTION | HAZARD CATEGORY | LINK TO RELATED SCENARIOS | ID | COMPONENT | FAILURE MODE |
|---|---|---|---|---|---|
| Actuation failure | Mechanical | PHA_ACT_001 | FMEA_ACT_001 | Winglet actuator (L&R) | Stuck |
| | | | FMEA_ACT_002 | Winglet actuator (L&R) | Free |
| | | | FMEA_ACT_003 | Winglet actuator (L&R) | Damaged (partially free) |
| | | | FMEA_ACT_004 | Elevon actuator (L&R) | Stuck |
| | | | FMEA_ACT_005 | Elevon actuator (L&R) | Free |
| | | | FMEA_ACT_006 | Elevon actuator (L&R) | Damaged (partially free) |
| | | | FMEA_ACT_007 | Flap actuator (L&R) | Stuck |
| | | | FMEA_ACT_008 | Flap actuator (L&R) | Free |
| | | | FMEA_ACT_009 | Flap actuator (L&R) | Damaged (partially free) |
| | | | FMEA_ACT_010 | Front wheel steering actuator | Stuck |
| | | | FMEA_ACT_011 | Front wheel steering actuator | Free |
| | | | FMEA_ACT_012 | Front wheel steering actuator | Damaged (partially free) |
| Populsion failure | Overheating | PHA_PRP_001 | FMEA_PRP_001 | Motor controller | Overheating |
| | | | FMEA_PRP_002 | DC Motor | Overheating |
| Propulsion failure | Mechanical | PHA_PRP_002 | FMEA_PRP_003 | Motor controller | Stress fracture |
| | | | FMEA_PRP_004 | Motor controller | Wear out due to friction |
| | | | FMEA_PRP_005 | DC motor | Stress fracture |
| | | | FMEA_PRP_006 | DC Motor | Wear out due to friction |
| Propulsion failure | Electrical | PHA_PRP_003 | FMEA_PRP_007 | Motor controller | Open circuit |
| | | | FMEA_PRP_008 | Motor controller | Short circuit |
| | | | FMEA_PRP_009 | Motor controller | Overvoltage |
| | | | FMEA_PRP_010 | DC Motor | Physical loss of connector |
| | | | FMEA_PRP_011 | DC Motor | Open circuit |
| | | | FMEA_PRP_012 | DC Motor | Short circuit |
| | | | FMEA_PRP_013 | DC Motor | Overvoltage |
| | | | FMEA_PRP_014 | DC Motor | Physical loss of connector |
| Avionics failure | Hardware (including Electrical) | PHA_AVCS_001 | FMEA_AVCS_001 | IMU/ GPS (Rockwell Collins Athena 111m) | Accumulation error (Abbe error) |
| | | | FMEA_AVCS_002 | IMU/ GPS (Rockwell Collins Athena 111m) | Accuracy error |
| | | | FMEA_AVCS_003 | IMU/ GPS (Rockwell Collins Athena 111m) | Non acquisition of satellites |
| | | | FMEA_AVCS_004 | DGPS (Novatel OEM4-G2) | Accuracy error |
| | | | FMEA_AVCS_005 | DGPS (Novatel OEM4-G2) | Non acquisition of satellites |
| | | | FMEA_AVCS_010 | I/O Board - Parvus COM-1274 | Soldering problems |
| | | | FMEA_AVCS_011 | I/O Board - Parvus COM-1275 | Overheating |
| | | | FMEA_AVCS_012 | I/O Board - Parvus COM-1276 | Physical loss of connector |
| | | | FMEA_AVCS_013 | Flight computer - ADL945PC-L2400 | Soldering problems |
| | | | FMEA_AVCS_014 | Flight computer - ADL945PC-L2401 | Overheating |
| | | | FMEA_AVCS_015 | Flight computer - ADL945PC-L2402 | Physical loss of connector |
| Avionics failure | Software | PHA_AVCS_002 | FMEA_AVCS_020 | Flight management system (FMS) | Incorrect transition |
| | | | FMEA_AVCS_021 | Flight management system (FMS) | Incorrect updates |
| | | | FMEA_AVCS_022 | Flight management system (FMS) | Wrong response to command |
| | | | FMEA_AVCS_023 | Autopilot (AP) | Incorrect transition |
| | | | FMEA_AVCS_024 | Autopilot (AP) | Incorrect updates |
| | | | FMEA_AVCS_025 | Waypoint data | Incorrect data |
| | | | FMEA_AVCS_026 | gs11m | |
| | | | FMEA_AVCS_027 | javmodem | |
| | | | FMEA_AVCS_028 | ap_rcap | |
| | | | FMEA_AVCS_029 | datastore | |
| | | | FMEA_AVCS_030 | ap_failsafe | Non detection of communication loss |
| | | | FMEA_AVCS_031 | pcsfilter | |
| | | | FMEA_AVCS_032 | sv203interface | |
| | | | FMEA_AVCS_033 | Reflection virtual machine | |
| | | | FMEA_AVCS_034 | CGL | |
| | | | FMEA_AVCS_035 | scripts (rfs and rff files) | |
| Flight critical system | | PHA_FCRS_001 | FMEA_FCRS_001 | Pilot receiver (Duplicated) | |
| | | | FMEA_FCRS_002 | Motor controller (Duplicated) | |
| | | | FMEA_FCRS_003 | Multiplexers | Stuck at 1 |

mitigated, given a specified context. Effectively, the safety case outline describes the framework of the argument structure which will be used in assuring the safety of the target system. It also links the justifying evidence (when it exists) to the safety requirements derived from hazard elimination or mitigation measures. This outline is discussed in greater detail in Section 6.

## 5.2 Safety Requirements

We consider the following high-level safety requirements, which have been derived from the hazard analysis described in section 5.1:

**A1** *The autopilot shall interpret all commands correctly.*

The safety significance here, is that it is important to take care to avoid going into areas where one should not go (i.e., outside the range) by avoiding overshoot when tracking to a waypoint.

**A2** *The autopilot shall execute safe maneuvers for all commands.*

This is ensured by placing limits on the output of each block in the autopilot. Speed is limited between Vmin and Vmax, but alpha and beta are not directly limited. Instead, (alpha, beta, $V$) can be converted to Euler angles and limits are placed on those. However, even if the Eulers appear to be in range, because of the wind, the angle of attack might be inaccurate. Therefore, since we are not sure about the wind, conservative limits are placed on the Euler angles – pitch $\theta_{min}$ and $\theta_{max}$; likewise for the roll (but not the yaw).

There are several design features relating to the safe execution of commands by the autopilot. For example, "throttle control altitude is default, for safety if the engine fails" [34]. The rationale for this is that there are two ways to control altitude — elevators for the angle of attack (pitch angle), or throttle for the altitude. If you use the angle of attack you will eventually stall during engine out.

Another example is that during immediate attitude capture "the aircraft is controlled within the safety limits of the controller". These safety limits are set to "conservative" values, using their expertise. This conservativeness is reasonable since it is a standard aircraft.

**A3** *The autopilot shall maintain accurate state information.*

If inaccurate information is transmitted to the GSO, then the pilot will not be able to safely control the UAV, nor will the autopilot be able to ensure that safe limits are maintained.

We choose A2 as the initial Base Year target safety requirement. Based on Figure 15 this can be refined to this lower level functional safety requirement (among others)

- The autopilot shall correctly compute the actuator outputs

which can then be further refined to the following low-level Base Year Target functional safety requirements:

**LL-SR-001** The autopilot module shall correctly compute the aileron control variable.

**LL-SR-002** The autopilot module shall correctly compute the elevator control variable.

The previous section describes the computations which are used to compute the aileron control variable. These are properties of the code, and can be seen as functional safety requirements, though we do not explicitly label them as such at this low level.

- The autopilot module shall correctly compute the cross-track error.
- The autopilot module shall correctly compute the desired heading.

As time permits, we will verify other aspects of the autopilot safety and functional safety.

The Swift UAS hazard analysis (Section 5.1) provides the justification for these safety requirements (among the complete set of safety requirements and functional safety requirements). For example, requirements A1 and A2, are, in part, derived from the mitigation measures for failure hazards PHA_DE.APP_AVCS_014 and PHA_DE.APP_AVCS_015 of the waypoint data and the autopilot modules respectively, during the descent phase (Table 3).

Safety properties of the software (which could be expressed ass requirements) include (but are not limited to) the following *language-specific* properties:

- absence from run-time errors (such as division by 0)

and *domain-specific* properties:

- correct use of units
- variables representing physical quantities (e.g., bank angle) remain within the appropriate bounds

Note that the UAS has a particular configuration, and a particular set of modules, instantiated via a script, executing a particular flight plan. Safety is specific to each of these choices. However, the general properties outlined above are largely independent of these variabilities.

Requirement A1 is essentially functional correctness of autopilot, a high-level property of the flight software. Roughly speaking, the correctness of the autopilot module is dependent on showing the correctness of the underlying subsystems. To show the correctness of the autopilot and the function calls instigated there we must show the correctness of the FMS and AP subsystems which the autopilot depends upon and initializes. The FMS subsystem interprets the list of commands given to an aircraft. The transition between these commands must be correctly executed, which implies that new modes will be set in the FMS and AP systems. The AP system must then correctly compute the output for each aircraft control surface based on which modes have been set. The correctness of the autopilot is dependent on each subsystem properly communicating any state transitions and then operating appropriately on that resulting state. At a high level the correctness of the autopilot depends on the following conditions:

1. The Autopilot module and its dependent objects are correctly initialized and the module makes its update calls properly.
2. The FMS system is correctly initialized. For each command the transitions are correctly implemented and the resulting modes that are set are correct for that command.
3. The AP system is correctly initialized and correctly interprets the modes set in the FMS. The output of surface control calculations is correct for the current set of modes.

Figure 15 lists the high level hierarchy of functional safety requirements for the autopilot module. This decomposition is based on the structure of the code defining the autopilot module and its subsystems.

The autopilot is correct if

- The system is properly initialized
  - aircraft state information is properly received from the sensors
  - the current, previous, and next waypoints are properly defined
  - the FMS object is properly initialized
  - the AP object is properly initialized
  - Input and Output variables are properly routed via the reflection systems scripts
  - Parameter data is properly initialized.
- The FMS system is correct if
  - The FMS system and AP system are properly initialized by the autopilot
  - The FMS system properly interprets all commands
  - The FMS system properly transitions between commands and waypoints
  - UpdateCommandTransitions is correct
    - The FMS system properly transitions into a COMMAND_LAND command
      - The FMS system properly transitions between phases when the command is COMMAND_LAND
      - the transition criteria for each phase is properly interpreted
      - the FMS system properly transitions between modes representing the different phases of the landing command
      - the FMS system properly sets the new waypoint for approach
    - The FMS system properly transitions into and executes all other commands
  - The FMS system properly sets FMS Lateral and Longitudinal modes for the corresponding command
    - setNewFMSWaypoint is correct
  - The FMS system properly sets AP Lateral and Longitudinal modes
    - SetFMSLatMode is correct
    - SetFMSLonMode is correct
  - The FMS system properly updates state variables in the AP system
    - UpdateFMSLatMode is correct
    - UpdateFMSLonMode is correct
- The AP system creates correct output for all aircraft control surfaces if
  - The autopilot correctly initializes the AP object
  - The FMS system correctly updates the AP modes and state variables.
  - Each PID loop
    - receives correct aircraft state information
    - receives correct current and previous waypoint information
  - PID controller objects are properly initialized
  - PID controller updates are correct for each aircraft controller surface
  - The CGL mathematics libraries provide correct results
  - The autopilot shall correctly compute the actuator outputs

Figure 15: Breakdown of autopilot functionality

# 6 System Safety Case Outline

We now present several fragments of the intended overall safety argument for the Swift UAS. We have constructed the safety case fragment using AdvoCATE, our assurance case automation toolset [15]. The fragments together represent an end-to-end "slice" of the argument (shown as a bird's eye view in Figure 16) across the overall safety case, starting from a top-level safety goal down to the evidence and/or proof obligations for a low level computation for a specific flight control surface. Broadly, this slice can be thought of as comprising a *manually created* fragment (shown enclosed in a dotted box in Figure 16) and an *automatically* generated fragment.

In the overall safety argument fragments to be shown subsequently, information for specific context elements and justification elements are mined from the information provided by the SME, specifically from [32–34,36,37].

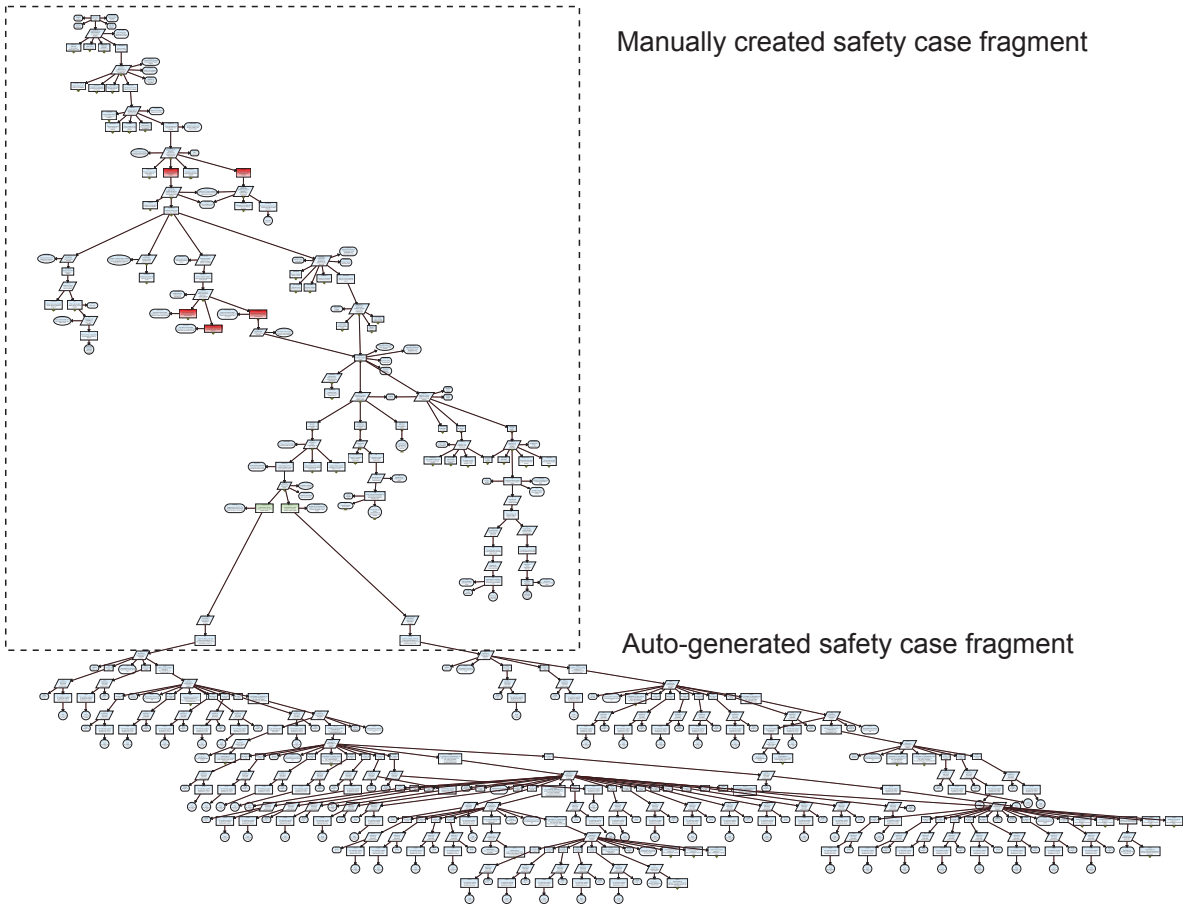



Figure 16: Bird's eye view of an end-to-end slice of the overall system safety case for the Swift UAS.

## 6.1 Manually Created Safety Case Fragment

The manually created safety case mainly reflects the part of the (slice of the) overall safety case which is relevant for the airborne system in the Swift UAS, and it is based on the preliminary hazard and risk analysis (Section 5).

The argument is created in *layers*, i.e., starting with a top-level safety claim, we identify sub-claims linking the top-level system safety claim to software. Subsequently, claims about the software are broken down into claims about the autopilot; then into claims about the autopilot controller (`AP`) module, and eventually into

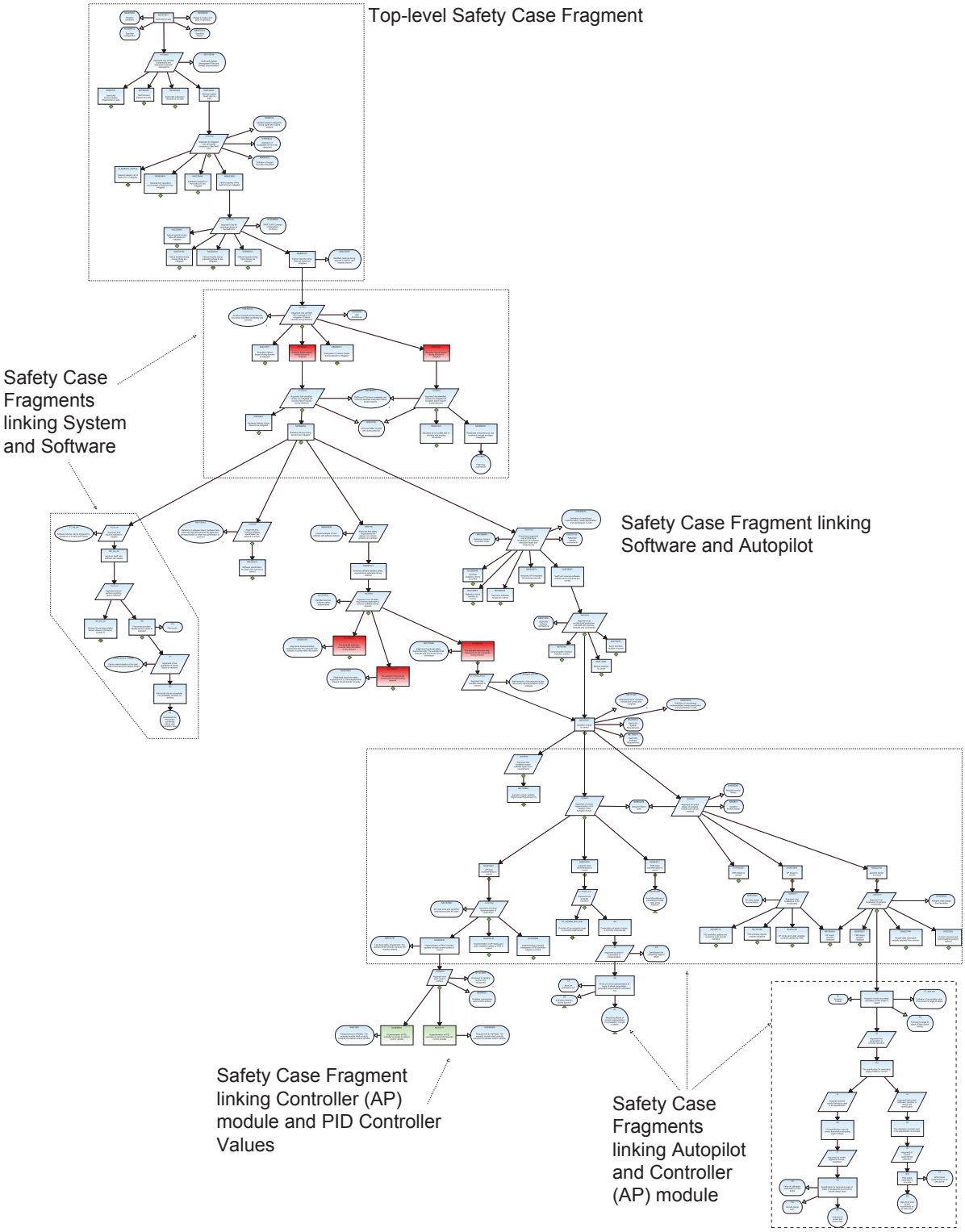



Figure 17: Bird's eye view of the manually created safety case fragment for the Swift UAS.

claims about the PID controllers which provide the appropriate control signals to the relevant flight control surfaces. This is shown in Figure 17 as a bird's eye view.

As shown in the figure, the argument fragment supporting the claims at the last level are automatically generated using AUTOCERT, and this is described in greater detail in Section 6.2. We now briefly describe each layer of this safety case fragment.

### 6.1.1 Top-level Safety Case Fragment

The case is made for a safe system by creating an argument structure that justifies that mishap risk has been reduced to acceptable levels, by either eliminating the identified hazards, or by mitigating them. Recalling the risk categories shown in Figure 14, the hazards which need to be considered in the safety case are those which are categorized as having unacceptable risk. By definition, when mishap risk has been reduced to acceptable levels, the system is safe. A key challenge is providing sufficient confidence that *all* the hazards which contribute to mishaps have been identified and addressed. We begin the safety argument by claiming that the Swift UAS is safe in the context of a given site of operation, for certain weather conditions, for a specified mission and in a specific configuration (Figure 18).

In Figure 18, one strategy for arguing that the failure hazards of the airborne system (the UAV) have been eliminated or mitigated, could be first argument over all hazards, across all operating phases (Strategy ID N70618522). An alternative approach is first argument over all subsystems (Strategy ID N18584532) instead. One reason to use the former is to initially address those failure hazards which can change risk categories depending on the mission phase. For example, failure of the nose wheel actuator does not pose mishap risk during the cruise phase, whereas it does, during the flare sub-phase of the descent phase. On the other hand, using the latter strategy, i.e., arguing over all subsystems, facilitates the creation of a safety argument which is easier to maintain and can be better modularized.

Note that regardless of the strategy that is used first, arguing for the mitigation of hazards that may change risk categories will be still required for ensuring that the safety case is complete. We hypothesize, however, that depending on the strategy used, the structure of the safety case will change. Testing this hypothesis is not part of the scope of this work and is left for future work.

### 6.1.2 Linking the System and Software

Figure 19 shows the safety case fragment where the claim that the UAV failure hazards during descent are mitigated (Goal ID N20891613) is decomposed to the claim that avionics failure hazards during descent are mitigated (Goal ID N2763522). In turn, this goal is refined to the claim that software failures (in the avionics system) are mitigated during descent (Goal ID N20268452). This claim is further developed in the safety case fragment shown in Figure 20.

Figure 19 also illustrates how two sources of non-formal information have been incorporated: the hazard analysis from the FMEA (e.g., as shown in Figure 4) as the context for the argument over all identified causes (Strategy ID N16380072), and the flight-day procedures (Evidence ID N1518857) as evidence of the measures employed to mitigate actuator failures resulting from maintenance issues.

### 6.1.3 Linking the Software and the Autopilot Module

As mentioned in Section 6.1.2, Figure 20 shows the safety case fragment that develops the claim of mitigating (avionics) software failures during descent (Goal ID N20268452). This claim has been developed into the claims that (1) the avionics software modules and commands are correct (Goal ID N35108042) and (2) subsequently that the autopilot module is correct (Goal ID N82535547) by using the strategy of making correctness arguments (Strategy ID N93041050). Note that arguing correctness is not always required when making a safety claim.

However, in our case the correctness of the avionics software is itself safety related. In other words, incorrect behavior is unsafe behavior and this context is made explicit in the argument fragment (Context ID N76400477), i.e., in the definition of correctness of the software components. In the particular case of the failsafe autopilot (which is part of the avionics software and also forms a part of the Swift UAS CMS), its correct behavior is required to assure safety.

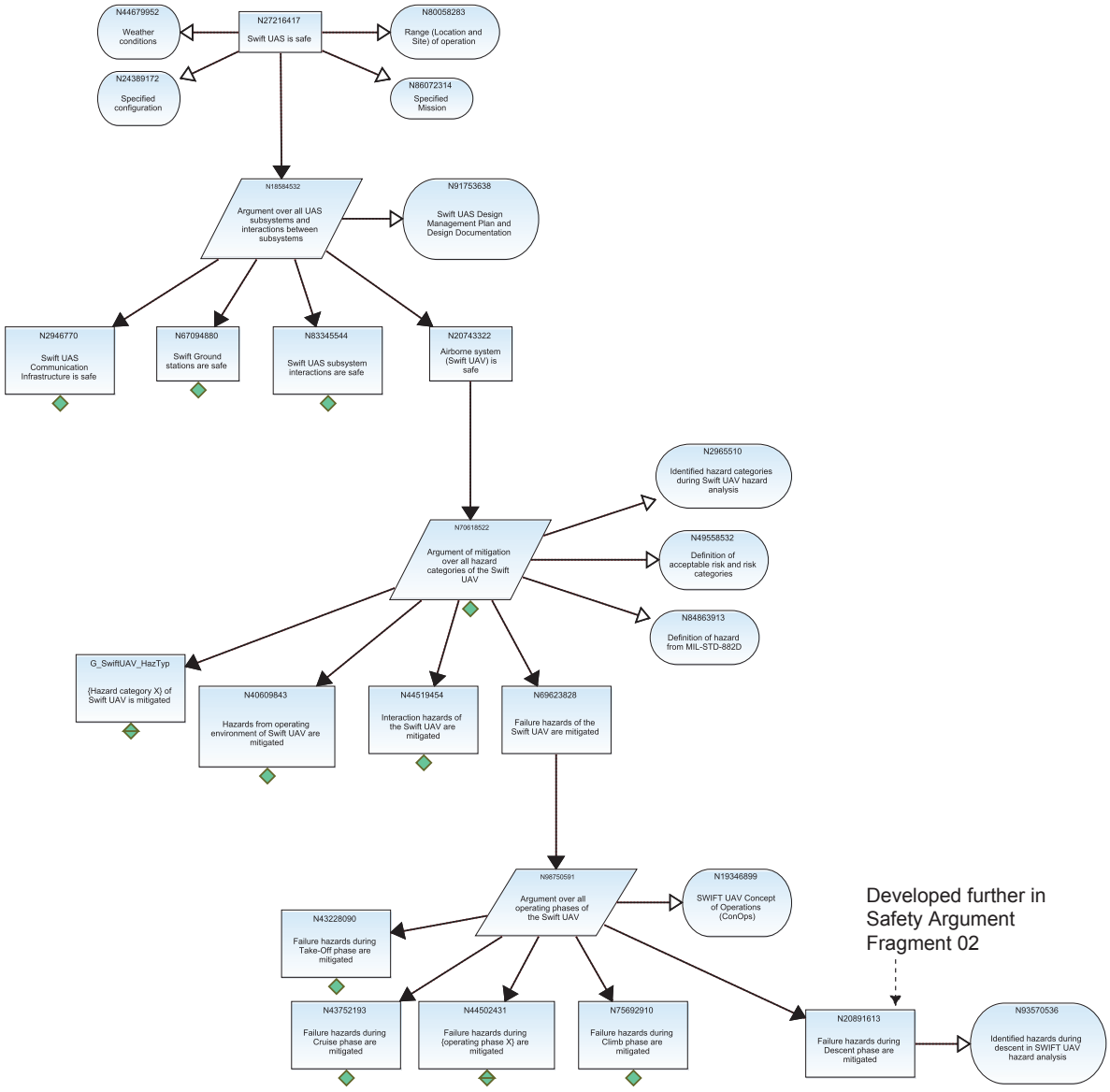



Figure 18: Safety argument fragment 01. Top level safety argument for the Swift UAS.

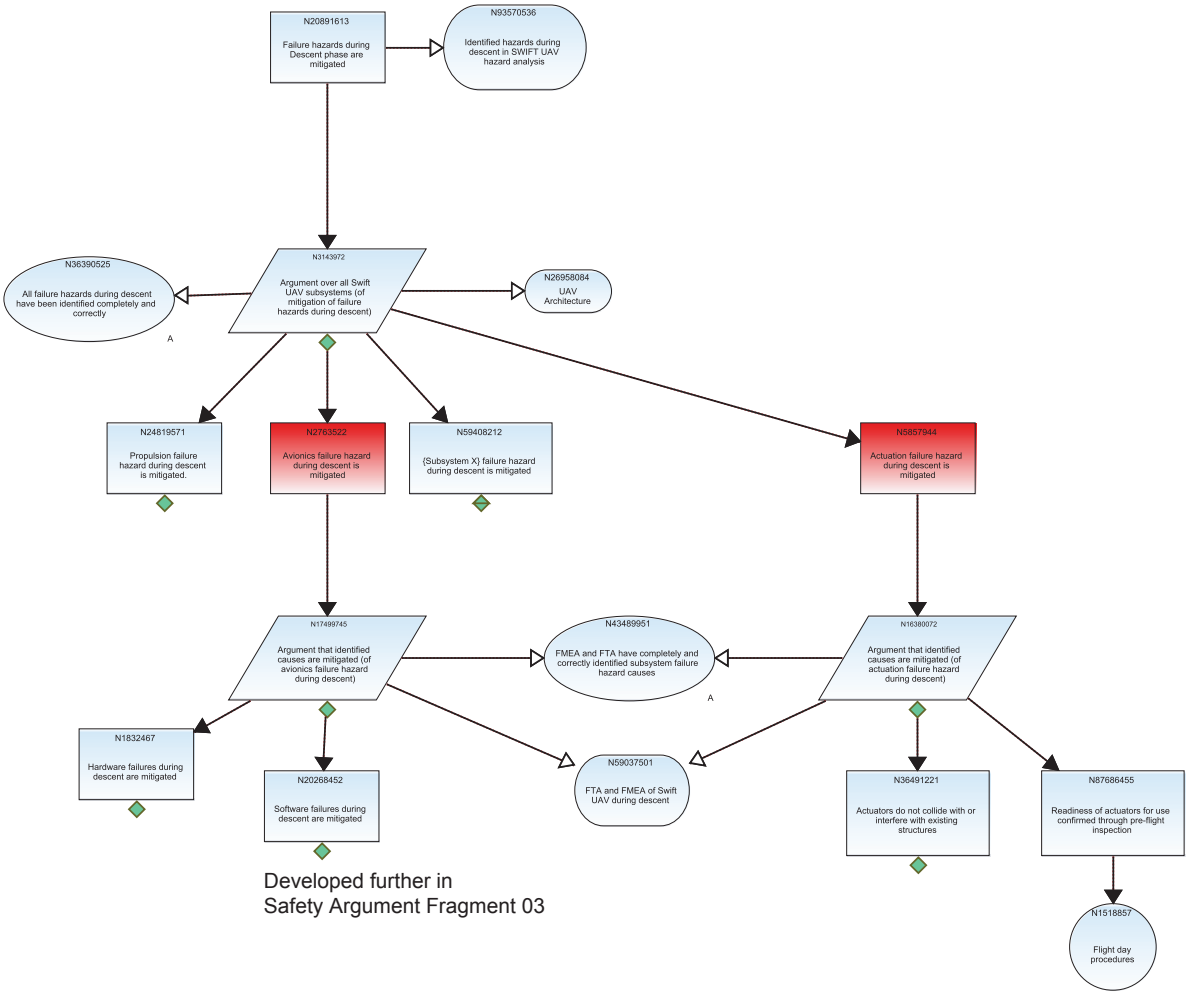



Figure 19: Safety argument fragment 02. Entry point to the software safety argument for the Swift UAS from the top-level safety argument.

Figure 20 also shows how the claim of mitigation of avionics software failures during descent (Goal ID N20268452) is also linked to the identified high-level functional safety requirements[30] **A1**, **A2** and **A3** as identified in Section 5.2. The latter are themselves reflected as specific goals:

(i) N14968801: The autopilot interprets all commands correctly during descent.

(ii) N19965398: The autopilot executes safe maneuvers for all commands during descent.

(iii) N11024219: The autopilot maintains accurate state information during descent.

These arise in the safety argument, in part, from the strategy that software failures can be mitigated by satisfaction of the safety requirements on avionics software during descent (Strategy ID N36621902). Finally, we also argue that software failures during descent are mitigated when the expected input the software is reliably provided (Strategy ID S1_Inp_rel, and Goal ID G2_inp_rel).

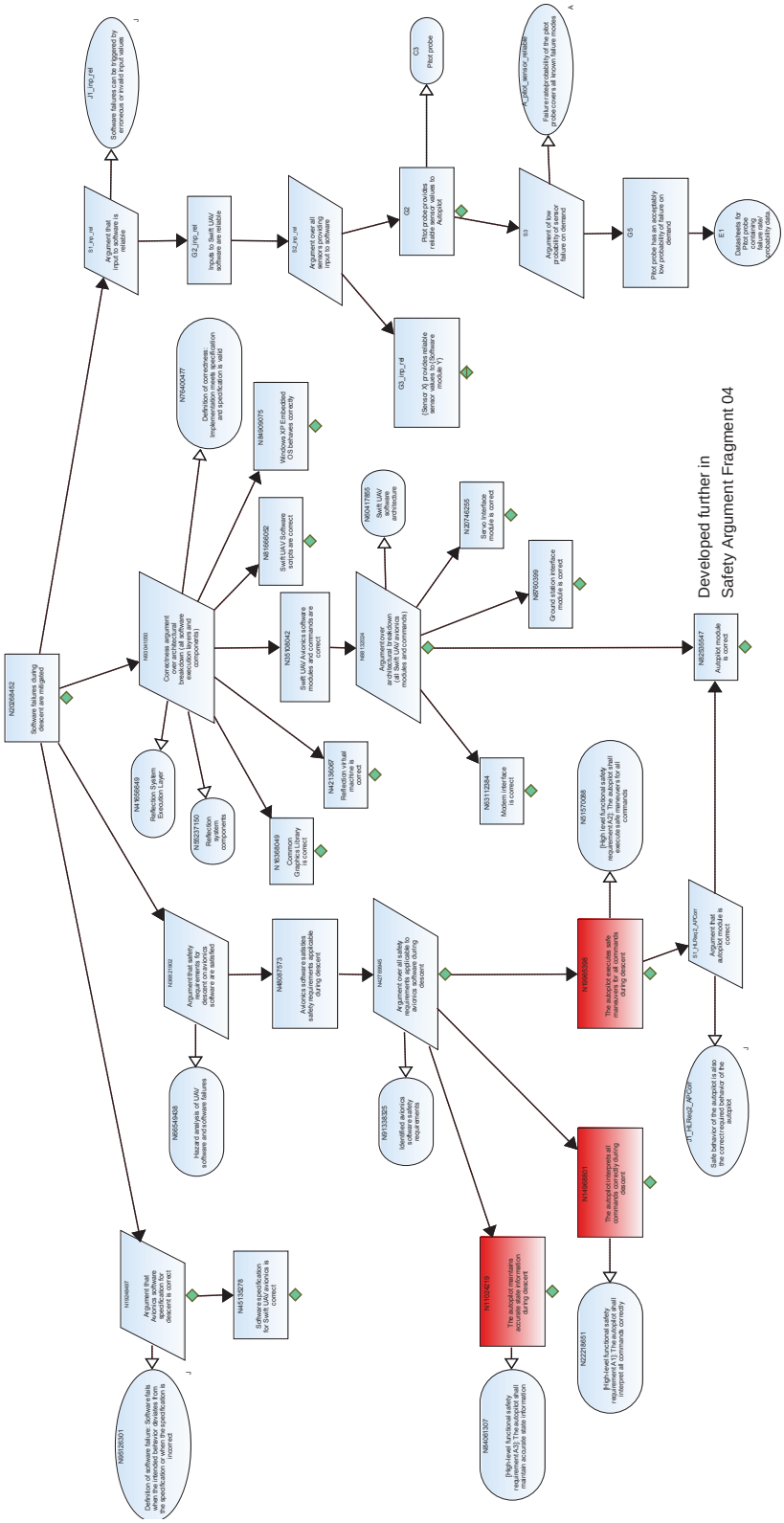



Figure 20: Safety argument fragment 03. Argument linking safety claims about the avionics software and the autopilot.

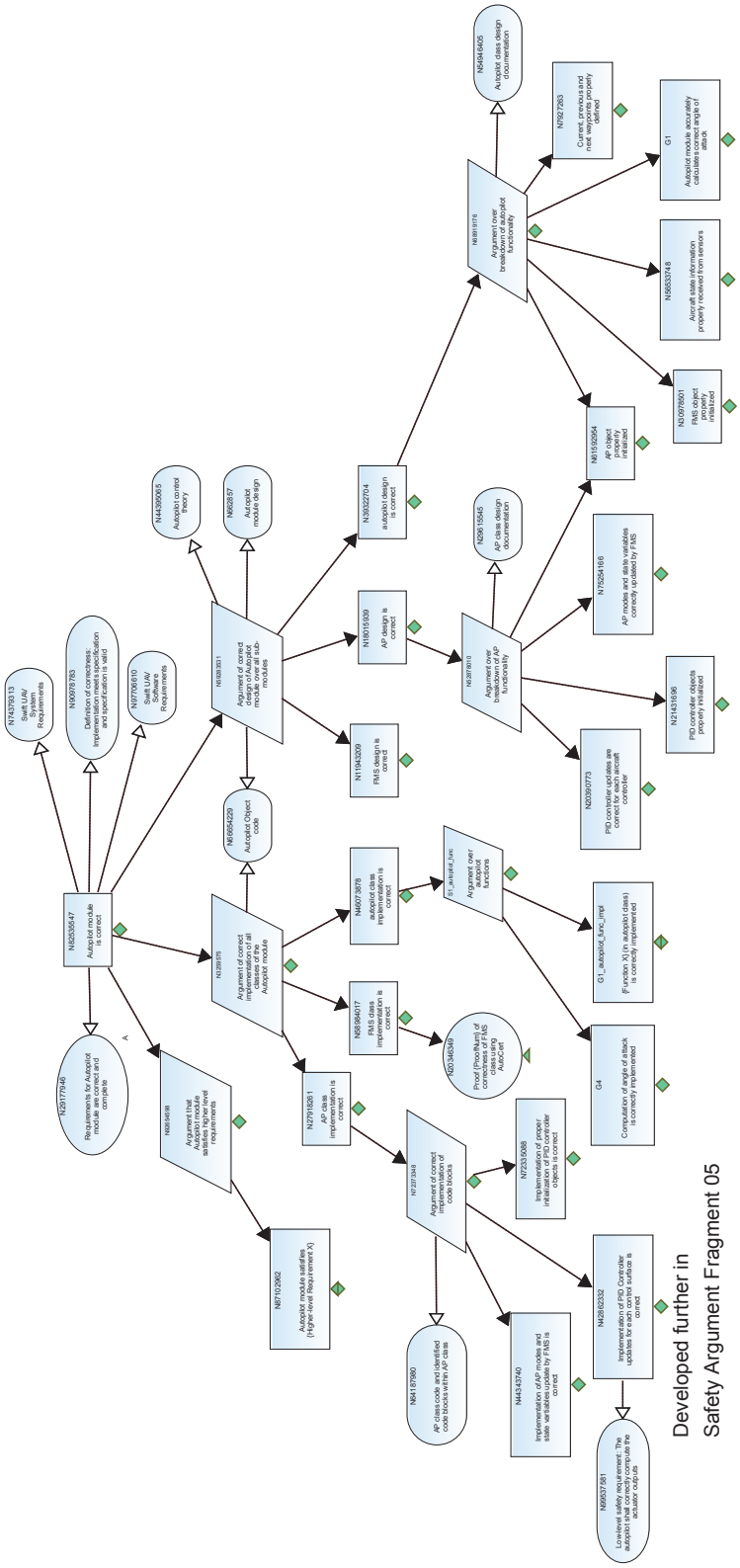



Figure 21: Safety argument fragment 04. Argument linking safety claim for the Swift UAS Autopilot to the `AP` submodule.

### 6.1.4 Linking the Autopilot and the Controller Module

Figure 21 shows that the claim that the autopilot module of the avionics software is correct (Goal ID N82535547) is undeveloped, i.e., it may not be sufficient to use the strategies of arguing correctness of the design (Strategy ID N59283531) and correctness of the implementation (Strategy ID N3259575) to claim the correctness of the autopilot. Thus, we reflect the need to show that the autopilot meets the higher-level requirements on the Swift UAS itself using an undeveloped and uninstantiated goal (Goal ID N87102962).

In this report, we mainly develop the claim (Goal ID N27918261) that the implementation of the `AP` module (class) is correct. In turn, arguing this claim amounts to arguing that the implementation of the PID controller for each control surface is correct (Goal ID N42862332).

### 6.1.5 Linking the Controller Module to aileron control

As mentioned in Section 6.1.4, the correctness of the autopilot module of the avionics software is refined into the claim of correctness of the PID controller updates of the actuator values for the flight control surfaces. This claim is developed by arguing for the correctness of the PID controller over each control surface. In this report, in particular, we are concerned with the correctness of the controller updates for the aileron (so as to relate the fragment to the sequence of calculations in the aileron described earlier in Section 2.3.2).

In Figure 22, we show the safety case fragment in greater detail starting from the claim that the autopilot module is correct (Goal ID N82535547), linking it to the correct implementation of `AP` class (Goal ID N27918261), and subsequently to the claim of correct PID controller updates for each flight control surface (Goal ID N42862332). This claim is made in the context of the low-level functional safety requirement as given in Section 5.2. Finally, this is refined into two sub-claims about the correct implementation of the controllers for two flight control surfaces:

(i) N39596683: Implementation of PID controller is correct for Aileron control variable, and

(ii) N5207777: Implementation of PID controller is correct for Elevator control variable.

These two claims are created in the context of the identified low-level Base Year target safety requirements (**LL-SR-01** and **LL-SR-02** respectively), as given in Section 5.2. In the next section, we show how the safety argument to support these two claims are automatically generated.

## 6.2 Semi-automatically Generated Safety Case

Our goal is to combine manually created safety case fragments with automatically generated fragments. First, we recall the lowest level of the manually created safety case fragment for the Swift UAS, as shown in Figure 23. In Figure 23, a claim of correctness in the PID controller updates for flight control surfaces is broken down into sub-claims of correct updates to the aileron and elevator control variables in the autopilot, respectively. The final component in the safety case, the actual argument for correctness of the control variables, is derived automatically.

In this section, we first present the automatically generated safety-case fragments for the Swift UAS. Then, in section 7 we describe our method for semi-automatic generation of the safety argument to support the two claims mentioned above.

In Figure 24 we see a bird's eye view of the automatically generated safety case fragments combined with the manually created safety case fragment. In particular, only the two goals from Figure 23 are shown in the top left of Figure 24 (enclosed in the dotted box), to indicate the link to the manually created safety case fragment. The automatically generated components are the two right branching structures, one for each of the low-level requirements. In greater detail, Figure 25 shows the initial part of the automatically generated safety case fragment for the lowest level goals shown in Figure 23. (Goal IDs N39596683, and N5207777).

We support two ways of integrating manual and generated fragments. First, by tagging goals in ASCE [3] with the string `autocert:n`, indicating that this goal is to be merged with the fragment generated by verifying the $n$th goal given in a separate AUTOCERT specification file. Second, the integration script can extract the relevant goals from the safety case and directly call AUTOCERT.

[30] As a matter of convention, the goals are not stated in the same way as the requirements. The goals are stated as predicates which can be evaluated as TRUE or FALSE, instead of stating them using the convention for stating requirements, i.e., using "shall" statements.

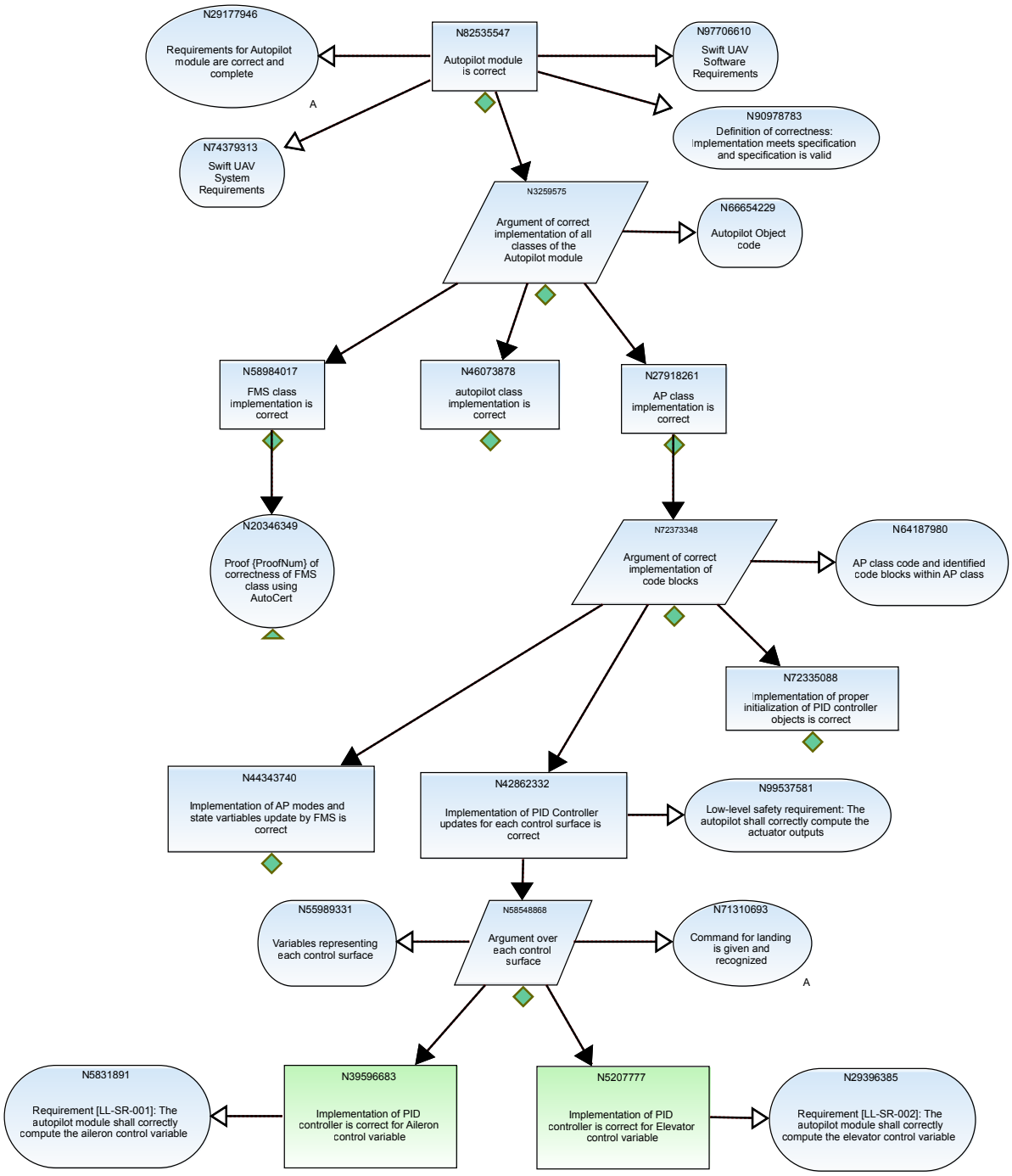



Figure 22: Safety Argument Fragment 05. Linking claims about the autopilot to claims about the `AP` controller to claims about the implementation of specific PID controllers for two flight control surfaces.

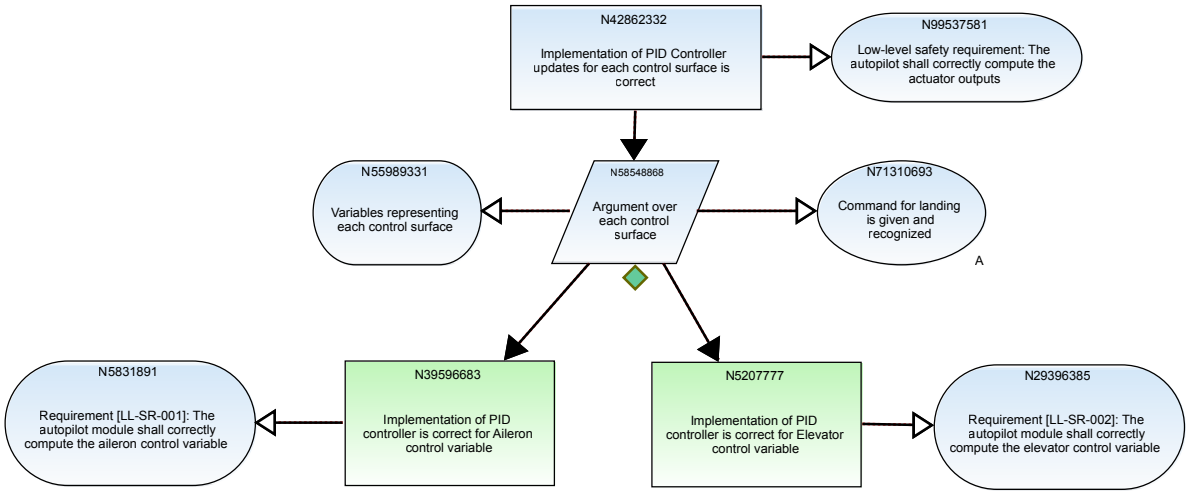



Figure 23: Safety case fragment linking claims about the autopilot to two flight control surfaces.

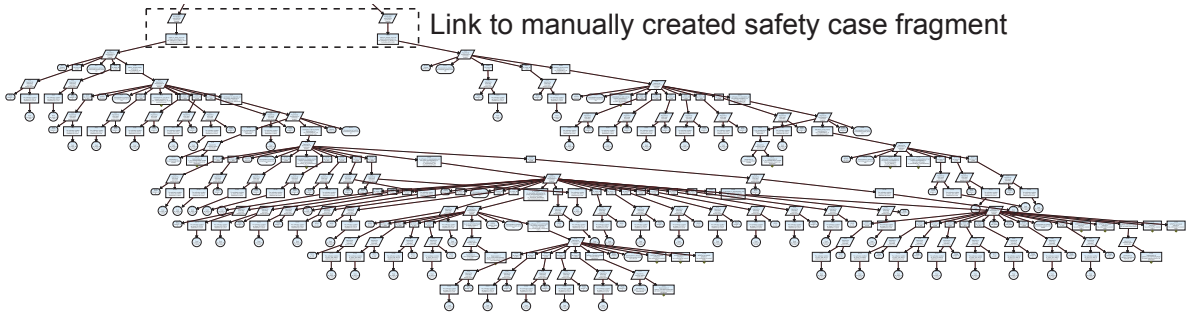



Figure 24: Bird's eye view of the automatically generated safety case for the Swift UAS

We run the AUTOCERT tool [17] to develop the claims in Figure 23 into a number of verification conditions (VCs). These formulas can then be discharged by automated theorem provers. AUTOCERT assumes that low-level library functions meet their specifications, and does not verify the bodies. Evidence of this has to be provided, therefore, either by testing, or by inspection. The fragment below each goal outlines the sequence of intermediate computations in the code used to establish the relevant goal: typically a property on a variable. Some of these intermediate steps correspond to lower level goals.

In Figure 25 we see details of the first and part of the second level underneath the point where the manually created and automatically generated safety case fragments were merged. The tree structure descending to the right, underneath the lowest strategy node (Strategy ID AC30) has not been shown here. The goal from the automatically generated safety case fragment has been merged with the goal from the manually created fragment. The strategy, model, and subsequent goal nodes are all automatically generated from XML taken from the AUTOCERT safety case generator. In this case, the argument is for the correctness of the aileron control variable `output->m_aileron_m1p1`, whose property is stated in the top level goal (Goal ID AC1).

This property is the condition that must be shown to hold, for the argument to hold. The strategy asserts that we show the correctness of the claim by decomposition of the correctness property, where the notion of decomposition is that embodied within AUTOCERT. The context (Context ID 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 verification conditions that must be shown to achieve this goal. These, too, are goal nodes. The diamond attached to the bottom of the goal node asserts that they are incomplete as no proofs have been generated. The claim (Goal ID AC28) represents a dependent variable, `m_rollError_rad` and its property. There

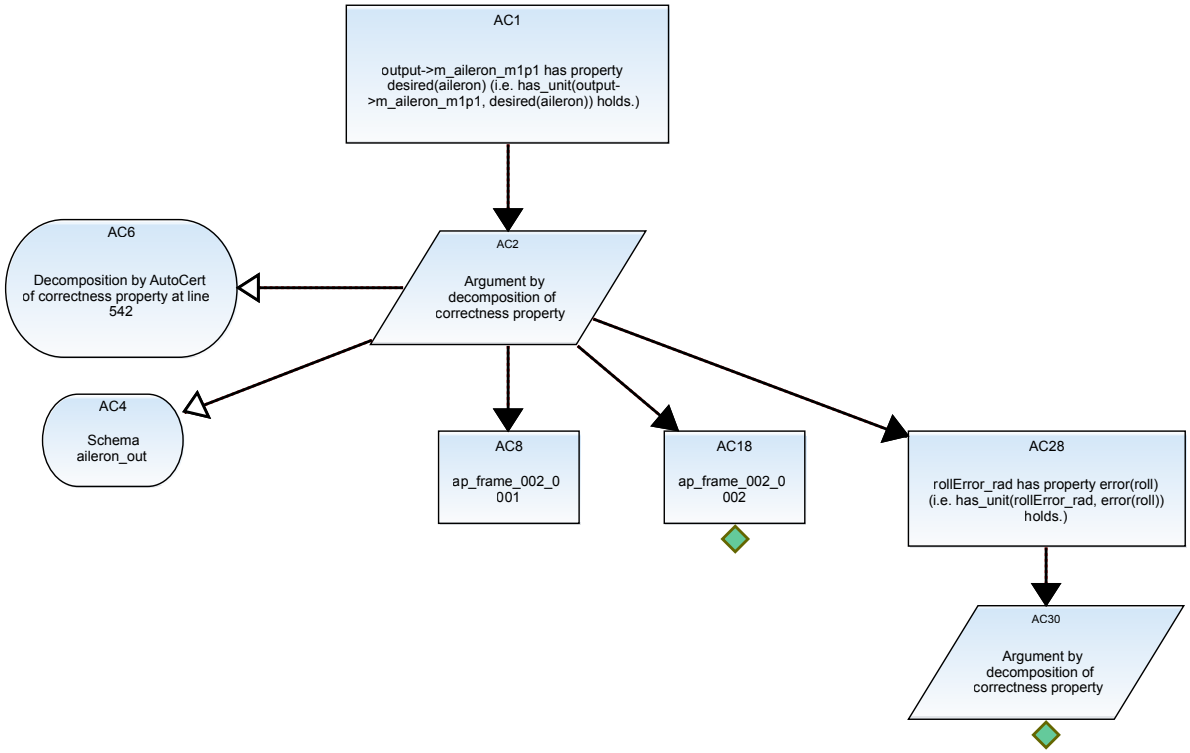



Figure 25: A step in the generated fragment of the safety case.

may be multiple dependent variables but in this case `output->m_aileron_m1p1` is dependent, directly, only on this variable. This goal represents the start of a recursive instance of the tree.

The auto-generated safety case fragment continues thus, until the reasoning reaches assumptions of the system or axioms. In the auto-generated safety case this is represented by complete leaf nodes. In our example, however (Figure 25), the argument is not yet complete, as shown by the GSN notation for undeveloped goals.

Theorem provers can be used to discharge the VCs. The proof provides evidence in the safety case, and the prover provides context. Figure 26 indicates that a proof was successfully found, using theorem prover `SSCPA--0.0`. The path to the proof object is shown (rather than the proof itself).

The safety case also needs to represent any assumptions that have been made about library functions, and what methods, if any, have been used to verify these. This information is obtained from a separately specified file (not shown here) and is represented in the safety case fragment as seen in Figure 27. Here, the function was inspected and a report was made of this inspection. In other cases the node could denote that testing was done with additional information giving a path to the test results.

Finally, the provers use various domain theories to discharge the VCs. The full text of the list of domain theories is hidden by the browser but in this case includes such theories as arithmetic reasoning and transformation geometry. Figure 28 shows a fragment of a semi-automatically created safety case narrative, corresponding to a step in the auto-generated fragment.

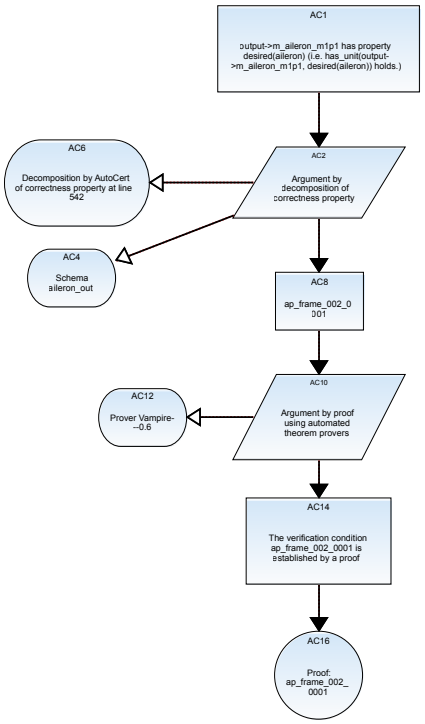



Figure 26: Proof of a VC by a Prover.

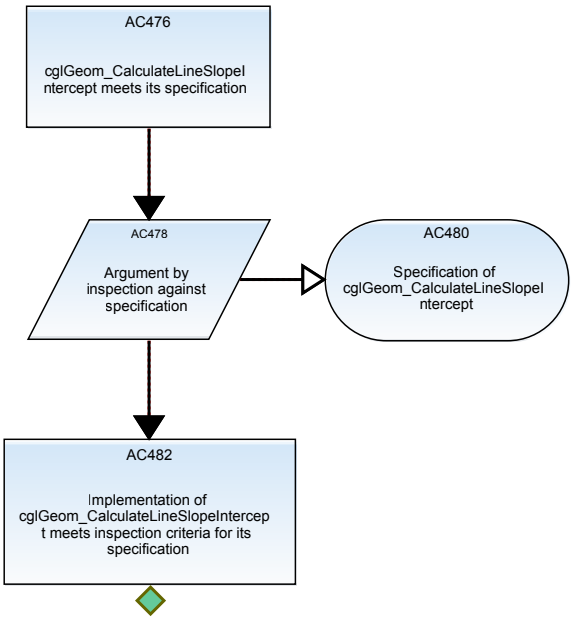



Figure 27: Inspection of a library function.

**Goal N39596683**: Implementation of PID controller is correct for aileron control variable `output->m_aileron_m1p1`. This goal requires us to formally prove that `ap` satisfies the following formal requirement:

```
has_unit(output->m_aileron_m1p1, 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_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 safety of the variable

- `output->m_aileron_m1p1`

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_m1p1` has a single relevant occurrence in the requirement. The safety 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 safety of the following variable:

- `rollError_rad`

Figure 28: Fragment of a draft safety case narrative

# 7 Transformation Methodology

## 7.1 Domain Theory

The automatically generated safety case fragments that are joined with the hand-generated fragments are derived from the certification process in AUTOCERT. The domain theory described in this section encompasses the mathematical knowledge necessary to certify specific aspects of the autopilot system, namely the aileron and elevator control variable calculations. This knowledge is compiled into AUTOCERT and used to generate a certification argument for the variables and components specified in the cert file, which, as seen in Section 7.4, can be created from the safety case fragment itself. One result of the certification process is the XML file containing all of the verification information necessary to automatically generate the safety case fragments representing the low level software component verification.

The domain theory consists of the annotation schemas, function specifications, and axioms that relate to aspects of the software component in question. Much of this domain theory can be seen in the resulting complete safety case. For instance from the annotation schema are derived the goals (and subgoals) of the safety case. The schemas are also used in the generation of verification conditions. The external functions in the code base are represented in AUTOCERT as function specifications. 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 in the safety case.

## 7.2 Domain Theory Description

Section 2.3 described the sequence of calculations used to compute the aileron control variable from several inputs drawn from the aircraft's current state and the 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 as a formal requirement and proven using a domain theory mainly formalizing geometric and navigational equations, and consisting of annotation schemas (Figures 29 and 30) and axioms (Figures 31 to 34). See [14] for the syntax of annotation schemas. We also need to specify properties of library functions (Figure 30). We emphasize that this domain theory is preliminary and has not yet been validated.

The axioms provide the necessary facts used to prove the generated verification conditions are correct. They mainly involve adjustments of orientations in radians through different quadrants based upon 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 preserved under addition or subtraction of $2\pi$. Others describe the validity of reversing the sign of a value while maintaining a property. Figure 35 gives the grammar of properties of the domain theory.

The purpose behind the axioms, schemas, and domain theory in general is to ultimately prove properties of the code via verification conditions. In the case of 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 current aircraft state. All 51 were proved using a suite of 5 automated theorem provers.

For the second requirement $output \rightarrow m\_elevator\_m1p1 :: desired(elevator)$, 13 verification conditions were generated. As for the first requirement, the verification conditions 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.

## 7.3 From Formal Proofs to Safety Cases

In this section we describe the transformation steps taking AutoCert generated XML to a safety case fragment. First we discuss the XML formats that define the data at each step. There are three steps, and hence three XML schemas, in this process. Then we describe the transformations that take place at each step and the architecture of the system.

```
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)]
, _::initial(heading)
, []
).

definition(crosstrack_error
, ['computing the cross track error']
,
(
cglGeom_CalculateLineSlopeIntercept(LineM, LineB, SrcWpPos, DstWpPos);
cglGeom_CalculateDistanceToLine(CurrACPos, LineM, LineB)
)
, [CurrACPos::pos(ne), SrcWpPos::pos(ne), DstWpPos::pos(ne)]
, _::distance
, []
).

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

function(distance_to_line
, ['distance to line']
,  cglGeom_CalculateDistanceToLine(CurrACPos, LineM, LineB)
, [CurrACPos::pos(ne), LineM::_, LineB::_]
, _::distance
, []
).

function(vector_dot_product
, ['dot product']
,  cglVec2_Dot(pA, pB)
, [pA::vec(3), pB::vec(3)]
, _::vec_dot(pA, pB)
, []
).

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

Figure 29: Math schemas.

```
definition(desired_heading
, ['computing the desired heading']
, [DesHead + DeltaHead]
, [DesHead::initial(heading), DeltaHead::desired(delta(heading))]
, _::desired(heading)
, []
).

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

function(desired_roll
, ['computing the desired roll']
,  'm_pid_HeadingErr2Roll->Update'(HeadingError)
, [HeadingError::error(heading)]
, _::desired(roll)
, []
).

function(desired_pitch
, ['computing the desired pitch']
,  'm_pid_AltitudeErr2Pitch->Update'(Y)
, [Y::error(altitude)]
, _::desired(pitch)
, []
).

function(elevator_out
, ['the elevator variable calculation']
,  'm_pid_PitchErr2Elevator->Update'(Y)
, [Y::error(pitch)]
, _::desired(elevator)
, []
).

function(aileron_out
, ['the aileron variable calculation']
,  'm_pid_RollErr2Aileron->Update1'(Y)
, [Y::error(roll)]
, _::desired(aileron)
, []
).
```

Figure 30: Math schemas (continued).

```
%----------------------------------------------------
% Autopilot Axioms
%----------------------------------------------------
%----------------------------------------
% mode distinctions
%----------------------------------------
fof(mode_contr2, axiom,
    (~('LONMODE_ALTITUDECMD' = 'LONMODE_PITCHCMD'))).

fof(mode_lat, axiom,
   (~('LATMODE_CROSSTRACK_NOFLYBACK' = 'LATMODE_CROSSTRACK'))).
%----------------------------------------
% error quadrant adjustments
%----------------------------------------
% roll
% E is in radians
% E < -PI
fof(roll_error_2_pi_plus, axiom,
    ! [E] : (
     (  has_unit(E, error(roll)) &
        lt(E, uminus(float_3_14159))    )
      =>  has_unit(plus(float_3_14159, plus(float_3_14159,E)), error(roll)) )).

% E > Pi
fof(roll_error_2_pi_minus, axiom,
    ! [E] : (
     (  has_unit(E, error(roll)) &
        lt(float_3_14159, E)   )
      => has_unit(plus(E, uminus(plus(float_3_14159, float_3_14159))), error(roll)) )).

% rollError
% E = Current - Desired
fof(roll_error_calc, axiom,
   ! [A,D,E] : (
    ( has_unit(A,current(roll)) & has_unit(D,desired(roll)))
    =>     has_unit(minus(A,D), error(roll))  )).

% E = Current - Desired
fof(heading_error_calc, axiom,
   ! [A,D] : (
    ( has_unit(A,current(heading)) & has_unit(D,desired(heading)))
    =>  has_unit(minus(A,D), error(heading)) )).

%---------------------------
% heading error adjustments
%---------------------------
%  E < -PI
fof(heading_error_2_pi_plus, axiom,
    ! [E] : (
     (   has_unit(E, error(heading))  &
         lt(E, uminus(float_3_14159))   )
      => has_unit(plus(float_3_14159, plus(float_3_14159,E)),
            error(heading))   )).

% E > PI
fof(heading_error_2_pi_minus, axiom,
    ! [E] : (
     (  has_unit(E, error(heading)) &
        lt(float_3_14159, E)   )
      =>  has_unit(plus(E,uminus(plus(float_3_14159, float_3_14159))),
                    error(heading))  )).
```

Figure 31: Axioms

```
%----------------------------------------
% current heading adjustment
%----------------------------------------
% H > PI
fof(current_heading_2_pi_minus, axiom,
   ! [H] : (
    ( has_unit(H, current(heading)) &
      lt(float_3_14159, H))
      => has_unit(plus(H, uminus(plus(float_3_14159, float_3_14159))),
              current(heading))  )).

% H < -PI
fof(current_heading_2_pi_plus, axiom,
   ! [H] : (
    ( has_unit(H, current(heading)) &
      lt(H, uminus(float_3_14159)))
      => has_unit(plus(float_3_14159, plus(float_3_14159,H)),
          current(heading))   )).
%------------------------
% desired heading and adjustments
%------------------------
% heading + delta_heading
fof(desired_from_delta, axiom,
   ! [X,D] : (
    ( has_unit(X,desired(delta(heading)))  & has_unit(D,initial(heading)))
    => has_unit(plus(D,X), desired(heading))  )).

% H < -PI
fof(desired_heading_2_pi_plus, axiom,
    ! [H] : (
    (  has_unit(H, desired(heading)) &
       lt(H, uminus(float_3_14159)) )
   => has_unit(plus(float_3_14159, plus(float_3_14159,H)),
              desired(heading))  )).

% H > PI
fof(desired_heading_2_pi_minus, axiom,
    ! [H] : (
    (  has_unit(H,  desired(heading)) &
       lt(float_3_14159, H) )
   =>  has_unit(minus(H,plus(float_3_14159,float_3_14159)),
           desired(heading))   )).

%--------------------------------
% pitch error adjustment
%--------------------------------
% E < -PI
fof(pitch_error_2_pi_plus, axiom,
  ! [E] : (
   ( has_unit(E, error(pitch)) &
     lt(E, tptp_uminus(tptp_float_3_14159)))
   => has_unit(plus(tptp_float_3_14159,plus(tptp_float_3_14159,E)),
              error(pitch)) )).

% PI < E
fof(pitch_error_2_pi_minus, axiom,
  ! [E] : (
   ( has_unit(E, error(pitch)) &
     lt(tptp_float_3_14159, E))
   => has_unit(plus(E, tptp_uminus(plus(tptp_float_3_14159,tptp_float_3_14159))),
              error(pitch))  )).
```

Figure 32: Axioms (continued)

```
%--------------------------------
% upper and lower bank limit axioms
%--------------------------------
% m_bankLimit_rad is a constant in the code
% case when desired bank >limit
fof(bank_limit_upper, axiom,  has_unit(m_bankLimit_rad, desired(roll))).

% case when desired bank < -limit
fof(bank_limit_lower, axiom, has_unit(uminus(m_bankLimit_rad),
desired(roll))).

%--------------------------------
% altitude and pitch
%--------------------------------
fof(altitude_error_calc, axiom,
  ! [A,D] : (
   ( has_unit(A,current(altitude)) & has_unit(D,desired(altitude)) )
   => has_unit(minus(A,D), error(altitude))  )).

% -PI <= (P - D) < PI
fof(error_pitch_calc, axiom,
  ! [P,D] : (
    ( has_unit(P, current(pitch)) & has_unit(D, desired(pitch)) )
     => has_unit(minus(P,D), error(pitch))  )).

%----------------------------------------
% Cross track delta heading adjustments
%----------------------------------------
fof(gXtrack_pos, axiom,
  lt(0, g_XtrackMaxCorrectionAngle_rad) ).

% reverse of delta heading
fof(delta_heading_minus, axiom,
 ! [D] : (
  ( has_unit(D, desired(delta(heading))) )
   => has_unit(uminus(D), desired(delta(heading))) )).

% g_XtrackMaxCorrectionAngle is delta heading
 fof(delta_heading_pos, axiom,
     has_unit(uminus(g_XtrackMaxCorrectionAngle_rad),
      desired(delta(heading)))  ).

%-------------------------------------------------------
% helper axioms for autopilot
%-------------------------------------------------------

fof(minus_minus_orig, axiom,
   ! [T] : ( tptp_uminus(tptp_uminus(T)) = T)).

fof(fabs, axiom,
  ! [X] : ( ( leq(0, X) ) => fabs(X) = X)).

fof(fabs_neg_1, axiom,
  ! [X] : ( ( lt(X,0) ) => fabs(X) = tptp_uminus(X))).
```

Figure 33: Axioms (continued)

```
%-----------------------------
% initial heading
%-----------------------------
% There are geometric calculations (conditions) in the
% initial heading assignments.  These affect the
% ordering of the atan2 arguments

% calculate initial heading angle
% PE Previous waypoint East
% PN Previous waypoint North
% WE Current waypoint East
% WN Current waypoint North

% standard direction, angle > PI/2
fof(initial_heading_1, axiom,
   ! [ H,  PE, PN, WE, WN ] : (
     (  has_unit(WN,pos(north)) & has_unit(WE,pos(east)) &
        has_unit(PN,pos(north)) & has_unit(PE,pos(east)) &
        H = atan2(minus(PE,WE), minus(PN,WN)) )
       => has_unit(H, initial(heading) )  )).

% need to reverse direction, angle < PI/2
fof(initial_heading_2, axiom,
   ! [ H,  PE, PN, WE, WN ] : (
     (  has_unit(WN,pos(north)) & has_unit(WE,pos(east)) &
        has_unit(PN,pos(north)) & has_unit(PE,pos(east)) &
        H = atan2(minus(WE,PE), minus(WN,PN)) )
       =>  has_unit(H, initial(heading) ) )).


fof(error_xtrack_from_slope_intercept, axiom,
 ! [ X, M, B , CE, CN] : (
    ( has_unit(CE, pos(east)) & has_unit(CN, pos(north)) &
      has_unit(M, slope) & has_unit(B, intersect) &
      X = divide(minus(minus(CE, times(CN,M)), B),
                         sqrt(succ(times(M,M)))))
      =>   has_unit(X, error(xtrack)) )).

% adjustment to assure that current and destination
% waypoints are different
% Src = source waypoint {north, east}
% WPN = m_waypointNorth_ft =(initially) Src[0]
fof(pos_update, axiom,
 ! [ WPN, Src, X] : (
  ( has_unit(WPN, pos(north)) &  has_unit(Src, pos(ne))  &
    X = update2(Src,0, plus(a_select2(Src,0), float_1_0)))
  =>  has_unit(X, pos(ne)) )).
```

Figure 34: Axioms (continued)

$$
\begin{array}{lcl}
T & ::= & heading \mid roll \mid aileron \mid xtrack \mid pitch \mid elevator \mid altitude \mid delta(T) \\
F & ::= & ba \mid lla \mid ne \mid north \mid east \\
U & ::= & desired(T) \mid current(T) \mid error(T) \mid pos(F)
\end{array}
$$

Figure 35: Grammar of domain specific terms

AUTOCERT generates an XML document with information describing the formal verification of requirements. The core of this is a chain of information relating requirements back to assumptions. Each step consists of an annotation schema for the definition, or "def", of a program variable, or "hotvar", the associated verification conditions that must be shown for the correctness of that definition, and the variables on which that variable, in turn, depends, or "dvars". We mine pertinent parts of this XML document to create a safety case fragment describing the chain of dependent variables, verification conditions, and definition schemas (considered as GSN models) that go into the verification of specific components or variables of a software system.

### 7.3.1 Formats

The first format we describe is a portion of the automatically generated XML from AUTOCERT. The syntax of the pertinent portions can be seen in Figure 36. This figure describes the XML tags holding data and defining its structure. Tags on the left hand side contain elements (or data) described on the right. Strings represent data components which get translated into the body of safety case nodes. Some of the tags and elements are self-explanatory.

The model name contains the identifier of the system under evaluation. Requirements come in two forms, system-requirements and component-requirements. For our purposes they contain the same information but describe verification of different levels of the software system. A requirement-id is a unique identifier assigned to the system or component requirement. This is only a small portion of the high level system elements. However these are the only elements we currently retrieve for the safety case fragments.

A *hotvar* is a variable in the software system that is under analysis. Creating the safety case requires a number of elements from *hotvar*s and the variables on which the *hotvar* depends. These dependent variables are also *hotvars*. The dependency is noted through the list of dependent variables in *dvar-list*. Each *dvar* is a *hotvar* elsewhere defined in the system. The *source-location* defines the *line* and *file* in which the *hotvar* is defined. The *safety-requirement* is the property which the *hotvar* must be shown to have. The *def-schema* and then *schema-name* gives the annotation pattern(s) which were applied to the *hotvar* in the source code to generate the verification conditions. The *def-vc-list* and its *vc* define the verification conditions which were generated, and hence must be shown to hold in order to verify the system or component requirement.

$$
\begin{array}{lcl}
\langle \textit{certification-info} \rangle & ::= & \langle \textit{model-name} \rangle \ \langle \textit{requirement-list} \rangle \\
\langle \textit{model-name} \rangle & ::= & \textit{String} \\
\langle \textit{requirement-list} \rangle & ::= & \langle \textit{system-requirement} \rangle^* \ \langle \textit{component-requirement} \rangle^* \\
\langle \textit{system-requirement} \rangle & ::= & \langle \textit{requirement-id} \rangle \ \langle \textit{hotvar} \rangle \\
\langle \textit{component-requirement} \rangle & ::= & \langle \textit{requirement-id} \rangle \ \langle \textit{hotvar} \rangle \\
\langle \textit{requirement-id} \rangle & ::= & \textit{String} \\
\langle \textit{hotvar} \rangle & ::= & \langle \textit{hotvar-name} \rangle \ \langle \textit{source-location} \rangle \ \langle \ \textit{safety-requirement} \rangle \\
 & & \langle \textit{def-vc-list} \rangle \ \langle \textit{dvar-list} \rangle \ \langle \textit{def-schema} \rangle \\
\langle \textit{hotvar-name} \rangle & ::= & \textit{String} \\
\langle \textit{source-location} \rangle & ::= & \langle \textit{line} \rangle \ \langle \textit{file} \rangle \\
\langle \textit{safety-requirement} \rangle & ::= & \textit{String} \\
\langle \textit{def-vc-list} \rangle & ::= & \langle \textit{vc} \rangle^* \\
\langle \textit{dvar-list} \rangle & ::= & \langle \textit{dvar} \rangle^* \\
\langle \textit{dvar} \rangle & ::= & \textit{String} \\
\langle \textit{def-schema} \rangle & ::= & \langle \textit{schema-name} \rangle \\
\langle \textit{schema-name} \rangle & ::= & \textit{String} \\
\langle \textit{line} \rangle & ::= & \textit{String} \\
\langle \textit{file} \rangle & ::= & \textit{String} \\
\langle \textit{vc} \rangle & ::= & \textit{String}
\end{array}
$$

Figure 36: Grammar of AUTOCERT generated XML (fragment)

The second XML schema (Figure 37) is an intermediate step between the AUTOCERT generated XML and

the safety case fragment format. It is generated by a XSLT transformation. A *safetycase* defines the root of the document. A *safetycase* can contain zero or more *argument*s. Each *argument* corresponds to a single *system-requirement* from the AUTOCERT generated document.

The *model-name* here is a model in the GSN sense, namely some data attached to a strategy (and not to be confused with the model names in Figure 36). In our case, these will be schema names. The *idReq* and *safety-req* are the same as *requirement-id* and *requirement-formula* in the syntax above. The *goal-def* defines the goal of the verification task in terms of the *hotvar* name, the *safety-requirement* and the location information. A *strategy-def* defines a strategy by which the *hotvar* will be shown to be correct. Typically this is encoded as *"Prove correctness of the computation at lines n to m"* where *nnn* is the line number from the location information above. The *subgoal-vc* elements define the verification conditions related to this particular variable. The *subgoal-dvar* defines a wrapper around a recursive instance of the XML schema already defined. Namely it defines the elements of a specific *dvar* from above, but in this case inserted into a tree-like structure. This structure allows us to represent, directly, the dependence of one variable upon another.

$$
\begin{array}{lcl}
\langle safetycase \rangle & ::= & \langle argument \rangle^* \\
\langle argument \rangle & ::= & \langle model\text{-}name \rangle^* \ \langle idReq \rangle \\
 & & \langle safety\text{-}req \rangle \ \langle goal\text{-}def \rangle \\
 & & \langle strategy\text{-}def \rangle \ \langle subgoal\text{-}vc \rangle^* \\
 & & \langle function\text{-}name \rangle \ \langle function\text{-}info \rangle \\
 & & \langle subgoal\text{-}dvar \rangle^* \\
\langle subgoal\text{-}dvar \rangle & ::= & \langle model\text{-}name \rangle^* \ \langle goal\text{-}def \rangle \\
 & & \langle strategy\text{-}def \rangle \ \langle subgoal\text{-}vc \rangle^* \\
 & & \langle function\text{-}name \rangle \ \langle function\text{-}info \rangle \\
 & & \langle subgoal\text{-}dvar \rangle^* \\
\langle vc\text{-}list \rangle & ::= & \langle vc \rangle
\end{array}
$$

Figure 37: Grammar of the intermediate XML document (fragment)

The final XML format is actually a proprietary format for the *Adelard ASCE* system, namely the XML-based `ASCE-GSN` notation, AXML[31]. The schema definition for the AXML is based on the GSN notation [40]. It can be found at [3]. The defined `ASCE-GSN` schema here defines either *node* or *link* tags. The *node* tags are GSN nodes. The type of node (e.g., Goal or Strategy) can be denoted as well as the title which defines the text displayed to the user. The node can also be given a reference. This reference can be used, in conjunction with the reference from another node, to define *link* elements. The *link* elements define arrows. Each of the tree structures defined in an intermediate XML file as defined above can be translated into a *node*. We can then define links between the related nodes.

### 7.3.2 Algorithm

We begin after the construction of the safety document XML file generated by AUTOCERT. This file contains the elements described in the previous section[32]. There are two steps to creating a complete safety case. The input to this set of transformations is the AUTOCERT generated XML document. This is transformed into an intermediate stage, which is another XML document. Finally the intermediate XML document is parsed and combined with a hand generated safety case fragment to create the complete safety case. We describe the details of this process. The overall architecture can be seen in Figure 38.

The first major and necessary step to creating a complete safety case involves running the generated XML file through an XSLT transformation. This involves relating the AUTOCERT XML file with an XSLT file to generate a new XML document. In this process we transform the AUTOCERT XML document into a document with the schema described in 37. The current method for doing this involves the Saxon XSLT processor (version 9 - home

[31] In the work described here, we generate AXML v1.3. In more recent work [15], we instead generate safety cases in an *Extended GSN* (EGSN) format which can either be rendered directly in our AdvoCATE tool, or translated into AXML to view in ASCE.

[32] Codebase: `https://babelfish.arc.nasa.gov/trac/autocert/browser/trunk/safetycase`

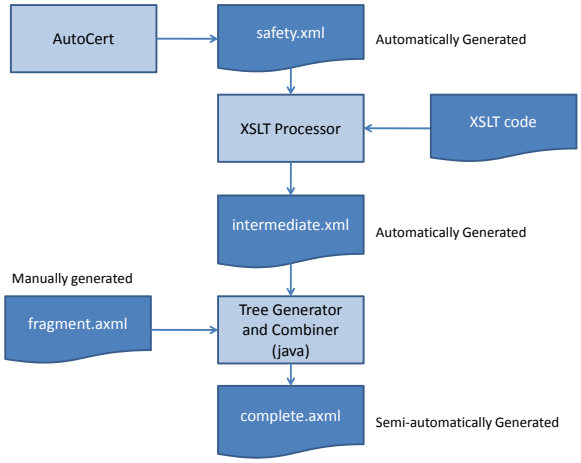



Figure 38: Architecture of the transformation from AUTOCERT XML to safety case AXML.

edition). The Saxon XSLT processor accepts an XML document and an XSLT file and generates the new XML document. The added benefit to this is that the syntactic well-formedness of both the input and output XML is validated in processing. The Saxon XSLT transformation will not generate syntactically incorrect XML.

The resulting intermediate XML file is then transformed into an AXML file. In this step it is also combined with the hand generated safety case fragment, which is also an AXML document. This process uses a Java program to read in both the hand generated fragment and the intermediate XML file. Both files are read and parsed as DOM objects. This allows for efficient processing on a node by node, element by element, basis. We want to retain all of the hand generated AXML and simply replace specific nodes with the tree fragments generated from the AUTOCERT XML file. Hence we prune the hand generated safety case and graft our tree in place.

The nodes in the hand generated document where the generated fragments are grafted are denoted with unique comments relating to a tree in the intermediate XML file, namely `autocert:n`. These nodes are identified and replaced by the top node from the intermediate XML file. The top node, and all subsequent nodes from the intermediate file, are transformed into the AXML format. Specifically for each element in the intermediate XML file we generate an AXML *node* element. These nodes are appended to the DOM structure created by parsing the hand generated AXML file. The nodes are created by traversing the DOM object tree created by parsing the intermediate XML file, again creating a node for each element in the intermediate files DOM object tree. By parsing the DOM object gives another validation of syntactically correct XML.

Once all of the nodes have been created, links elements are created for the new nodes. Links already existed for the hand generated fragment. This involves traversing the DOM object tree a second time. Links are created for each node to its children. Since we have grafted a new node from the intermediate XML tree in place of a pruned node from the hand generated tree we have have a connection between the hand generated tree to the newly generated nodes. The new DOM object is written out to a new file. Currently the AXML tree fragments created from the intermediate XML file are also written to a new AXML file and can be viewed separately.

### 7.3.3 Validating the Transformation

With respect to the automatic generation of XML and AXML we take a number of steps to validate the results. Some of the steps are implicit in the processing. The generation of the XML document from AUTOCERT is validated by the standard testing procedure for AUTOCERT results. This system is a set of scripts that run AUTOCERT against a number of examples and verifies that files have been created, verification conditions were

generated and that proofs were accomplished depending on a specific set of command line options. Different options generate different sets of results and output. In particular, safety case generation can be instigated by a command line switch. When XML is to be generated we check to verify that the XML file has been created. These results are then sent to user and logged.

At each step in the process of transforming the XML into a safety case fragment the syntactic correctness of the XML and AXML can be verified implicitly by the transformations themselves. Any error in the syntax of the XML will prevent it from being properly parsed and will result in an error being reported. In the case of the XSLT transformation, if a syntactic error is found in either the XML or the XSLT the XSLT processor will report the condition and fail to generate output. In the AXML generation phase, Java DOM objects will fail to load improper XML and throw an error which is caught by a set of exceptions built into the program. Finally, the `ASCE` system will fail to load AXML that is not properly formed or has duplicate nodes.

Finally we rely on visual inspection of the results in an appropriate viewer, such as `ASCE`. The resulting safety case is inspected with an eye towards assuring that the generated claims, arguments, and evidence are consistent with the stated safety properties. And hence that those safety properties hold for the referenced portions of code. These results of the inspection are recorded and the safety case itself will also act as evidence to this effect.

As described in the previous sections, we have run the tool on the Swift autopilot code, in particular file ap.cpp of the autopilot, in order to verify the two Base Year safety requirements. The resulting safety case consists of an argument that describes with a sequence of computations, such as delta heading, cross track error, initial heading, and so on. The argument states at which lines in the code these computations occur, and on what they depend. We have verified by inspection that each of these locations and the computation dependencies are correct.

## 7.4 From Safety Cases to Formal Specifications

A corollary product to the safety case combiner tool is a system that generates a cert file to be used in conjunction with AUTOCERT. A cert file contains meta-information about the system to be verified such as the requirement to be verified and any assumptions about the system. A safety-case fragment may often be created before the software verification is completed and hence illuminating software components that need to be verified. The cert file generator will create the necessary file for the verification process to proceed.

As noted, when combining nodes from the hand-generated and the automatically generated fragments, a unique identifier `autocert:n` is placed in the comment field of the node to indicate locations where one fragment should be grafted on to another. The node in the hand-generated case will contain information about the requirement needing to be verified and can also have associated nodes containing assumptions about the system. This information, when encoded into a hand-generated safety case fragment, can be extracted and used to create a specific cert file. The cert file then will be used to verify the system component relating to that requirement or requirements. By this method the hand-generated safety case fragment can be used to identify software components that are not verified and the cert file generator facilitates the generation of the necessary materials to do the verification.

The cert file generator looks through the hand-generated fragment for the unique `autocert:n` identifiers. Depending on the type of node the identifier occurs in the system is able to infer if the labeled node is a requirement or an assumption. Both are reformatted into an acceptable syntax for AUTOCERT. They are then written to a file. If a cert file already exists, that can be used as well, with the new requirements being appended to the end of the file. There is no limit to the number of requirements and assumptions that can be discovered and created.

In the autopilot case, the nodes representing the aileron and elevator control verification could have had text representing the actual system requirement being verified.

- *output.m_aileron_m1p1::desired(aileron)*
- *output.m_elevator_m1p1::desired(elevator)*

Both represent the actual variable names to verify and the properties they are assumed to have. Further, we could have had code level assumptions represented as nodes in the safety case fragment. For instance we may have assumptions in the safety case about the state of the aircraft.

- *'airplaneData→m_heading_rad' :: current(heading)*
- *'airplaneData→m_pos_altitude_ft' :: current(altitude)*

These are extracted from the safety case and wrapped in a *requirement* or *assumption* predicate, as needed. They are then written to a file where AUTOCERT can access it and verify those specific requirements.

# 8 Evaluation Metrics

This section describes the metrics which we use to evaluate the slice of the safety case created for the Swift UAS (Figure 16). In particular, we define metrics for four quantities[33] of interest, namely:

1. Coverage
2. Degree of Automation
3. Understandability
4. Uncertainty (Confidence)

Each of these is now described in greater detail.

## 8.1 Coverage

Coverage for the safety case can be interpreted in different ways. We initially consider four distinct notions of coverage, namely:

**Coverage of hazards:** Coverage of hazards measures the proportion of the hazards, identified during hazard analysis, that have been covered by the safety case. We define a measure $COV_H$ to quantify the coverage of hazards, whose valid values lie in the range $[0, 1]$.

**Coverage of high-level safety requirements:** Coverage of high-level safety requirements measures the proportion of high-level safety requirements (defined for eliminating/ mitigating hazards), covered by the safety case. We define a measure $COV_{\mathcal{R}_{HL}}$ for measuring the coverage of high-level requirements, whose valid values lie in the range $[0, 1]$.

**Coverage of low-level safety requirements:** Coverage of low-level safety requirements, i.e., the functional safety requirements, measures the proportion of low-level safety requirements (obtained from refining high-level safety requirements), covered by the safety case. We define a measure $COV_{\mathcal{R}_{LL}}$ for measuring the coverage of low-level requirements, whose valid values lie in the range $[0, 1]$.

**Coverage of claims:** Coverage of claims[34] measures the internal completeness of the safety case, i.e., the fraction of the total number of claims in the safety case, that have been instantiated and completely developed. We define a metric $COV_C$ for measuring the internal completeness, whose valid values lie in the range $[0, 1]$.

Note that $COV_{\mathcal{R}_{HL}}$ is correlated with $COV_H$, since safety requirements are derived from the hazard analysis; indeed, the former are directly linked to elimination or mitigation measures for those hazards categorized as having unacceptable risk. Since low-level safety requirements are derived from high-level safety requirements, $COV_{\mathcal{R}_{HL}}$ is also correlated with $COV_{\mathcal{R}_{LL}}$. Additionally, $COV_C$ is also correlated with $COV_H$ since several of the claims in the safety case reflect claims of mitigation or elimination of the relevant hazards. Here, we do not report on the exact relations between the coverage measures above, and it is left as future work.

[33] We use the terminology given in [59]; specifically: a *quantity* is a "property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a reference".

[34] In this report, *claims* are used interchangeably with *goals*.

### 8.1.1 Base Measures

The measures of coverage are derived from base measures, which are now defined.

- $H$: Number of identified hazards having unacceptable risk, is computed by counting those hazards classified as having unacceptable risk, in the list of hazards obtained from hazard analysis.
- $R_{HL}$: Total number of high-level safety requirements. Note that one or more high-level safety requirements may exist for mitigating or eliminating a hazard having unacceptable risk. Hence, if $r(H_i)$ is the number of high-level safety requirements per hazard $H_i$, then $R_{HL}$ is given as $\sum_i^H r(H_i)$.
- $R_{LL}$: Number of low-level safety requirements. Note that one or more low-level safety requirements may exist for each high-level safety requirement. If $r(R_i)$ is the number of low-level safety requirements per high level requirement $R_i$, then $R_{LL}$ is given as $\sum_i^{R_{HL}} r(R_i)$.
- Let $\mathbb{X} = \{D, UI, UD, UID\}$, be a set of claim categories, where
  1. $D$: *Developed* claims, refer to the claims which have been developed into sub-claims using strategies to the point where they can be tied to solutions (evidence). Thus, developed claims represent a complete chain of argumentation from evidence to claims (verification conditions).
  2. $UI$: *Uninstantiated* claims, refer to claims that have not been exactly instantiated. Thus, they may be considered as "placeholders" for a future claim to be included into an ongoing safety argument.
  3. $UD$: *Undeveloped* claims, refer to claims which have been instantiated but not developed, i.e., the exact claim is known but the strategies to decompose the claim down to sub-claims and eventually to tie it to specific evidence is not yet known or has not been included in the safety case.
  4. $UID$: *Uninstantiated and undeveloped* claims, refer to claims which require both instantiation and further development so they can be linked to evidence.
- Define the set $Claims_x(E)$ to mean the set of claims of category $x \in \mathbb{X}$ for entity $E$ (where $E$ is a hazard or a requirement). Then we have $c_x(E) = |Claims_x(E)|$ as the total number of claims of category $x \in \mathbb{X}$ for an entity E in the safety case.
  - If $E$ is a hazard, we have the measure $c_x(H_i) = |Claims_x(H_i)|$ reflecting the number of claims of category $x \in \mathbb{X}$ for a hazard $H_i$ where $i = (1 \ldots H)$. Thus, $c_D(H_1)$ measures the number of developed claims for hazard $H_1$, and the total number of developed claims for all hazards is given as $\sum_{i=1}^{H} c_D(H_i)$.
  - Similarly, if E is a (high-level or low-level) requirement, the measure $c_x(R_i) = |Claims_x(R_i)|$ reflects the number of claims of category $x \in \mathbb{X}$ per requirement $R_i$, where $i = (1 \ldots R_{HL})$ for high-level requirements and $i = (1 \ldots R_{LL})$ for low-level requirements. Thus $c_D(R_1)$ measures the number of developed claims for requirement $R_1$. The total number of developed claims for all high-level requirements $R_i^{HL}$ is given as $\sum_{i=1}^{R_{HL}} c_D(R_i^{HL})$, assuming that claims for separate requirements are disjoint. Similarly, the total number of developed claims for all low-level requirements $R_i^{LL}$ is given as $\sum_{i=1}^{R_{LL}} c_D(R_i^{LL})$, again assuming disjointness.
- $C_x$: Total number of claims in the safety case of category $x \in \mathbb{X}$, where $\mathbb{X}$ is defined as before. Here $C_{UI}$, for example, measures the total number of uninstantiated claims in the safety case, and $C_D$ measures the total number of developed claims in the safety case.
- $C$: the total number of claims. We have $C = \sum_{x \in \mathbb{X}} C_x$, since the separate claim classes are disjoint.

### 8.1.2 Measuring Coverage

We compute coverage measures from the base measures as:

1. The coverage of hazards is the fraction of the total number of *developed* claims for all hazards to the total number of claims for all hazards. Thus,

$$COV_H = \frac{\sum_{i=1}^{H} c_D(H_i)}{|\bigcup_{i=1}^{H} Claims_x(H_i)|} \tag{1}$$

   If all such claims are unique, i.e., the claims for each hazard are disjoint, this equates to

$$COV_H = \frac{\sum_{i=1}^{H} c_D(H_i)}{\sum_{i=1}^{H} \sum_{x \in \mathbb{X}} c_x(H_i)} \tag{2}$$

2. The coverage of high-level requirements is the fraction of the total number of *developed* claims for all high-level requirements to the total number of claims for all high-level requirements. Thus,

$$COV_{\mathcal{R}_{HL}} = \frac{\sum_{i=1}^{R_{HL}} c_D(R_i^{HL})}{|\bigcup_{i=1}^{R_{HL}} Claims_x(R_i^{HL})|} \tag{3}$$

   If all such claims are unique, i.e., the claims for each high-level requirement are disjoint, this equates to

$$COV_{\mathcal{R}_{HL}} = \frac{\sum_{i=1}^{R_{HL}} c_D(R_i^{HL})}{\sum_{i=1}^{R_{HL}} \sum_{x \in \mathbb{X}} c_x(R_i^{HL})} \tag{4}$$

3. Equivalently, the coverage of low-level requirements is the fraction of the total number of *developed* claims for all low-level requirements to the total number of claims for all low-level requirements. Thus,

$$COV_{\mathcal{R}_{LL}} = \frac{\sum_{i=1}^{R_{LL}} c_D(R_i^{LL})}{|\bigcup_{i=1}^{R_{LL}} Claims_x(R_i^{LL})|} \tag{5}$$

   If all such claims are unique, i.e., the claims for each low-level requirement are disjoint, this equates to

$$COV_{\mathcal{R}_{LL}} = \frac{\sum_{i=1}^{R_{LL}} c_D(R_i^{LL})}{\sum_{i=1}^{R_{LL}} \sum_{x \in \mathbb{X}} c_x(R_i^{LL})} \tag{6}$$

4. The coverage of claims (internal completeness) is the proportion of the total number of *developed* claims to the total number of claims. Thus,

$$COV_C = \frac{C - \sum_{\mathbb{X} \setminus \{D\}} C_x}{C} = \frac{C_D}{C} \tag{7}$$

### 8.1.3 Coverage for the Swift UAS Safety Case Fragment

We apply the measures described in sections 8.1.1 and 8.1.2, to the slice of the Swift UAS safety-case shown in Figure 16). Tables 5 and 6 summarize the corresponding measurements. Since the argument fragments are disjoint we can use summations in the calculations.

Based on Table 5 and Table 6, we make the following observations:

Table 5: Base measures (coverage) for the Swift UAS safety case.

| Measure | Value | Comments |
|---|---|---|
| $H$ | 14 | This reflects the number of failure hazards considered at the component level in the preliminary hazard analysis (PHA). |
| $R_{HL}$ | 6 | Reflects the total number of high-level safety requirements specified in the PHA. Of these, one requirement has been considered in the measurement, i.e., the base-year target safety requirement. |
| $R_{LL}$ | 2 | Reflects the number of low-level requirements derived from the base-year target safety requirement. |
| $C_D(H_1)$ | 149 | Total number of developed claims for the avionics failure hazard (identified here as $H_1$) in the UAV, during descent. |
| $C_D(H_2)$ | 2 | Total number of developed claims for the actuation failure hazard (identified here as $H_2$) in the UAV, during descent. |
| $C_D(H_{3\ldots14})$ | 0 | Total number of developed claims for each of the remaining hazards ($H'-2$). The value of this base measure is zero since the claims for these hazards have not been developed in the safety case fragment. |
| $C_D(R^{HL}{}_{A2})$ | 148 | Number of developed claims for the high-level requirement **A2**. Note that **A2** is the initial base-year target safety requirement, therefore we have only considered this requirement in the measurements. |
| $C_D$ | 157 | Total number of developed claims in the overall safety case fragment for the Swift UAS (including those claims not related to hazards identified in the hazard analysis) |
| $C$ | 220 | Total number of claims in the overall safety case fragment for the Swift UAS |

Table 6: Derived coverage measures for the Swift UAS safety case.

| Measure | Value | Comments |
|---|---|---|
| $COV_H$ | 0.74 | (i) The total number of developed claims for all considered hazards is given as $\sum_{i=1}^{H} c_D(H_i) = \sum_{i=1}^{14} c_D(H_i)$ = 149 + 2 = 151<br>(ii) The total number of claims for all hazards is given as $\sum_{i=1}^{H} \sum_{\mathbb{X}} c_x(H_i) = \sum_{i=1}^{14} \sum_{\mathbb{X}} c_x(H_i)$ = 204 |
| $COV_{\mathcal{R}_{HL}}$ | 0.8 | (i) The total number of developed claims for all considered high-level requirements is given as $\sum_{i=1}^{R_{HL}} c_D(R_i^{HL}) = \sum_{i=1}^{6} c_D(R_i^{HL}) = C_D(R_{A2}^{HL})$ = 148<br>(ii) The total number of claims for all considered high-level requirements is given as $\sum_{i=1}^{R_{HL}} \sum_{\mathbb{X}} c_x(R_i^{HL}) = \sum_{i=1}^{6} \sum_{\mathbb{X}} c_x(R_i^{HL})$ = 184 |
| $COV_{\mathcal{R}_{LL}}$ | 0.88 | (i) The total number of developed claims for all considered low-level requirements is given as $\sum_{i=1}^{R_{LL}} c_D(R_i^{LL}) = \sum_{i=1}^{2} c_D(R_i^{LL})$ = 136<br>(ii) The total number of claims for all considered low-level requirements is given as $\sum_{i=1}^{R_{LL}} \sum_{\mathbb{X}} c_x(R_i^{LL}) = \sum_{i=1}^{2} \sum_{\mathbb{X}} c_x(R_i^{LL})$ = 32 + 122 = 154 |
| $COV_C$ | 0.71 | Computed as $C_D/C$ = 157/220 |

- The quality of the computed measures depends on the quality of the artifacts being measured. With feedback on the validation of the hazard list and its subsequent refinement, we foresee changes to the measurements give in Table 5 and Table 6. Additionally, the coverage measures consider both the manually created safety case fragments and the automatically generated fragments together.

- The measurements are meaningful only in the context of the slice of the safety case (Figure 16), since a complete safety case for the Swift UAS is not yet available. For instance, the measures of coverage do not account for missing claims[35] related to five of the six high-level requirements, and twelve of the fourteen identified failure hazards considered in the PHA.

  Thus, the true coverage of hazards and the high-level requirements is less than that shown in Table 6, if the

[35]Note that these claims have not been included in the safety case fragment. The reason for their exclusion was mainly to focus on demonstrating the creation of an end-to-end safety case, rather than a comprehensive and complete one.

relevant missing claims are included in the fragment and subsequently measured. If we assume that at least one claim per hazard / high-level requirement must exist in the safety fragment, we may compute the error in our measures of hazard coverage and high-level requirements coverage, respectively, as $\epsilon(COV_H) = |\frac{151}{204+12} - \frac{151}{204}| = |0.70 - 0.74| = 0.04$, and $\epsilon(COV_{\mathcal{R}_{HL}}) = |\frac{148}{184+5} - \frac{148}{184}| = |0.78 - 0.80| = 0.02$. Furthermore, if the preliminary hazard list is also considered, the error values will increase since we have only counted the hazards relevant for building the safety case slice.

However, there is no well-defined way of knowing *a priori*, exactly how many claims a requirement or a hazard induces. By examining the base measure, e.g., $C_D(H_{3...14})$, as shown in Table 5, and the hazard coverage $COV_H$, as shown in Table 6, we have an indication regarding the true coverage of hazards and the extent to which the considered hazards are covered by the safety case. Effectively, we may now interpret (i) hazard coverage *with reference to the list of hazards* as the fraction of covered hazards (say $H_c$) to the total hazards, i.e., $\frac{H_c}{H} = \frac{2}{14}$ = 0.1428. (ii) extent of hazard coverage *by the safety case fragment* as $COV_H$ = 0.74.

- We believe that both measures are required to correctly gauge hazard coverage (assuming that the arguments chains on which measurement is applied, are themselves valid). Including the notion of argument validity into the coverage measures (either as a Boolean measure or a confidence measure), is a promising avenue for future work. Similarly, we can derive a measure of requirements coverage (with reference to the list of hazards), where the measure $COV_{\mathcal{R}_{HL}}$ indicates the extent to which the safety case covers these requirements.

- With respect to the measure $COV_{\mathcal{R}_{LL}}$, we measure perfect coverage since all claims that are developed for the low level requirements are both automatically generated, and terminate either in a solution or a verification condition. As we will see in section 8.2.2, although we measure perfect coverage of the low-level requirements by the safety argument, the extent to which the low-level requirements are actually covered (by the verification) is not perfect. The latter is measured using measures for degree of automation. This is described next.

## 8.2 Degree of Automation

To measure degree of automation, we consider two measures; namely:

1. $DEG_{A1}$, measuring the fraction of automatically generated claims to the total number of claims in the safety case. This measure effectively gauges how much of the safety case contains automatically generated claims, and its valid values lie in the range $[0, 1]$.

2. $DEG_{A2}$, measuring the fraction of the total amount of code for which verification conditions are generated to the amount of code that covers all the requirements processed. This measure gauges the extent to which automatically generated *sub-claims* cover an automatically generated *claim*, and the valid values lie in the range $[0, 1]$.

### 8.2.1 Base Measures

To compute the values of these measures for degree of automation, we define the following base measures:

- $C$: Total number of claims in the safety case. We can compute this as $C = \sum_{\mathbb{X}} C_x$, where $\mathbb{X}$, and $C_x$ are as defined in section 8.1.1.

- $C_A$: Total number of automatically generated claims in the safety case.

- $LOC(R_i)$: Lines of code corresponding to a requirement $R_i$. Since the automatic generation of safety claims is created from the proof of correctness of the code implementing a requirement, this base measure gives a gauge of the amount of code which must be automatically processed for covering a claim reflecting a specific requirement.

- $LOC_V(R_i)$: Verified lines of code, corresponding to a requirement $R_i$, gives a measure of the amount of code which has been automatically processed, for generating verification conditions.

Table 7 lists the measurements after applying these base measures.

Table 7: Base measures (degree of automation) for the Swift UAS safety case.

| Measure | Value | Comments |
|---|---|---|
| $C$ | 220 | From Table 5 |
| $LOC(R_1^{LL})$ | 44 | Auto-generation is applied starting at low-level requirements LL-SR-001 and LL-SR-002 respectively. This measure counts the number of LOC covering LL-SR-001. |
| $LOC_V(R_1^{LL})$ | 42 | Number of LOC corresponding to low-level requirement LL-SR-001 which are verified. |
| $LOC(R_2^{LL})$ | 7 | Auto-generation is applied starting at low-level requirements LL-SR-001 and LL-SR-002 respectively. This measure counts the number of LOC covering LL-SR-001 |
| $LOC_V(R_2^{LL})$ | 5 | Number of LOC corresponding to low-level requirement LL-SR-001 which are verified. |
| $C_A$ | 152 | Computed as the total number of (automatically generated) claims for all low level requirements. This term is given by the denominator of equation (6) as $\sum_{i=1}^{R_{LL}} \sum_{\mathbb{X}} c_x(R_i^{LL})$ |

### 8.2.2 Measuring Degree of Automation

Now, we compute the measures of degree of automation from the base measures as:

$$DEG_{A1} = \frac{C_A}{C} \tag{8}$$

Thus, $DEG_{A1} = 152/220 = 0.69$. We interpret this measure to mean that approximately 69% of the claims in the slice of the safety case for the Swift UAS are automatically generated. The proportion of claims which were not automatically generated is therefore given as $1 - DEG_{A1} = 0.31$. This proportion reflects the claims which are created from the hazard analysis and potentially could be automatically generated.

To compute the total fraction of the amount of code which is automatically processed, we compute $DEG_{A2}$ as:

$$DEG_{A2} = \frac{\sum_{i=1}^{R_{LL}} LOC_V(R_i)}{\sum_{i=1}^{R_{LL}} LOC(R_i)} \tag{9}$$

Thus, $DEG_{A2} = \left\{\sum_{i=1}^{2} LOC_V(R_i)\right\} / \left\{\sum_{i=1}^{2} LOC(R_i)\right\} = \frac{42+5}{44+7} = 0.9215$.

This measure is interpreted to mean that the automatically generated claims cover about 92% of the code representing the two low-level requirements (LL-SR-001 and LL-SR-002). This is attributed to portions of the code for which schemas were not created (and therefore was neither exercised by AUTOCERT, nor had claims automatically generated). This proportion of the code, i.e., $0.0785$ of the code for the low-level requirements, can also be processed for automatic generation of safety claims.

## 8.3 Understandability

### 8.3.1 Challenges to Measuring Understandability

Defining a measure of *understandability* or comprehensibility requires defining how we can quantify the degree to which a safety argument can be understood. One possible avenue for this is to define a subjective scale, say $\langle 0 \ldots 5 \rangle$, where $0$ implies incomprehensible, whereas $5$ implies a perfectly comprehensible safety argument.

Measurement of comprehensibility and quantification using such a scale requires subjective assessment by the relevant stakeholder.

Since a safety argument aggregates diverse sources of evidence, based on the stakeholder assessing the safety case, certain portions of the safety case are likely to be more comprehensible than other portions, e.g., domain knowledge about the Swift UAS (such as flight control theory used in creating the autopilot) referenced as context or justification or as evidence in the safety case, is likely to be better accessible and understandable to the subject matter expert than, say, to a stakeholder who has a lesser depth of aviation domain knowledge, or to a stakeholder who has a different area of expertise.

Hence, it is reasonable to expect that a subjective evaluation of certain aspects of the safety argument by a set of stakeholders who have the same background is likely to be consistent. However, the challenge in defining a measure of understandability for the overall safety argument, is creating a measure which reflects a consistent measurement, and not one which changes relative to the measurement instrument (in this case, the stakeholder), i.e., it is unlikely that two different stakeholders with different expertise will evaluate a given portion or all of the safety case consistently.

In [26], a guideline is given on the steps that may be taken to evaluate comprehensibility (subjectively); in particular, [26] gives a four-step process for argument evaluation, the first step of which is *argument comprehension*. The recommendation for comprehensibility evaluation is to identify key elements in the safety case, to identify the links between argument and evidence, and to re-represent the argument in discussion with the originator of the safety case.

### 8.3.2 Towards Measuring Understandability

A potential way to characterize the overall understandability is to compute the proportion of elements in the safety case which were understandable, to the total number of elements in the safety case.

Let $U_e$ be the understandability of an element of the safety case where $e \in \mathbb{E} : \{$Claim, Strategy, Context, Justification, Evidence$\}$. $U_e$ is measured on a scale $\langle$very low, low, medium, high, very high$\rangle$, corresponding to the degree of understandability.

Let $N(e)$ be the number of elements of type $e$ in the safety case, and $N(U_e)$ be the number of elements for which understandability is available on the defined scale. The measure of overall understandability of the safety argument in relation to a given element $\mathcal{U}_e$ may be given as

$$\mathcal{U}_e = \frac{N(U_e)}{\sum_{\mathbb{E}} N(e)}, e \in \mathbb{E} \tag{10}$$

Thus, we would interpret $\mathcal{U}_{Claim}$ as the proportion of claims in the overall safety argument which is understandable on the defined scale. On this basis, a measure of understandability may be provided for each element in the safety case.

## 8.4 Confidence in the Safety Argument

Despite the many potential advantages that a safety case can provide with respect to the explicit consideration of safety assurance, subjectivity inherent in the structure of the argument and its supporting evidence, as well as the lack of sufficient statistical data, pose a key challenge to the measurement and quantification of confidence in the overall safety case. Consequently, confidence in a safety case is often assessed by appealing to qualitative reasoning.

In this section, we explore the challenges of measuring confidence in safety cases; in particular, we propose an approach for confidence measurement by integrating probabilistic reasoning with Bayesian Networks (BNs) [38] into safety arguments represented in the Goal Structuring Notation (GSN) [39]. An overarching motivation is, eventually, to integrate the confidence measures obtained from safety arguments into a quantitative framework for risk analysis [53].

In our proposed approach for measuring confidence in safety cases, we build the *confidence* argument for the safety argument by quantifying the uncertainty in the latter, where applicable, by using BNs. In particular, we use BNs to measure the confidence in the claims made (and, as a consequence, in the argument) by computing

the joint distribution of a set of random variables (r.v.) that represent the quantified sources of uncertainty present in, and derived from, the safety argument.

### 8.4.1 Illustrative Example

Figure 39 shows an extract from the safety argument for the target system[36]. Through hazard analysis, we have determined that the safe functioning of the autopilot requires the correct calculation of the angle of attack of the aircraft (G1). Now, we discuss ways to measure confidence in the argument and quantify the uncertainty in this claim.

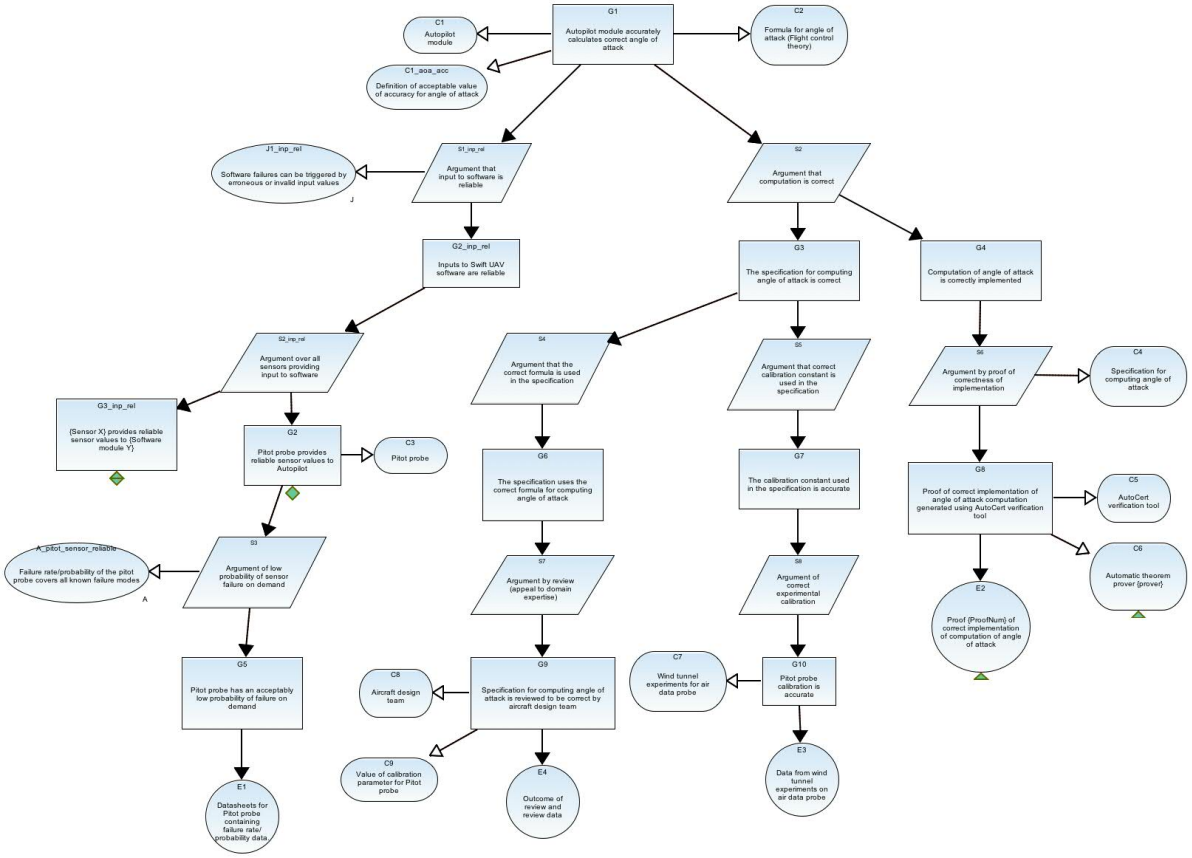



Figure 39: Safety argument fragment for correct angle of attack

We address G1 by arguing that

1. G1.1: the Pitot (air-data) probe provides the correct sensor values to the autopilot
2. G2.1: the specification is correct and
3. G2.2: the implementation of this specification is also correct.

In turn, these claims are justified in part (using the strategies shown in Figure 39)

1. E1: evidence arising from wind tunnel experiments calibrating the air-data probe
2. E2: subjective assessment of the formula used in the specification as evidenced by the outcome of a review,
3. E3: formal verification of the implementation, using a proof of correctness, and
4. E4: evidence of low probability of failure on demand (PFD) obtained from sensor datasheets.

[36]Note that this fragment represents a portion of the overall safety argument for the Swift UAS

Note that in figures 16, and 17, this argument fragment is shown in its factored form, i.e., divided into its main argument legs, which represent the development of each of the three goals G1.1, G2.1, and G2.2. The rationale for this refactoring is to keep the argument structure consistent with overall safety case fragment, where argument about specifications are separated from arguments about implementation and input. The argument structure as shown in Figure 39 has been constructed explicitly for the purpose of illustrating how we perform confidence measurement in a safety argument.

To gauge whether G1 is to be accepted, e.g., by a regulator, it is reasonable to present an additional argument to justify the sufficiency of confidence in the claim (and, as a consequence, the overall argument fragment shown). For instance, as in [28], a qualitative confidence argument may be created in which it is argued that (a) there is credible support for the inference asserted via the claims G1.1, G2.1 and G2.2 that G1 is true, (b) the assurance deficits for this asserted inference have been identified and (c) that the residual assurance deficits are acceptable. Unfortunately, although there is some guidance available on identifying where the assurance deficits lie [45], there is little guidance on *how* it may be gauged that the residual assurance deficit is acceptable. Here, the challenge for the regulator is in assessing that a qualitative argument (i.e., the confidence argument) provides sufficient confidence in another qualitative argument (i.e., the safety argument).

### 8.4.2 Uncertainty in the Safety Argument

The sources of uncertainty in the argument for G1, as shown in Figure 39, are mainly:

**(U1)** *Uncertainty in the sensor values* is stochastic (aleatory) and is attributed, in part, to the PFD of the Pitot probe, and to any errors in converting the sensed analog values to an appropriate digital equivalent. The former is given by the variance in the PFD (or measured failure rate in the case of continuous demand) obtained, say, through statistical testing of the sensor. We assume, for the sake of simplicity, that analog to digital conversion is perfect.

**(U2)** *Uncertainty that specification is correct* contains both aleatory and epistemic uncertainties: the calibration error of the Pitot probe (when the probe has not failed) is a source of aleatory uncertainty that contributes to the overall uncertainty in the correctness of the specification, whereas the uncertainty as to whether the formula for computing the angle of attack is itself correct and is correctly used is a source of epistemic uncertainty. Calibration of the air-data probe is experimentally performed in a wind tunnel [35]. A confidence level can be used to effectively specify the confidence in the experiment and is obtained from statistical analysis of the corresponding empirical data. The confidence that the correct formula is used to compute the angle of attack is subjectively gauged by reviewing the specification against flight control theory by domain experts, e.g., the aircraft design team.

**(U3)** *Uncertainty that the implementation is correct* is the uncertainty in the verification procedure i.e., the proof of correctness. The verification chain contains a combination of several steps and related tools [17] each of which induces an uncertainty that together contribute to the overall uncertainty that the proof is perfect. For this paper, we mainly gauge (U3) via subjective judgment from the developers of the verification tools. Modeling of the sources of uncertainty in the verification chain is left for future work.

Both (U2) and (U3) are epistemic uncertainties. Additional epistemic uncertainties arise from assurance deficits [28] in the safety argument itself, and are also subjectively quantified.

**(U4)** *Uncertainty in the sufficiency of the sub-claims* is the uncertainty whether the sub-claims, e.g., G1.1, G2.1, G2.2, are appropriate and sufficient to infer the parent claim (sub-claim), e.g., G1, or whether there is a need for additional sub-claims.

**(U5)** *Uncertainty in the appropriateness of the context* reflects on whether the context used for a claim or a strategy is appropriate and trustworthy.

### 8.4.3 Measuring Confidence

To assess the uncertainty (confidence) in the claim G1, first we model the confidence in the claim and the sources of uncertainty (U1) - (U5), respectively, as discrete r.v.; subsequently we characterize the overall confidence in the argument as the joint distribution of the r.v., and we use BNs to quantify this joint distribution. A Bayesian paradigm is appropriate in this context because it permits the inclusion of both subjective and quantitative data. Additionally, BNs allow us to (1) compute the joint distribution of r.v. by exploiting the conditional independence between the r.v. and (2) perform inference when evidence[37] is available. The structure of the network encodes the assumptions of conditional independence. Thus, the arcs represent dependencies between the r.v. and may be interpreted as correlation. Each of the r.v. has a defined set of states and an associated probability distribution over those states. Figure 40 shows the BN model which we use to model the sources of uncertainty and the confidence in the claim G1.

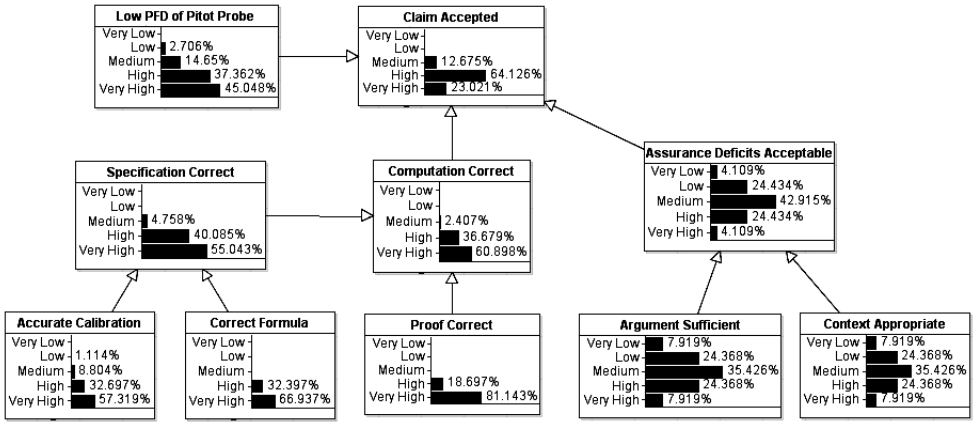



Figure 40: BN modeling sources of uncertainty and confidence for the argument fragment shown in Figure 39.

In Figure 40, the root node Claim Accepted (a node with only incoming arcs) of the BN models the confidence in the claim G1. The leaf nodes (nodes without incoming arcs) model each of the identified sources of uncertainty, e.g., the node Proof models the confidence in the solution E3: Proof of correctness, corresponding to the source of uncertainty (U3). The intermediate nodes (nodes with both incoming and outgoing arcs, e.g., Computation Correct) abstract and aggregate relevant leaf nodes; additionally, they serve to reduce the complexity associated with the specification of conditional probabilities and in post-specification inference.

All the nodes in the BN have the same set of five states: ⟨very low, low, medium, high, very high⟩ which are mapped to the interval [0, 1] as shown in Table 8. Such a mapping allows including confidence values that have been obtained from both quantitative data (e.g., the confidence level associated with the experimental calibration of the air data probe), and from qualitative means (e.g., the reviewer confidence in specification correctness).

Table 8: Mapping r.v. states to a unit interval

| State | Interval |
|---|---|
| Very Low | [0, 0.2) |
| Low | [0.2, 0.4) |
| Medium | [0.4, 0.6) |
| High | [0.6, 0.8) |
| Very High | [0.8, 1] |

[37]Note that evidence supplied in the BN is distinctly different from the evidence supplied in the safety argument itself. The former is evidence of increasing, decreasing, or complete credibility in the latter.

The quantitative specification for each of the leaf nodes is given as a prior probability distribution over the states of the node; in particular, we use a (doubly) truncated Normal distribution [23] whose mean is the prior belief (or measure) of confidence and the variance is picked so as to appropriately represent the confidence in this prior itself.

To obtain a truncated Normal distribution, one normalizes the probability mass of a standard Normal distribution, which has been truncated over the region [0, 1]. The resulting (truncated Normal) distribution is useful for modeling subjective belief since (a) the assignment of the probability mass to a specific subset of states of the RV is obtained by changing the mean and (b) a variety of distribution shapes are achieved by changing the variance [23].

For intermediate nodes and the root nodes we specify a prior conditional probability distribution (CPD) in a parametric way, again using a truncated Normal distribution. Here, the mean of the distribution is the weighted average of the parent r.v. while the variance is the inverse of the sum of the weights [23]. The weights can be considered as modeling the "strength of correlation" between the r.v. In the context of a safety argument, this would be viewed as the importance assigned to the contribution of a certain source of uncertainty to the overall confidence.

Thus, if $C_c$, $C_p$, $C_s$ and $C_{cc}$ are the r.v. modeling the confidence in the accurate calibration of the air data probe, the correctness of the proof, the correctness of the specification, and the correct computation respectively, $\pi(X)$ is a prior distribution over a random variable $X$, and $\mathcal{N}_T(\mu, \sigma^2)$ is the truncated Normal distribution with mean $\mu$ and variance $\sigma^2$, we have:

- $\pi(C_c) \sim \mathcal{N}_T(\mu_c, \sigma_c^2)$, where $\mu_c$ is given by the confidence measure of the experiment. In Figure 40, $\pi(C_c) \sim \mathcal{N}_T(0.9, 0.05)$ corresponds to the prior measure of a 90% confidence level in the calibration experiment of the air data probe.

- $\pi(C_p) \sim \mathcal{N}_T(\mu_p, \sigma_p^2)$, where $\mu_p$ is given by the subjective measure of confidence in the proof. In Figure 40, $\pi(C_p) \sim \mathcal{N}_T(0.9, 0.01)$ would be interpreted, for instance, as there being *a priori* "very high" confidence in the proof of correctness to be supplied as evidence.

- $\pi(C_{cc}|C_p, C_s) \sim \mathcal{N}_T(\mu_{cc}, \sigma_{cc}^2)$ is the CPD of the confidence in correct computation, given the confidence in the proof and the specification; $\mu_{cc}$ is given as $((100C_p + 100C_s)/200)$, i.e., the weighted average of the parent r.v., with each given equal weight; $\sigma_{cc}^2$ is chosen as the inverse of the sum of weights, i.e., 0.005.

The specification of the priors for the rest of the r.v. is given in a similar way. The BN of Figure 40 completely specifies the prior confidence in the overall argument; whereas the prior confidence to be expected in the claim, given the prior distribution of the parent r.v., is computed as $\{\text{high}\} \leftrightarrow \mathcal{N}_T(0.7257, 0.0145)$.

# 9 Discussion

## 9.1 Approach

In this report we have described the preliminary safety case fragments which we have assembled: a manually developed top-level system safety case, and lower-level fragments automatically generated from the formal verification of safety requirements. There are four distinguishing features to our work.

1. The argumentation in our safety case provides a level of detail which goes well beyond the state of the practice. Safety cases typically leave many details implicit or informal. Indeed, safety cases rarely go down to the level of software implementations. Making safety-relevant data and its connections to requirements explicit is highly worthwhile since a safety case serves primarily as a form of communication. The safety case is also useful, we believe, as a form of book-keeping: keeping track of data, and how it fits into the "big picture".
2. Safety cases do not generally combine manually developed and automated fragments. Where automation is used, it tends to be as a black box that provides a single piece of evidence, and not a full argument fragment. We believe much more can be done to increase the degree of rigor and formality.

3. We combine traditional safety analysis techniques with formal methods. Although there has been previous work on the use of formal methods in safety cases, much of this has ignored the safety aspect.
4. We combine diverse sources of information. Safety is inherently heterogeneous, and "formal" and "non-formal" should not be seen as opposites, but as complementary and equally important. Software is part of a system and parameters and specifications must be justified. For example, Table 9 shows a selection of variables defined in the target system code that have values derived from outside sources[38], and Figure 2 shows how our software verification methodology rests on connections between the formal specifications and the wider safety process.

## 9.2 Scope and Automation

It is in the nature of this project that detailed domain knowledge is required. However, our domain knowledge is limited and there has been an inevitable and non-trivial learning curve. Moreover, getting access to domain experts (who have, nevertheless, been very helpful) has also presented a bottleneck. However, we believe we have now laid a solid basis for further work.

Though 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, it is our intention to combine the results of multiple tools. 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 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 like this, and we plan to actively investigate integrating results from other tools.

Simulation is crucial when developing flight software, and Section 4 describes its use in the EAV flight software methodology (see also Appendix C). We would like to characterize more formally the simulation process and how it fits into subsequent analyses and decisions. Similarly, the testing of the flight software should provide evidence, both current testing practice, and its combination with formal techniques, such as the use of AUTOCERT-derived verification conditions to test code and library functions.

Other forms of static analysis should be applied to verify that signals lie within physical bounds, the absence of run-time errors such as division by zero and the degree to which a function is defined, numerical properties such as accuracy and robustness, and concurrency properties such as freedom from priority inversion and deadlocks. In general, for each class of property, there are specific tools which are better suited for their analysis.

Our hazard analysis (Section 5.1) 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 defined system boundaries.

## 9.3 Trustworthiness

We have identified several challenges in characterizing the trustworthiness of a safety case, i.e., via quantifying confidence in a safety argument. These challeges are mainly relevant to validating the model used for quantifying confidence.

(i) First, we believe that justifying the basic BN structure and the assumptions of conditional independence could be achieved, in part, by automatically generating the BN from the GSN-based safety argument, where for

[38] We have not yet considered all of these in our preliminary safety case.

Table 9: Defined variables and their values (excerpt)

| File/Class | Variable Name | Default value in code | Default value from script | Units | Uses |
|---|---|---|---|---|---|
| ap.h/.cpp | m_bankAngleLimit_rad | 35.0 * CGL_MATH_RAD_PER_DEG | | radians | |
| | m_pTerrainDB | NULL | | class object | "if instantiated it can be set to a file containing terrain data" |
| | m_altitudeMode | ALTITUDEMODE_MSL | | enumerated type | "MSL = Mean Sea Level – Barometric; Default value in ap class. Alternative values can be AGL = Above Ground Level or RADAR– changes set via scripts" |
| | m_bankLimit_rad | 40.0* CGL_MATH_RAD_PER_DEG | | radians | |
| | m_minThrottle_m1p1 | -0.8 | | controller output | |
| | m_maxThrottle_m1p1 | 0.8 | | controller output | |
| | g_XtrackMaxCorrectionAngle_rad | 45.0* CGL_MATH_RAD_PER_DEG | | radians | "Maximum cross track correction angle. Limiting value on heading corrections. (i.e. The maximum value that the aircraft can correct for: limited by banking angle (?)) " |
| fms.h/.cpp | m_landingFlareMaxAlt_ftAGL | 100.0 | 100 | feet | |
| | m_landingFlareMinAlt_ftAGL | 50.0 | 50 | feet | |
| | m_landingWheelsDownAlt_ft_AGL | 2.0 | 2 | feet | |
| | m_landingDescentRate_fps | -20.0 | | feet per second | |
| | m_landingFlareMaxDescentRate_fps | -15.0 | | feet per second | |
| | m_landingFlareMinDescentRate_fps | -0.1 | | feet per second | |
| | m_landingFlareThrottle_m1p1 | -0.65 | -0.65 | controller output | |
| | m_landingGlideSlope_rad | 15.0* CGL_MATH_RAD_PER_DEG | | radians | |

each source of uncertainty identified, a corresponding node (or nodes) exists in the BN.

(ii) Next, specifying leaf node probabilities and the prior CPD for the relevant intermediate/root nodes is straightforward, where empirical data is available. When only subjective judgment is available, quantifying confidence and selecting an appropriate prior distribution is problematic despite extensive research on belief elicitation methods [8].

We believe that one way to address this issue is to identify metrics using techniques such as the Goal-Question-Metric (GQM) method [5] and to correlate these metrics to confidence levels based on a defined quality model, e.g., we hypothesize that a metric such as coverage (by a safety argument) of hazards (in a hazard list) would correlate with the confidence in the sufficiency of the argument.

(iii) Finally, we require greater investigation to justify the weights used in specifying CPD requires. Assuming that the strategies used to decompose goals are viewed as being equally important, using equal weights appears to be a reasonable way forward.

Our preliminary investigation has emphasized the importance of treating assurance in an integrated way through linking qualitative safety arguments to quantitative arguments about uncertainty and confidence. This integration reaps the benefits of GSN in clearly communicating safety arguments to the many stakeholders of the safety case, while ensuring rigor in measuring confidence via probabilistic reasoning using BNs. We believe that when integrated into an engineering processes, the safety arguments in this approach will influence the development, assessment and management activities, whereas the confidence arguments will influence the level of rigor required in these activities to achieve the desired level of confidence in the safety arguments.

# 10 Future Work

We are developing a methodology and toolchain for combining automatically generated "bottom-up" safety case fragments, with manually developed "top-down" system safety cases, and have shown here how we can combine AUTOCERT-generated formal verifications with other safety case fragments. The safety case fragments which we have developed are just one small part of what we anticipate will be the eventual safety case. For example, one necessary extension will be to include information from the aircraft design process (see Figure 41 in Appendix C).

This is an example-driven project. Our study of the target system, its associated domain theory, and discussions with domain experts have been invaluable in uncovering issues in the automated construction of heterogenous safety cases. First and foremost, we plan to continue this collaboration. Although we have concentrated on developing safety case fragments for the Swift UAS, we intend to look at other configurations and specific missions in the EAV group's family of UASs. Just as the underlying Reflection architecture is intended as a plug and play embedded real-time environment, we might aim for "plug and play safety cases" (or product lines). Another interesting topic relating to safety cases "in the large" would be to develop techniques which allow us to show that changes to a safe aircraft *preserve* safety.

We now list some specific lines of work that we believe are worth pursuing.

**More formal verification:** The low-level target safety requirements that we looked at were based on functional and physical unit correctness. There are other properties that should be checked for in this code, such as runtime safety and physics-based bounds. We also want to look at combining results from different verification tools, as discussed in Section 9.

**Automated assembly:** We aim to develop a mark-up language (perhaps based on XML) for declaring safety-relevant information, in particular evidence. By stating what the data shows, assumes, and depends on, we anticipate that we will be able to automatically assemble more parts of the safety case.

Relating to this, it would be worth constraining the language used in nodes of the safety case. Such a *controlled natural language* could be based on an ontology, and would allow us to further characterize coverage, well-formedness, comprehensibility, and would allow us to more precise express queries. For example, assertions of expert opinion could be limited to certain classes of statement.

**Inclusion of formal verification knowledge:** There are many different ways of converting formal verification knowledge into safety case fragments. However, the structure of the safety case is driven by both the safety

and the verification methodology. Examples of high-level choices include whether to decompose over all requirements vs all scenarios vs all code branches.

Rather than hard-coding these design choices in the transformation, they could be represented declaratively using templates. This 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 in a dag-like manner, though doing this naively will inevitably lead to spaghetti diagrams.

The automated generation of safety case narratives will build on previous work on the generation of safety documents [14]. Holloway has written on the use of structured text formats for safety cases [30] and the draft OMG structured argument metamodel [2] also supports text formats.

We have initially targeted the ASCE safety case format, but aim to support support multiple formats such as the D-Case safety case format [54] from Kinoshita and Takeyama. We also intend to look at ideas from hiproofs [4, 16] to impose hierarchical structuring on the generated safety case fragments, since the level of schema steps will be too low-level for some purposes.

**Alignment with NASA Standards, NPR and Guidelines:** NASA has numerous safety-relevant procedural requirements, standards, and guidelines at both the software and system levels (Appendix D). It will be important to develop a safety case methodology which is aligned with these.

We also plan to continue the uncertainty/confidence work, both the theoretical basis and its application to our target system. We expect that this can be used to provide a basis for Probabilistic Risk Assessment [53].

**Evaluation measures:** The measures we described to objectively evaluate our approach (Section 8), were manually applied, as were the measurements that were collected as a consequence. Future work will involve the definition of procedures and algorithms to collect metrics automatically, to the extent possible. An important aspect of the safety case methodology is determining when the safety case is trustworthy enough, i.e., determining that the residual uncertainty in the safety argument and the claims made is acceptably small. This aspect also has important implications on the "completeness" of the safety case, i.e., whether all the relevant hazards have been addressed; specifically, the hazards which are a consequence of (unforeseen) system interactions are of particular importance [42].

We hypothesize that capturing domain knowledge in a knowledge base or an ontology might provide a reasonable way to check whether hazards addressed in the system safety case correspond well with the domain knowledge represented in the knowledge base. More specifically, whether hazards addressed in the safety case miss references to concepts or relationships specified in the domain model.

**Safety case manipulation:** The inclusion of automatically generated fragments, and all relevant sources of information, will lead to increasingly large safety cases. Since the primary motivation of safety cases is communication of safety relevant information to the various participants in the safety process, we believe that a safety case should not be viewed as a static and unchanging artifact but that, rather, should be amenable to manipulation in various automated ways. The *Query-View-Transformation* (QVT) standard for model manipulation could be applied here. Some examples include:

**Queries** What parts of a safety case need to be changed given a change in the underlying system, or in supporting evidence?

**Views** Show different views of the safety case. Abstract a high-level view of safety, say at the level of ConOps. For example, view the case by mission phase (as opposed to, say, by subsystem). Different views for different stakeholders (e.g., system developer vs software engineer vs project manager). Show claim nodes of different classes. Extract the list of expert assertions.

**Transformations** Map a safety case to other useful artifacts, e.g., a diagnostic model [52], or runtime monitors. Diagnostic models can be defined using Bayesian networks of failure nodes, so there may be an interesting link to our confidence work.

**Hierarchical structure:** Verification using AUTOCERT results in an argument structure which is derived from the tree of computations. However, reasoning at the level of computations and lines of code may well be too low-level for some purposes, so one possibility is to impose an additional level of hierarchical structure using *hiproofs* [4, 16]. This would also provide a unified theoretical basis for querying safety cases. *Safety case patterns* [41] could provide another possible structuring mechanism.

In addition to these avenues of future work, a necessary area of future work[39], is to align the safety analysis with the existing development process. Specifically, in Section 3.4, we identified some system requirements relevant for safety. Of those, we have only verified that requirement **R12** can be tied to our hazard analysis and subsequently be reflected in the safety case fragments (See Table 10, in Appendix A, as well as Figure 19). To insure that the system safety analysis is reflected in ongoing development (and that any information from the development which is relevant for the safety case is included), it is imperative to close the loop of verifying the relevance of the identified system requirements to the safety of the Swift UAS. Specifically this implies identifying and analysis those hazards which are implicitly addressed by the system requirements, and ensuring that the corresponding safety requirements are explicitly stated.

In the option years, we had originally proposed:

1. Specification of extended safety case components and data definition language
2. Automatic generation of extended safety case and safety case narrative
3. Specification of techniques to query, view, and transform safety cases

Each of these items relates directly to topics outlined above. Further work on the target system safety case (Item 2) will drive the research. Generation of safety case narratives should be a straightforward extension, and will ease use of the technology. Item 1 relates to the automated assembly of sources of information, and Item 3 covers the various manipulations of safety cases described above. This also relates to the generation of narratives. Finally, as our work progresses and matures, we anticipate developing recommendations for a safety case methodology that is aligned with NASA standards and procedures.

[39] Primarily from an engineering perspective and relatively less so from a research viewpoint.

# References


[1] JO-3 Flight Operations Manual Moffett Federal Airfield.

[2] System assurance task force: Argument metamodel. Technical report, Object Management Group, February 2010. Sysa/10-02-6.

[3] Adelard LLP. Assurance and Safety Case Environment (ASCE). `http://www.adelard.com/asce/`, Last accessed 2011.

[4] David Aspinall, Ewen Denney, and Christoph Lüth. Querying proofs (work in progress). In *Conference on Intelligent Computer Mathematics*, Bertinoro, Italy, 2011.

[5] V. Basili, G. Caldiera, and D. Rombach. Goal question metric approach. In *Encyclopedia of Software Engineering*, pages 528–532. John Wiley, 1994.

[6] Nurlida Basir, Ewen Denney, and Bernd Fischer. Constructing a safety case for automatically generated code from formal program verification information. In *The 27th International Conference on Computer Safety, Reliability and Security (SafeComp '08)*, Newcastle, England, 2008.

[7] Nurlida Basir, Ewen Denney, and Bernd Fischer. Deriving safety cases for the formal safety certification of automatically generated code. In *International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert '08)*, Budapest, Hungary, 2008.

[8] P. Bishop, R. Bloomfield, B. Littlewood, A. Povyakalo, and D. Wright. Towards a formalism for conservative claims about the dependability of software-based systems. *IEEE Transactions on Software Engineering*, 37(5):708 – 717, 2011.

[9] R. Bloomfield and P. Bishop. Safety and assurance cases: Past, present and possible future – an Adelard perspective. In *Proceedings of the 18th Safety-Critical Systems Symposium*, Feb. 2010.

[10] R.E. Bloomfield, B. Littlewood, and D. Wright. Confidence: its roles in dependability cases for risk assessment. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

[11] Reece A. Clothier, Jennifer L. Palmer, Rodney A. Walker, and Neale L. Fulton. Definition of airworthiness categories for civil unmanned aircraft systems (UAS). In *27th International Congress of the Aeronautical Sciences*, Acropolis Conference Centre, Nice, June 2010.

[12] Reece A. Clothier, Jennifer L. Palmer, Rodney A. Walker, and Neale L. Fulton. Definition of an airworthiness certification framework for civil unmanned aircraft systems. *Safety Science*, 49(6):871–885, 2011.

[13] K. Douglas Davis. Unmanned Aircraft Systems Operations in the U.S. National Airspace System. Interim Operational Approval Guidance 08-01, Mar. 2008.

[14] Ewen Denney and Bernd Fischer. A verification-driven approach to traceability and documentation for auto-generated mathematical software. In *Automated Software Engineering (ASE '09)*, 2009.

[15] Ewen Denney, Ganesh Pai, and Josef Pohl. AdvoCATE: An Assurance Case Automation Toolset. In *Proceedings of the Workshop on Next Generation of System Assurance Approaches for Safety Critical Systems (SASSUR)*, 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP), Sep. 2012.

[16] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A hierarchical notion of proof tree. In Martin Escardó, Achim Jung, and Michael Mislove, editors, *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI), May 2005*, volume 155 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 341–359. Elsevier Science Direct, May 2006.

[17] Ewen Denney and Steven Trac. A software safety certification tool for automatically generated guidance, navigation and control code. In *IEEE Aerospace Conference Electronic Proceedings*, Big Sky, Montana., 2008. IEEE.

[18] J.B. Dugan, S.J. Bavuso, and M.A. Boyd. Fault trees and markov models for reliability analysis of fault tolerant systems. *Journal of Reliability Engineering and System Safety*, 39:291–307, 1993.

[19] J.B. Dugan, S.J. Bavuso, and M.A. Boyd. Dynamic fault tree models for fault tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–373, Sept. 1992.

[20] FAA. *Safety Risk Management Order 8040.4*. Federal Aviation Administration, Jun. 1998.

[21] FAA Air Traffic Organization. *Safety Management System Manual version 2.1*. Federal Aviation Administration, May 2008.

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

[23] N.E. Fenton, M. Neil, and J.G. Caballero. Using ranked nodes to model qualitative judgments in bayesian networks. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1420–1432, Oct. 2007.

[24] N.E. Fenton and S.L. Pfleeger. *Software metrics: A rigorous and practical approach*. International Thomson Computer Press, 1997.

[25] Goal Structuring Notation Working Group. GSN Community Standard Version 1, Nov. 2011.

[26] Ibrahim Habli. Reviewing and evaluating safety cases. Tutorial, Feb. 2011.

[27] Charles Haddon-Cave. The Nimrod Review: An independent review into the broader issues surrounding the loss of the RAF Nimrod MR2 Aircraft XV230 in Afghanistan in 2006. Report, The Stationery Office, London, UK, Oct. 2009.

[28] R. Hawkins, T. Kelly, J. Knight, and P. Graydon. A new approach to creating clear safety arguments. In *Proceedings of the Safety Critical Systems Symposium*, Feb. 2011.

[29] K.J. Hayhurst, J.M. Maddalon, P.S. Miner, M.P. DeWalt, and G.F. McCormick. Unmanned aircraft hazards and their implications for regulation. In *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, pages 1–12, October 2006.

[30] C.M. Holloway. Safety case notations: Alternatives for the non-graphically inclined? In *Proceedings of the IET 3nd International Conference on System Safety*, Savoy Place, London, 2008. IET Press.

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

[32] C. Ippolito. A vision for greener aviation, Swift UAS Design Management Plan.

[33] Corey Ippolito. Autopilot Software Design Documentation, design.vsd.

[34] Corey Ippolito. Design of an Autonomous Control System for a Small-Scale UAV.

[35] Corey Ippolito. Wind tunnel calibration of the exploration aerial vehicle (EAV) five-hole pitot probe, August 2006.

[36] Corey Ippolito. Reflection Programmer's Manual. NASA Ames Research Center, December 2009.

[37] Corey Ippolito. Experimental autonomous vehicles (EAV) laboratory. Presentation, May 2010.

[38] F.V. Jensen. *Bayesian Networks and Decision Graphs*. Springer-Verlag, 2001.

[39] T. Kelly and R. Weaver. The goal structuring notation – a safety argument notation. In *Proceedings of the Dependable Systems and Networks Workshop on Assurance Cases*, Jul. 2004.

[40] Tim Kelly. *Arguing Safety: A Systematic Approach to Managing Safety Cases*. PhD thesis, University of York, 1998.

[41] Tim Kelly and John McDermid. Safety case patterns – reusing successful arguments. In *Proceedings of IEE Colloquium on Understanding Patterns and Their Application to System Engineering*, 1998.

[42] N. G. Leveson. A new approach to hazard analysis for complex systems. In *Intl. Conference of the System Safety Society*, Aug. 2003.

[43] N.G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[44] 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 Transactions on Software Engineering*, 33(5):347–365, May 2007.

[45] C. Menon, R. Hawkins, and J. McDermid. Interim standard of best practice on software in the context of DS 00-56 Issue 4. Standard of Best Practice Issue 1, Software Systems Engineering Initiative, University of York, 2009.

[46] Eugene A. Morelli. Advances in experiment design for high performance aircraft.

[47] NASA Aircraft Management Division. *NPR 7900.3C, Aircraft Operations Management Manual*. NASA, Jul. 2011.

[48] NASA Ames Research Center, Office of the Director. Airworthiness and Flight Safety. APR 1740.1 Ames Procedural Requirements, Sept. 2011.

[49] National Aeronautics and Space Administration (NASA). Facility System Safety Guidebook. NASA-STD-8719.7, Jan. 1998.

[50] NIST. Metrics and measures. http://samate.nist.gov/index.php/Metrics_and_Measures.html.

[51] James Reason. *Human Error*. Cambridge University Press, 1990.

[52] Ashok Srivastava and Johann Schumann. The case for software health management. In *4th IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2011.

[53] M. Stamatelatos et al. Probabilistic risk assessment. Procedures and Guide for NASA managers and practitioners 1.1, NASA Office of Safety and Mission Assurance, Aug. 2002.

[54] Makoto Takeyama. A note on D-Cases as proofs as programs. Technical report, National Institute of Advanced Industrial Science and Technology, Osaka, Japan, 2010. AIST-PS-2010-007.

[55] UK Ministry of Defence (MoD). *Safety Management Requirements for Defence Systems*, 2007.

[56] U.S. Department of Defense (DoD). Procedures for performing a Failure Modes, Effects and Criticality Analysis. MIL-STD-1629A, Nov. 1980.

[57] U.S. Department of Defense (DoD). Standard Practice for System Safety. MIL-STD-882D, Feb. 2000.

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

[59] Working Group 2 of the Joint Committee for Guides in Metrology (JCGM/WG 2). International vocabulary of metrology - basic and general concepts and associated terms. Technical Report JCGM 200:2008, Bureau International des Poids et Mesures (BIPM), 2008.

# A Traceability

Traceability between the different (safety) requirements and sub-requirements (which form the functional safety requirements), to the identified hazards, goals within the safety argument, evidence and external sources of information is shown in Table 10. The table represents a requirements traceability matrix, and is partially filled, reflecting ongoing work.

Table 10: Requirements traceability matrix

| Requirement | Sub-Requirement | Hazard | Goal | Evidence | External | Schema | Axiom |
|---|---|---|---|---|---|---|---|
| | | | TRACE TO | | | | |
| Actuators must not interfere or collide with existing structure | | PHA_DE.APP_ACT_001, PHA_DE.APP_ACT_002, PHA_DE.APP_ACT_003, PHA_DE.APP_ACT_004 | G24_UAV_Descent_Actuation | E31_UAV_Descent _Actuation | Flight-day procedures | | |
| | | | | | | | |
| Commands must be interpreted correctly | | PHA_DE.APP_AVCS_014, PHA_DE.APP_AVCS_015 | | | | | |
| No command shall make the autopilot execute an unsafe maneuver | | PHA_DE.APP_AVCS_014, PHA_DE.APP_AVCS_015 | | | | | |
| State information is accurate | | | | | | | |
| | | | | | | | |
| The autopilot is correct | | PHA_DE.APP_AVCS_015 | G16_UAV_Descent_Avionics_ SW_Autopilot_autopilot, G6_UAV_Descent_Avionics_ SW_Autopilot_autopilot | E15_UAV_Descent_ Avionics_SW_Autopilot_ autopilot | | | |
| | The system is properly initialized | | | | | | |
| | aircraft state information is properly received from the sensors | | G27_UAV_Descent_Avionics_S W_Autopilot_autopilot | | | | |
| | The current, previous, and next waypoints are properly defined | | G28_UAV_Descent_Avionics_ SW_Autopilot_autopilot | | | | |
| | The FMS object is properly initialized | | G29_UAV_Descent_Avionics_ SW_Autopilot_autopilot | | | | |
| | The AP object is properly initialized | | G30_UAV_Descent_Avionics_ SW_Autopilot_autopilot | | | | |
| | Input and Output variables are properly routed via the reflection systems scripts | | | | | | |
| | Parameter data is properly initialized. | | | | | | |
| The AP system creates correct output for all aircraft control surfaces | | PHA_DE.APP_AVCS_013 | G23_UAV_Descent_Avionics _SW_Autopilot_AP , G9_UAV_Descent_Avionics _SW_Autopilot_AP | | | | |
| | The autopilot correctly initializes the AP object | | G29_UAV_Descent_Avionics_ SW_Autopilot_autopilot | | | | |
| | The FMS system correctly updates the AP modes and state variables. | | G41_UAV_Descent_Avionics _SW_AP, G104_UAV_Descent_Avionics_S W_AP | | | | |
| | PID controller objects are properly initialized | | G44_UAV_Descent_Avionics _SW_Autopilot_AP, G75_UAV_Descent_Avionics _SW_Autopilot_AP | | | | |
| | PID controller updates are correct for each aircraft controller surface | | G45_UAV_Descent_Avionics _SW_Autopilot_AP | E28_UAV_Descent_ Avionics_SW_Auto pilot_AP_Aileron, E44_m_aileron_ m1p1 | Specification review by SME | | |

# B Parameters and Variables

This appendix contains descriptions of the parameters and variables used in the instance of the autopilot code. The parameters and variables are defined and initialized in a number of locations, including the code base and in the scripting language that instantiates the instance of the autopilot in the Reflection virtual machine.

Table 11 contains variables and system constants defined in the code base. They are initialized either in the code or in the scripts that load the modules into the Reflection VM. The script values will over ride the constructor initialized values.

Tables 12 and 13 describe the initial gain values for the PID controllers used in the autopilot system. Table 12 describes the PID controller gain and bounds set in the code. Table 13 shows the values set through the scripts after a module (namely the autopilot) is loaded into the Reflection virtual machine. As can be seen the gain values can be re-initialized through the scripts. The maximum and minimum values are only initialized in the code. Using the scripts to redefine the gain values allows the researcher to experiment, on the fly, with different response characteristics without having to rebuild the code base.

Tables 14 and 15 show variables that are regulated in the code. By regulated it is meant that the variables are constrained to be within explicit bounds by a macro function that is called after assignments to those variables. The first table, Table 14, lists variables that have unique upper and lower bounds. Table 15 contains variables that contain values in radians. These are regulated at plus or minus 2*PI.

The final two tables 16 and 17, show variables that represent external values coming into the `autopilot` module and aircraft data from sensors stored in the `gs111m` module. Table 16 shows the variable name, the units and use of the value stored in that variable. There is also a cross reference to Table 17 showing the name of the variable in the `gs111m` module. Some sensor information, collected in the `gs111m` module, is not used in the autopilot. However it is logged and sent to the ground station. It may possibly be used in other modules in the system.

Table 11: Defined variables and their values

| File/Class | Variable Name | Default value in code | Default value from script | Units | Uses |
|---|---|---|---|---|---|
| ap.h/.cpp | m_bankAngleLimit_rad | 35.0 * CGL_MATH_RAD_PER_DEG | | radians | |
| | m_pTerrainDB | NULL | | class object | if instantiated, it can be set to a file containing terrain data |
| | m_altitudeMode | ALTITUDEMODE_MSL | | enumerated type | MSL = Mean Sea Level – Barometric, Default value in ap class. Alternative values can be AGL = Above Ground Level or RADAR changes set via scripts |
| | m_bankLimit_rad | 40.0*CGL_MATH_RAD_PER_DEG | | radians | |
| | m_minThrottle_m1p1 | -0.8 | | controller output | |
| | m_maxThrottle_m1p1 | 0.8 | | controller output | |
| | g_XtrackMaxCorrectionAngle_rad | 45.0 * CGL_MATH_RAD_PER_DEG | | radians | Maximum cross track correction angle. Limiting value on heading corrections. (i.e. The maximum value that the aircraft can correct for, limited by banking angle (?)) |
| fms.h/.cpp | m_landingFlareMaxAlt_ftAGL | 100.0 | 100 | feet | |
| | m_landingFlareMinAlt_ftAGL | 50.0 | 50 | feet | |
| | m_landingWheelsDownAlt_ft_AGL | 2.0 | 2 | feet | |
| | m_landingDescentRate_fps | -20.0 | | feet per second | |
| | m_landingFlareMaxDescentRate_fps | -15.0 | | feet per second | |
| | m_landingFlareMinDescentRate_fps | -0.1 | | feet per second | |
| | m_landingFlareThrottle_m1p1 | -0.65 | -0.65 | controller output | |
| | m_landingGlideSlope_rad | 15.0*CGL_MATH_RAD_PER_DEG | | radians | |

Table 12: PID controllers and initial settings in code

| Variable name | pGain | iGain | dGain | iMax | iMin |
|---|---|---|---|---|---|
| m_pid_RollErr2Aileron | 1.0 | 0.0 | 0.0 | 1000.0 | -1000.0 |
| m_pid_HeadingErr2Roll | -1.0 | 0.0 | 0.0 | 1000.0 | -1000.0 |
| m_pid_CircleDistErr2Heading | 0.01 | 0.0 | 0.0 | 1000.0 | -1000.0 |
| m_pid_CrossTrackErr2Heading | -1.0 | 0.0 | 0.0 | 1000.0 | -1000.0 |
| m_pid_Yacc2Rudder | 0.01 | 0.0 | 0.0 | 1000.0 | -1000.0 |
| m_pid_HeadingErr2Rudder | 0.01 | 0.0 | 0.0 | 1000.0 | -1000.0 |
| m_pid_PitchErr2Elevator | 5.0 | 0.05 | 0.0 | 1000.0 | -1000.0 |
| m_pid_AltitudeErr2Pitch | -0.005 | -0.00005 | 0.0 | 1000.0 | -1000.0 |
| m_pid_AirspeedErr2Pitch | 0.01 | 0.0001 | 0.0 | 1000.0 | -1000.0 |
| m_pid_VertSpeedErr2Pitch | -0.01 | -0.001 | 0.0 | 1000.0 | -1000.0 |
| m_pid_AirspeedErr2Throttle | -0.01 | -0.001 | 0.0 | 1000.0 | -1000.0 |
| m_pid_AltitudeErr2Throttle | -0.005 | -0.00005 | 0.0 | 1000.0 | -1000.0 |
| m_pid_GlideSlopeDevThrottle | 0.01 | 0 | 0.0 | 1000.0 | -1000.0 |

Table 13: PID controllers and initial settings in scripts

| Variable name | pGain | iGain | dGain |
|---|---|---|---|
| m_pid_RollErr2Aileron | 0.5 | 0.0 | 0.0 |
| m_pid_HeadingErr2Roll | -1.0 | 0.0 | 0.0 |
| m_pid_CircleDistErr2Heading | 0.00 | 0.00001 | 0.0 |
| m_pid_CrossTrackErr2Heading | 0.004 | 0.0 | 0.0 |
| m_pid_Yacc2Rudder | 0.00 | 0.0 | 0.0 |
| m_pid_HeadingErr2Rudder | 0.03 | | |
| m_pid_PitchErr2Elevator | 1.0 | 0.0001 | 0.0 |
| m_pid_AltitudeErr2Pitch | not set | | |
| m_pid_AirspeedErr2Pitch | 0.05 | 0.0001 | 0.0 |
| m_pid_VertSpeedErr2Pitch | -0.013 | 0.000 | 0.0 |
| m_pid_AirspeedErr2Throttle | -0.175 | 0.000 | 0.0 |
| m_pid_AltitudeErr2Throttle | -0.060 | -0.000100 | 0.0 |
| m_pid_GlideSlopeDevThrottle | not set | | |

Table 14: Variables regulated by a defined value in PID loops.

| Variable | Min | Max | Units | Use of Variable | Calculated In |
|---|---|---|---|---|---|
| output.m_aileron_m1p1 | -1.0 | 1.0 | controller output | output to actuator | ap |
| m_pidTargets.-m_desiredroll_rad | - m_bankLimit_rad | m_bankLimit_rad | radians | internal calculation | ap |
| m_pidTargets.-m_desiredpitch_rad | -20 * CGL_MATH_-RAD_PER_DEG | 20 * CGL_MATH_-RAD_PER_DEG | radians | internal calculation | ap |
| output.m_elevator_m1p1 | -1.0 | 1.0 | controller output | output to actuator | ap |
| output.m_rudder_m1p1 | -1.0 | 1.0 | controller output | output to actuator | ap |
| output.m_throttle_m1p1 | -1.0 | 1.0 | controller output | output to actuator | ap |
| m_pidTargets.-m_xtracksignal_-deltaHeading | -(CGL_MATH_PI * 0.5) | (CGL_MATH_PI * 0.5) | radians | internal calculation | ap |
| m_pidTargets.-m_xtracksignal_-deltaHeading (second instance) | -g_XtrackMax-CorrectionAngle_rad | g_XtrackMax-CorrectionAngle_rad | radians | internal calculation | ap |

Table 15: Variables regulated plus or minus 2 * PI in PID loops. If the value of the variable is greater than PI, the variable is regulated to -2*PI. If the value of the variable is less than -PI the variable is set to 2*PI

| **Variable** | **>PI** | **<-PI** | **Units** | **Use of variable** | **Calculated In** |
|---|---|---|---|---|---|
| airplaneData.m_heading_rad | -2*PI | 2*PI | radians | internal calculation | ap |
| m_pidTargets.m_desiredheading_rad | -2*PI | 2*PI | radians | internal calculation | ap |
| headingError_rad | -2*PI | 2*PI | radians | internal calculation | ap |
| rollError_rad | -2*PI | 2*PI | radians | internal calculation | ap |
| pitchError_rad | -2*PI | 2*PI | radians | internal calculation | ap |
| glideSlopeDev_rad | -2*PI | 2*PI | radians | internal calculation | ap |

Table 16: Data from `gs111m` sensors/module used in the Autopilot.

| **Variable** | **Units** | **Defines** | `gs111m` **Source Variable** |
|---|---|---|---|
| m_pos_north_ft | feet | aircraft position, north of origin | m_posNorth_ft |
| m_pos_east_ft | feet | aircraft position, east of origin | m_posEast_ft |
| m_pos_altitude_ft | feet | aircraft altitude | m_posUp_ft |
| m_airspeed_fps | feet per second | aircraft airspeed | m_ias_fps |
| m_vertspeed_fps | feet per second | aircraft vertical speed | m_velUp_fps |
| m_accel_Y_fps2 | feet per second squared | aircraft acceleration | m_accelY_ba_fps2 |
| m_pitch_rad | radians | aircraft pitch angle | m_eulerPitch_rad |
| m_roll_rad | radians | aircraft roll angle | m_eulerRoll_rad |
| m_heading_rad | radians | aircraft heading | m_trueHeading_rad |
| m_flightPathAngle_rad | radians | aircraft path angle | un-enabled |

Table 17: Relevant data collected in the `gs111m` module

| **Property** | **Variable** | **Units** | **Defines** | **Used in autopilot as** |
|---|---|---|---|---|
| Indicated Airspeed | m_ias_fps | feet per second | indicated airspeed (true airspeed at ground level) from pitot tube | m_airspeed_fps |
| Velocity | m_NorthVelocity_fps | feet per second | vector North Velocity | |
| Velocity | m_EastVelocity_fps | feet per second | vector East Velocity | |
| Position | m_posNorth_ft | feet | aircraft position, north of origin from GPS | m_pos_north_ft |
| Position | m_posEast_ft | feet | aircraft position, east of origin from GPS | m_pos_east_ft |
| Acceleration | m_accelX_ba_fps2 | feet per second squared | acceleration in X component | unused in EAV |
| Acceleration | m_accelY_ba_fps2 | feet per second squared | acceleration in Y component | m_accel_Y_fps2 |
| Acceleration | m_accelZ_ba_fps2 | feet per second squared | acceleration in Z component | unused in EAV |
| Vertical | m_velUp_fps | feet per second | vertical airspeed | m_vertspeed_fps |
| Attack | m_attackAngle_rad | radians | attack angle (alpha) | unused in EAV |
| Side slip | m_sideslipAngle_rad | radians | side slip angle (beta) | unused in EAV |
| Pitch | m_eulerPitch_rad | radians | Euler Pitch | m_pitch_rad |
| Roll | m_eulerRoll_rad | radians | Euler Roll | m_roll_rad |
| Heading | m_trueHeading_rad | radians | True Heading | m_heading_rad |
| True Airspeed | | | see indicated airspeed | |
| Angular Velocity | | | rate gyros + kalman filter | unused in EAV |
| Battery Voltage | m_rawBattVolt_byte | volts | Battery Voltage (collected by modem module) | |

# C Aircraft Design

Figure 41 shows the control system design process as being implemented for the Swift UAV. Roughly, the diagram can be read in a left to right, top to bottom fashion. The process starts by generating models of the intended system. These models then relate to the actual hardware to be mounted on the aircraft, and they also provide the 6-DOF simulation[40] that will be used to fine tune the implementation on the UAV. The resulting system is iteratively tested and refined with different parameters. At each iteration it is tuned against the simulation and further refined.

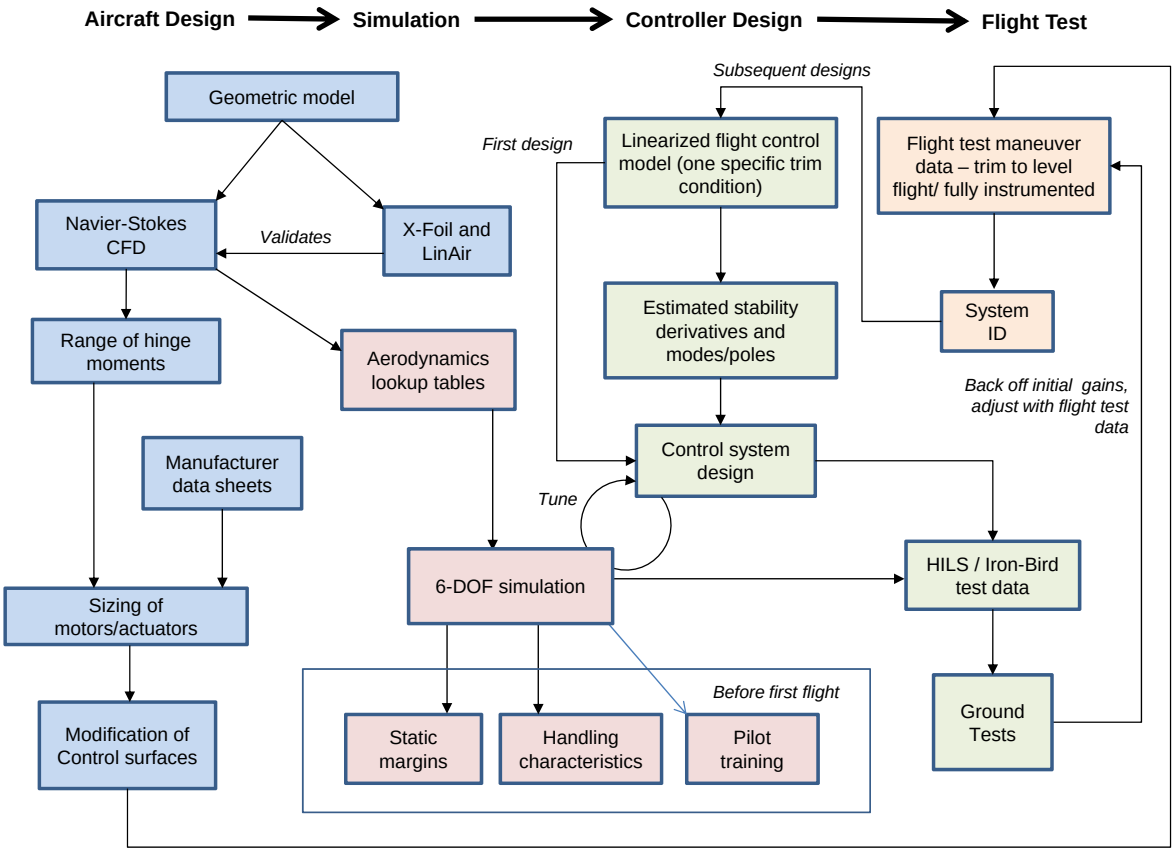



Figure 41: Aircraft design flow: A portion of the steps and products generated in the modification of control surfaces and control system design, as part of the overall aircraft design.

[40] 6 Degrees of Freedom simulation.

# D NASA Regulatory Requirements

NASA's regulatory environment consists of *NASA procedural requirements* (NPRs), some of which (though not all) reference *NASA standards*, and possibly guide books. NPRs are available at the NASA Online Directives Information System (NODIS) (http://nodis3.gsfc.nasa.gov/main_lib.html). Standards and guidance documents are obtained from the NASA standards website (https://standards.nasa.gov/). Relevant documents and websites for this project include:

- NPR 7150.2A - NASA Software Engineering Requirements
  http://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002A_&page_name=main&search_term=NPR%207150%2E2A
- NPR 7123.1A - NASA Systems Engineering Processes and Requirements
  http://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7123_001A_&page_name=Preface&search_term=NPR%207123%2E1A
- NPR 7900.3B - Aircraft Operations Management Manual
  http://www.hq.nasa.gov/office/codeq/doctree/79003.htm
- NASA-GB-8719.13 - NASA Software Safety Guidebook
  https://standards.nasa.gov/documents/detail/3315126
- NASA-STD-8719.13 - NASA Software Safety Standard
  https://standards.nasa.gov/documents/detail/3314914
- NASA-STD-8739.8 - Software Assurance Standard
  https://standards.nasa.gov/documents/detail/3315130
- NASA/SP-2009-569 - Bayesian Inference for NASA Probabilistic Risk and Reliability Analysis
  https://standards.nasa.gov/documents/detail/3315758
- NPR 8715.5A - NASA Range Flight Safety Program
  http://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_8715_005A_&page_name=main&search_term=8715%2E5
- NASA Range Safety
  http://kscsma.ksc.nasa.gov/Range_Safety/Overview.html
- Range safety documents
  http://kscsma.ksc.nasa.gov/Range_Safety/NASALinks.html
- NPR 8715.5A: Appendix A (Range safety definitions: CMS, etc.)
  http://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_8715_005A_&page_name=AppendixA
- NPR 8705.5A: Probabilistic Risk Assessment (PRA) Procedures for Safety and Mission Success for NASA Programs and Projects
  http://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_8705_005A_&page_name=main
- NPR 8715.3C: (System Safety) NASA General Safety Program Requirements
  http://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_8715_003C_&page_name=main

There are also Ames Procedural Requirements:

- APR 8705.1: System Safety and Mission Assurance
  http://server-mpo.arc.nasa.gov/Services/CDMSDocs/Centers/ARC/Dirs/APR/APR8705.1.html

# E Software Certification Overview

Here, some implications of the existing practice of process-based software safety assurance on the safety assurance activities for the Swift UAS, are described.[41]

## Categories and Software Levels

Some of the hazards identified in Table 3 may have a root cause in a software function and can only be mitigated by verifying the correct operation of the underlying software. For example, a race condition between two critical autopilot control tasks may lead to a fault and loss of control of the autopilot. This race condition may have been rooted in incorrect design of the underlying operating system. Therefore, it is essential that all software components be properly verified to ensure the safety case claims can be supported. For this reason, it is recommended that RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, published December 1, 1992 be used as a guidance document to ensure all claims related to software have sufficient evidence to support these claims. The evidence to support DO-178B certification can be extensive and there are many approaches to producing this evidence. One potential approach is discussed below.

Any latent defects in the Swift autopilot system whose anomalous behavior, as shown by the system safety assessment, would cause or contribute to a catastrophic failure of the aircraft means that the software is considered safety-critical, Level A according to guidance in RTCA/DO-178B.

Table 18 provides the relevant definitions of the appropriate software level and associated failure condition:

Table 18: Software certification levels

| **Software Certification Level or Failure Condition** | **Definition** |
|---|---|
| Level A | Software whose anomalous behavior, as shown by the system safety assessment, would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft. |
| Catastrophic | Failure conditions that would prevent continued safe flight and landing |

All software operating on the Swift UAS platform would be required to meet DO-178B, Level A unless the system were partitioned to contain faults. Partitioning and fault containment are beyond the scope of this safety case. The following components of software operate on the Swift UAS platform and can impact the safety of the vehicle, including:

- Autopilot
- FMS
- CGL
- Windows XP Embedded Operating system
- Script Files
- Reflection Virtual Machine
- Board-specific software (e.g, BIOS, PCI, etc.) and device drivers
- INS/GPS software (and associated operating system, drivers and board support software
- 900 MHz Radio Software (if software is used)

The possible failure conditions that may lead to application software malfunctioning within a given partition may be summarized as follows:

- Incorrect results are provided to the application by the operating system.
- Expected results are not provided, or are provided past their deadlines.

[41] The content of this section has been provided by FAA DER, Joe Wlad.

- Application code is not executed as expected (not run, incorrectly run, or incorrectly sequenced).
- Fault conditions are not detected or handled incorrectly.
- Data is incorrectly modified
- Timing errors
- Priority inversion
- Deadlock conditions
- Incorrect or untimely response provided by the OS to external or user generated events
- Failure of operating system or board support software to ensure resources are available as required for a given application

Table 19 provides a high-level description of DO-178B certification activities which would be employed to demonstrate compliance with DO-178B, Level A objectives. These processes would apply to all software components used on the Swift UAS.

Table 19: Software certification activities

| **RTCA/DO-178B Processes** | **Description** |
|---|---|
| Planning | Defining the Certification Strategy and Planning documents, to include the Plan for Software Aspects of Certification (PSAC), Software Development Plan (SDP), Software Verification Plan (SVP) and appropriate standards. |
| Configuration Management | The process by which the software and software lifecycle data will be controlled and managed, including problem reports and software updates |
| Quality Assurance | Assurance of Independence and enforcement of software lifecycle processes and activities |
| Requirements | Definition and review of High and low-level software requirements |
| Design | Definition and review of Software Design |
| Coding | Definition and review of Software source and object code. |
| Integration | Cohesion of software modules into one or more functional components |
| Verification | Reviews of requirements, design, source and object code as well as test plans, procedures, results and coverage analysis |
| Documentation | Production of all required DO-178B documentation to support Level A certification |
| Certification Liaison | Relationship with the certification authority and if required approval of PSAC, SAS and SCI documents |

Since much of the software used on the Swift UAS is considered "PDS" or previously developed software, one would need to undertake a reverse engineering process to reconstruct all the required DO-178B documentation and activities. Guidance exists in the form of a Federal Aviation Administration Position Paper (CAST-18: Reverse Engineering in Certification Projects)[42], which can be used to ensure this effort is done with as little certification risk as possible. Figure 42 gives an abstract overview of this process.

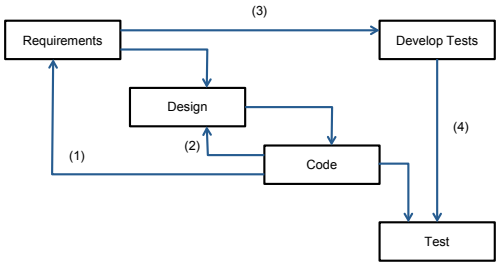



Figure 42: Activities for reverse engineering DO-178B documentation for previously developed software.

[42] http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/

1. The first step is to examine the entire code structure and reverse engineer high-level and low-level requirements.
2. Thereafter, reverse engineer design data (either in the form of pseudo-code or similar documentation). Then the requirements and design data would be traced together and linked with the source code itself. Convenient tools such as DOORS would be use to accomplish this task.
3. Step 3 is to produce the test cases from the requirements (not from the design or source code).
4. These requirements-based tests are executed in Step 4 and then the coverage of the source and object code would be analyzed against DO-178B, level A requirements. This process would be repeated until complete source and object coverage is achieved.

The documentation required to support DO-178B certification evidence includes, at a minimum, the products shown in Table 20:

There are other certification considerations that may be applicable to the software on the Swift UAS platform, such as use of non-deterministic functions (such as freeing of memory, use of Direct Access Memory control, among other), use of object oriented design (C++, Java), ability to qualify tools used in the verification process, among others but these are considered out of scope for this phase of the safety case.

Table 20: Software certification evidence

| **Product Name** | **Description** |
|---|---|
| Plan for Software Aspects of Certification | Provides the Certification Authorities an overview of the means of compliance and insight into the planning aspects for delivery of the product. |
| Software Quality Assurance Plan | Defines the SQA process and activities. |
| Software Configuration Management Plan | Defines the CM system and change control process. |
| Software Development Plan | Define the processes used for requirements analysis, development, and test for the software product. Include the standards for requirements, design, and code. |
| Software Verification Plan<br><br>Software Requirements Standard<br>Software Design Standard<br>Software Coding Standard | Defines the test philosophy, test methods and approach to be used to verify the software product. |
| Software Test Plan | Documents the project specific approach to verifying the software product. |
| Software Requirements Specification | Defines the high-level requirements applicable to the air data computer software |
| Software Design Document | Describes the design of the certifiable air data computer software. |
| Software Configuration Index | Identifies the components of the certifiable air data computer software with version information necessary to support regeneration of the product. |
| Software Configuration Index | Identifies the components of the air data computer software with version information necessary to support regeneration of the air data computer software. |
| Software Life Cycle Environment Configuration Index | Identifies the tools used to build and test air data computer software. |
| Software Development information | Software Development information includes as a minimum:<br>• Reference to the applicable requirements<br>• Reference to the implementation (Design & Code)<br>• Evidence of reviews for the Requirements, Design, Code, and Test procedures and test results<br>• Software Test Procedures<br>• Software Test Results<br>• Analysis documents for verification, coverage analysis, and any special case analysis.<br>• Change History (CM System)<br>• Applicable Problem Reports |
| Traceability Matrix | Provides traceability from the requirements, to the built software, to test for the delivered software product. |
| Software Accomplishment Summary | Documents the actual versus planned (wrt PSAC) activities and results for the project. Provides a summary of the means of compliance used for the software. Justifies any deviations from the plans. |
| Results | Documents the results of the functional and structural coverage testing. This includes the actual results and any applicable analyses performed including coverage analysis. |