NASA/TM–2012–216481

Hierarchical Safety Cases

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

Iain J. Whiteside
Centre for Intelligent Systems and their Applications
University of Edinburgh, Scotland

December 2012

NASA/TM–2012–216481

Hierarchical Safety Cases

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

Iain J. Whiteside
Centre for Intelligent Systems and their Applications
University of Edinburgh, Scotland

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

December 2012

Acknowledgments

This work was carried out by Iain Whiteside at the NASA Ames Research Center during an internship funded by the AFCS element of the SSAT project in the Aviation Safety Program of the NASA Aeronautics Mission Directorate.

Available from:

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

Abstract

We introduce *hierarchical safety cases* (or hicases) as a technique to overcome some of the difficulties that arise creating and maintaining industrial-size safety cases. Our approach extends the existing Goal Structuring Notation with abstraction structures, which allow the safety case to be viewed at different levels of detail. We motivate hicases and give a mathematical account of them as well as an intuition, comparing them to other related concepts. We give a second definition which corresponds closely to our implementation of hicases in the AdvoCATE Assurance Case Editor and prove the correspondence between the two. Finally, we suggest areas of future enhancement, both theoretically and practically.

Contents

List of Figures

1 Introduction

A *safety case*, or more generally an assurance case, is a structured argument, supported by a body of evidence, which provides a convincing and valid justification that a system is acceptably safe (or assured) for a given application in a given operating environment. The development of a safety case has become common practice for the certification of safety-critical systems in the nuclear, defense, oil and gas, and rail domains. Indeed, the development and acceptance of a safety case is a key element of safety regulation in many safety-critical sectors. The Goal Structuring Notation (GSN) is emerging as the de facto representation for the argument structure of a safety case, representing safety cases using a "boxes and arrows" approach. Until recently, safety cases were typically constructed manually, but tools are now emerging to assist (and sometimes automate) construction of safety cases. The AdvoCATE Assurance Case Editor, developed here at NASA Ames, is one such tool [5].

Figure 1 shows a small fragment of a safety case for the Swift Unmanned Aircraft System, which is being developed at NASA Ames [4], as represented in AdvoCATE. The argument proceeds top-down from the high-level goal that the Swift UAS is safe. All facets of the argument are explicit, including the *assumptions* made about a *goal* and the *contextual* information necessary to make sense of the goal. For example this safety case only justifies safety of the Swift given appropriate weather conditions and within its defined range of operation. So-called *strategies* are used to break down high-level goals into simpler subgoals. Once these subgoals are simple enough, they are solved by *evidence*: references to artifacts that guarantee the property.

Necessarily, such diagrams become very large indeed. As an anecdotal example, a typical safety case for a medium-size North Sea production platform covers anywhere from 490–660 pages [10]. This makes them difficult to develop, evaluate (or understand), and maintain. Often, the safety case is constructed using design patterns or has some natural higher-level structure that is clear to the author (if only at the time of writing), but can become obscured by the detail so it is "hard to see the wood for the trees". It is also becoming feasible for some parts of safety cases to be constructed automatically from external tools such as a theorem prover or sets of hazard and requirement tables [3]. Such safety case fragments often have inherent structure that could and should be exploited to help comprehension.

Based on these observations, we propose to extend the GSN notation (which we will simply refer to as safety cases from now on) to include hierarchical structuring mechanisms. We call the these structures *hicases*, and claim that they help to clarify the structure of a safety case and improve the quality of the argument. The main contributions of this work are:

1. A theoretical description of GSN safety cases and an extension to this model for hierarchical safety cases. In fact, we give two equivalent definitions: one closer to an implementation.

2. We relate an unfolding of the hicases to an ordinary safety case by means of a *skeleton* operation as well as providing a natural embedding of ordinary safety cases in hicases.

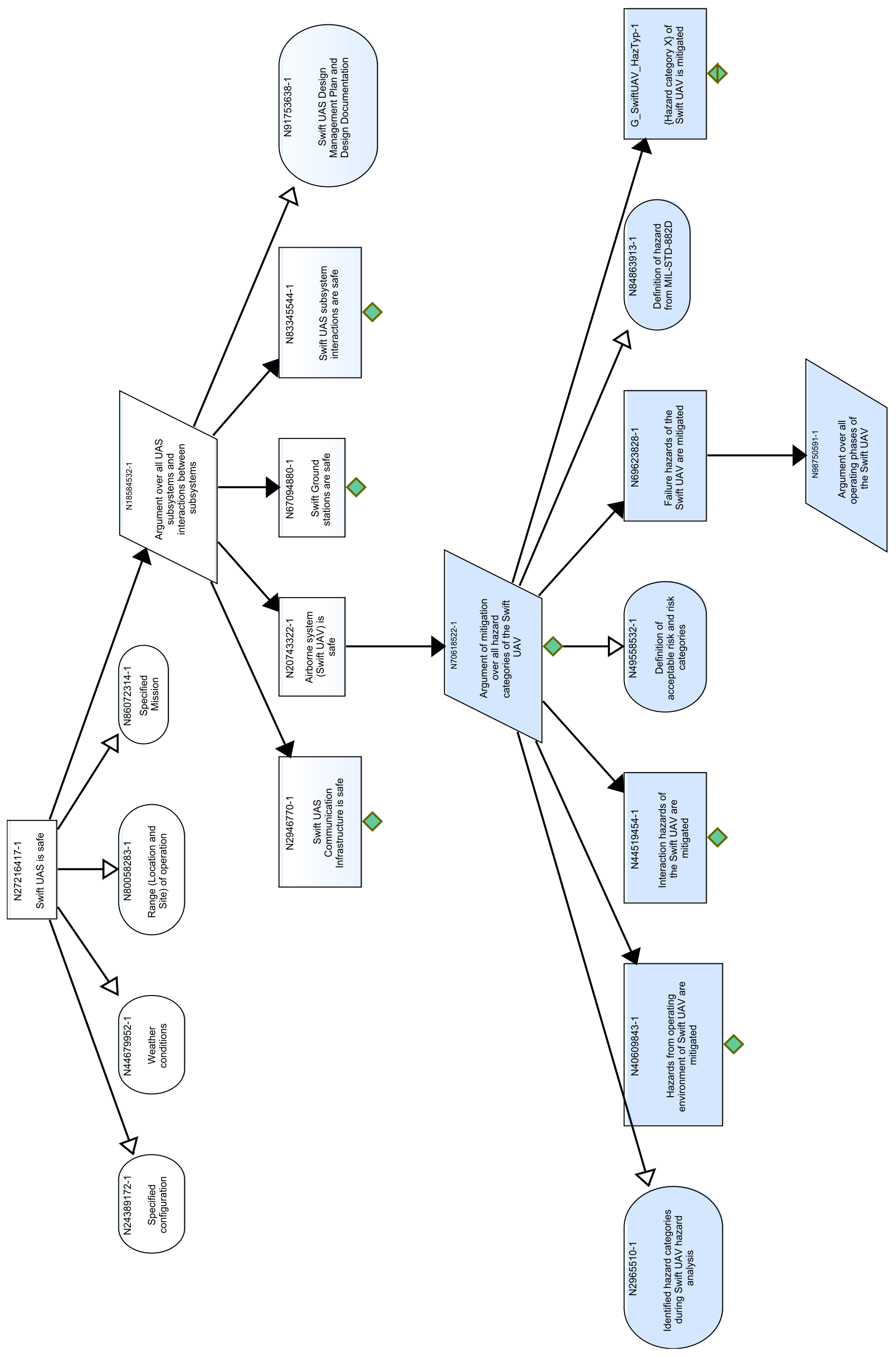

N27216417-1
Swift UAS is safe
N86072314-1
Specified Mission
N80058283-1
Range (Location and Site) of operation
N44679952-1
Weather conditions
N24389172-1
Specified configuration
N18584532-1
Argument over all UAS subsystems and interactions between subsystems
N91753638-1
Swift UAS Design Management Plan and Design Documentation
N83345544-1
Swift UAS subsystem interactions are safe
N67094880-1
Swift Ground stations are safe
N20743322-1
Airborne system (Swift UAV) is safe
N2946770-1
Swift UAS Communication Infrastructure is safe
N70618522-1
Argument of mitigation over all hazard categories of the Swift UAV
G_SwiftUAV_HazTyp-1
{Hazard category X} of Swift UAV is mitigated
N84863913-1
Definition of hazard from MIL-STD-882D
N69623828-1
Failure hazards of the Swift UAV are mitigated
N98750591-1
Argument over all operating phases of the Swift UAV
N49558532-1
Definiton of acceptable risk and risk categories
N44519454-1
Interaction hazards of the Swift UAV are mitigated
N40609843-1
Hazards from operating environment of Swift UAV are mitigated
N2965510-1
Identified hazard categories during Swift UAV hazard analysis

Figure 1: A safety case fragment for the Swift UAV

3. An implementation of hicases in the AdvoCATE tool, which we use to provide the hicase examples in this work.

The rest of this report is structured as follows. In Section 2, we introduce the GSN notation for safety cases in more detail. Then, in Section 3 we give motivating examples of hierarchy based on the Swift safety case. Then, in Section 4 we give a formal definition capturing the notion of safety case and extend it to hierarchical safety cases. We provide alternative definitions in Section 5 and describe our implementation in Section 6. Finally, we conclude with related and future work in Section 7.

2 Goal Structuring Notation

The Goal Structuring Notation for safety cases, defined in [1], is fast becoming a standard model for argumentation of safety cases. In this section, we give a brief overview of GSN, but for a full description, we refer the reader to the standard [1].

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) [1] for documentation. The elements of the GSN are shown in Figure 2. 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[1] 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 2. 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.

A safety case will always be rooted with a top-level goal: most often that the system is safe, but it can be useful to relax this condition and consider *partial* safety cases: which are sets of safety cases, that still need to be connected and tied down to evidence. We can draw analogies between GSN diagrams and proof trees. Strategies can be interpreted as tactics or inference rules; evidence nodes can been seen as axiomatic strategies[2]. The context, assumption, and justification nodes can really be seen as attributes to goals and strategies. Also, the core GSN standard is very flexible: it doesn't

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

[2]The analogy is not complete. Goals in safety cases can, for example, have multiple strategies solving them independently for *extra assurance.*

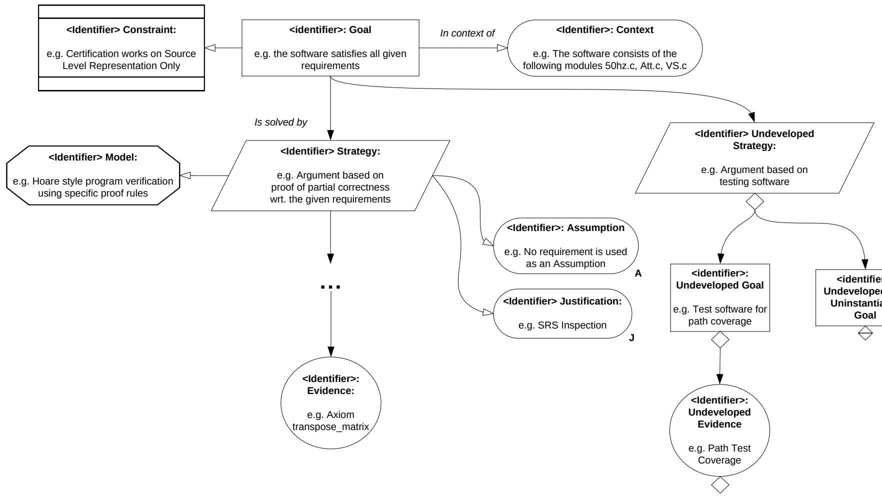

<Identifier> Constraint:
e.g. Certification works on Source Level Representation Only
<identifier>: Goal
e.g. the software satisfies all given requirements
In context of
<Identifier>: Context
e.g. The software consists of the following modules 50hz.c, Att.c, VS.c
Is solved by
<Identifier> Model:
e.g. Hoare style program verification using specific proof rules
<Identifier> Strategy:
e.g. Argument based on proof of partial correctness wrt. the given requirements
<Identifier> Undeveloped Strategy:
e.g. Argument based on testing software
<Identifier>: Assumption
e.g. No requirement is used as an Assumption
A
<Identifier> Justification:
e.g. SRS Inspection
J
<identifier>: Undeveloped Goal
e.g. Test software for path coverage
<identifier>: Undeveloped and Uninstantiated Goal
<Identifier>: Evidence:
e.g. Axiom transpose_matrix
<Identifier>: Undeveloped Evidence
e.g. Path Test Coverage

Figure 2: Syntax of the GSN notation

prescribe many syntactic restrictions and no semantics. In this work, though, we utilise our definitions to restrict the types of safety case that can be constructed to those that are sensible. The core GSN standard is then extended with the basic infrastructure for *modules*. Safety case modules are similar to their traditional programming language counterparts and facilitate data-reuse as well as some degree of abstraction. Modules are compared with our approach in Section 7.1.

3 Examples of Hierarchy

In this section, we present several examples of hierarchical safety cases. Our examples are derived from the Swift safety case, which we have studied to find motivating instances of 'abstractable structure', which we represent using hierarchical nodes (or hinodes).

3.1 Abstract Evidence

Example 3.1 (Abstract Evidence)**.** Consider the segment of a safety case taken from the automatically generated part of the Swift safety case given in Figure 3. This fragment of the safety case is generated using the AutoCert tool [7], and the segment shown represents a *direct* proof of a verification condition for a software module in Swift. A lot of the details of the proof are transformed into the safety case, such as the theorem prover used, the name of the proof object etc, and we may abstract away from this in the hierarchical presentation. A **hierarchical evidence** node can be constructed. *Inside* it is the subproof rooted at the AC10 strategy: *argument by proof using automated theorem provers*. Since this sub-argument is complete — that is, it has no remaining goals — it could be understood abstractly as an evidence node. The hierarchical representation in AdvoCATE is shown in Figure 4 with node *H1*. We call this the *open* view of a hinode. We can view it as a *black box* — in the *closed* view — as shown in Figure 5 where the advantages of abstraction become clear: we have simplified the representation of the safety case with this presentation. As there are many verification conditions, we have many instances of this structure; furthermore, we iterate this procedure up to proof tree, offering opportunities for nested hierarchy. Iterated abstraction could then greatly reduce the size of the safety case when viewed.

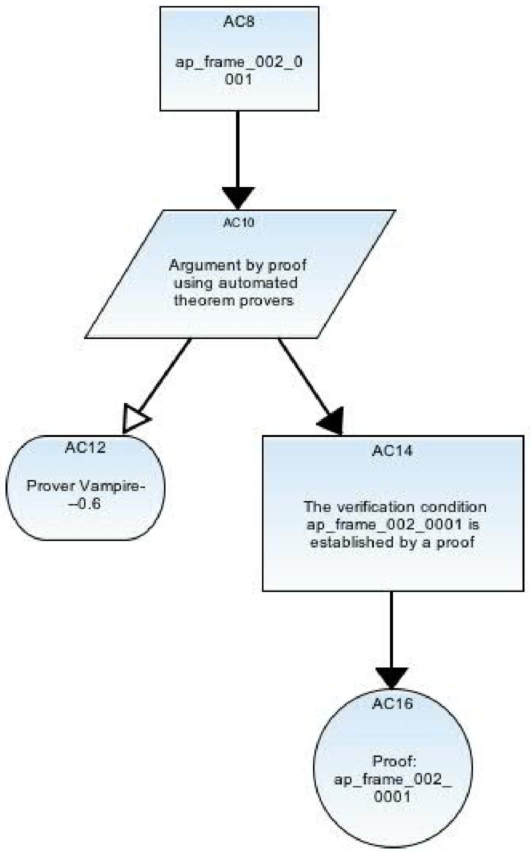

AC8
ap_frame_002_0001
AC10
Argument by proof using automated theorem provers
AC12
Prover Vampire-–0.6
AC14
The verification condition ap_frame_002_0001 is established by a proof
AC16
Proof: ap_frame_002_0001

Figure 3: A fragment of an auto-generated safety case

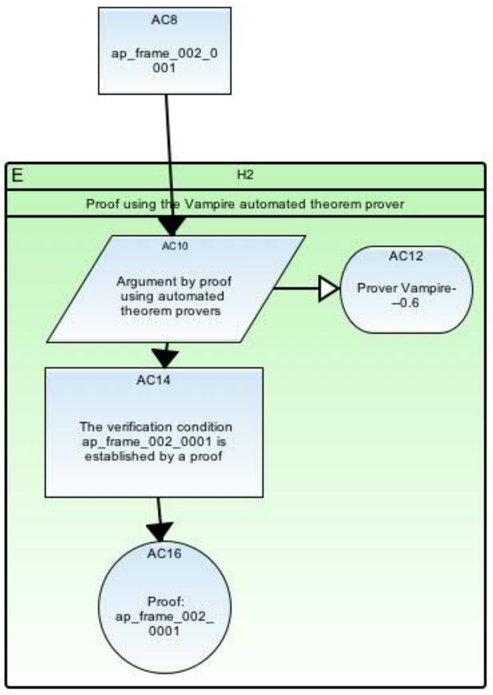

AC8
ap_frame_002_0001
E
H2
Proof using the Vampire automated theorem prover
AC10
Argument by proof using automated theorem provers
AC12
Prover Vampire-–0.6
AC14
The verification condition ap_frame_002_0001 is established by a proof
AC16
Proof: ap_frame_002_0001

Figure 4: Constructing hierarchy as a new abstract evidence node

3.2 Abstract Strategies

If, in the example above, we'd chosen a slice through a safety case where some paths were not fully enclosed (or that is not *fully developed*), we would have an **abstract strategy** node. The paths not fully embraced within the hinode can then be considered as the subgoals of this node. A typical use of an abstract strategy is to group together a meaningful chain of strategy applications. In an analogy with tactical theorem proving, a composite tactic like *INTROS*, which applies as many introduction rules to a goal as possible, could be seen as a hierarchical strategy.

Example 3.2 (Abstract Strategies 1)**.** Consider the safety case fragment (from the middle of the Swift safety case) in Figure 6. It represents an argument about correctness of the software system during the descent phase; however, the two strategies arguing this can be grouped together as a single strategy: *'Correctness argument over architectural breakdown'*. We can represent this as an abstract strategy, which is shown (in the open view) in Figure 7. As with abstract evidence, we can also *close* this node giving us the safety case in Figure 8. It is important to note that the context, assumption, and justification nodes are also enclosed within the hierarchical strategy. It would also be feasible to exclude them (and connect to hinodes), but we decided that the hinodes

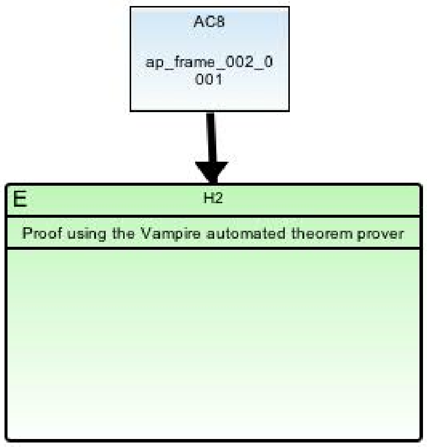


Figure 5: Viewing the abstract evidence node as a *black-box* node

would be too cluttered in this view, although we could off this as an option in future implementations.

Example 3.3 (Abstract Strategies 2)**.** Abstract strategies can also be employed to hide side-conditions (or trivial subgoals), by fully enclosing particular paths of the safety case. This gives the developer (or viewer) the flexibility to only concentrate on the *important* paths through the safety case during a cursory viewing. If required, the hierarchical strategies could be opened to view the full detail.

Figure 9 shows a fragment of the Swift safety case, which argues the safety of the auto-pilot design. The goals N61592954 and N7927263 have been considered by the developer to be unimportant and, indeed, their justifications have already been abstracted and closed. We wish to abstract this safety case to:

- Hide the unimportant goals within a hierarchical strategy.
- Also group the Strategy S2 to leave us with one outgoing subgoal: 'the specification for computing angle of attack is correct'.

The resulting hicase gives us our first instance of nested hierarchy and is seen in Figure 10. The advantage of hierarchy for managing the size of safety cases can be seen when this hierarchical strategy is closed, as in Figure 11. The output goal, G3, is solved with a deep tree (of which only the first level is shown here) which provides some evidence for its relative importance compared with the other subgoals.

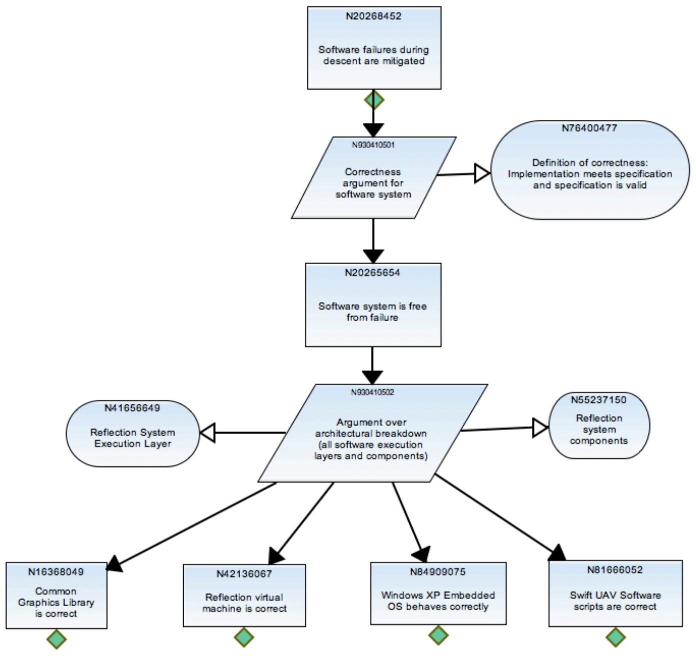

N20268452
Software failures during descent are mitigated
N930410501
Correctness argument for software system
N76400477
Definition of correctness: Implementation meets specification and specification is valid
N20265654
Software system is free from failure
N930410502
Argument over architectural breakdown (all software execution layers and components)
N41656649
Reflection System Execution Layer
N55237150
Reflection system components
N16368049
Common Graphics Library is correct
N42136067
Reflection virtual machine is correct
N84909075
Windows XP Embedded OS behaves correctly
N81666052
Swift UAV Software scripts are correct

Figure 6: A safety case fragment, where we can group two strategy applications

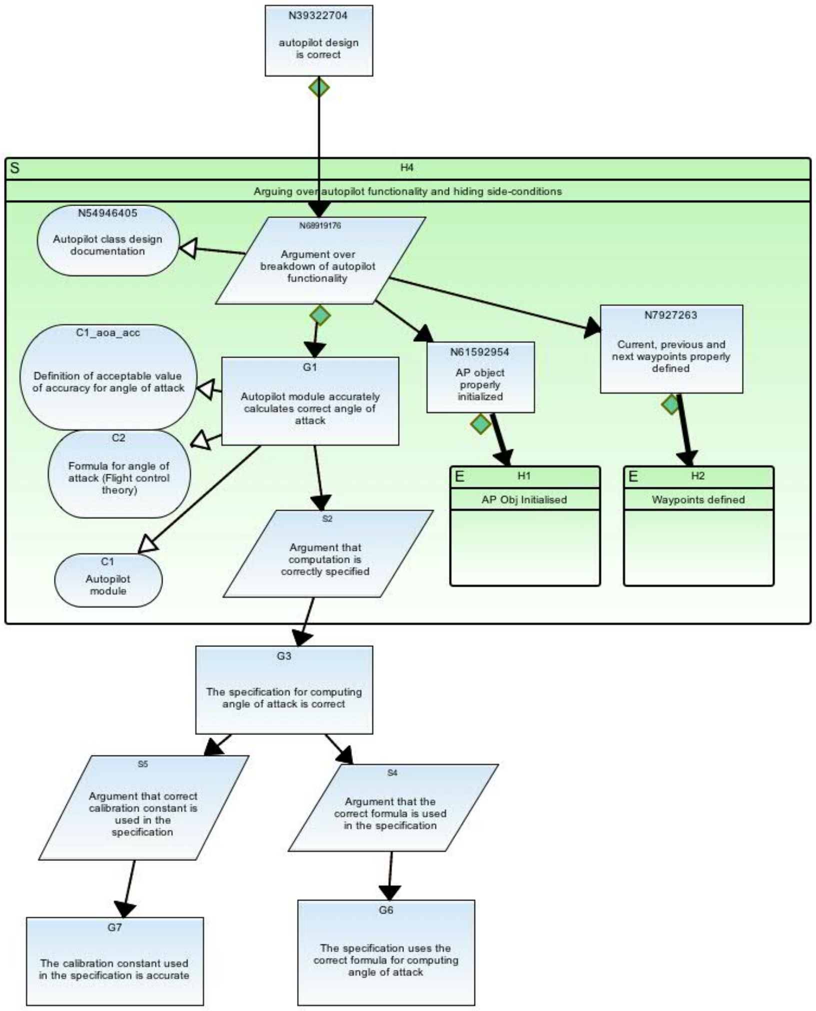

N39322704
autopilot design is correct
S
H4
Arguing over autopilot functionality and hiding side-conditions
N54946405
Autopilot class design documentation
N68919176
Argument over breakdown of autopilot functionality
C1_aoa_acc
Definition of acceptable value of accuracy for angle of attack
G1
Autopilot module accurately calculates correct angle of attack
N61592954
AP object properly initialized
N7927263
Current, previous and next waypoints properly defined
C2
Formula for angle of attack (Flight control theory)
E
H1
AP Obj Initialised
E
H2
Waypoints defined
S2
Argument that computation is correctly specified
C1
Autopilot module
G3
The specification for computing angle of attack is correct
S5
Argument that correct calibration constant is used in the specification
S4
Argument that the correct formula is used in the specification
G7
The calibration constant used in the specification is accurate
G6
The specification uses the correct formula for computing angle of attack

Figure 7: A hierarchical strategy in the Swift safety case

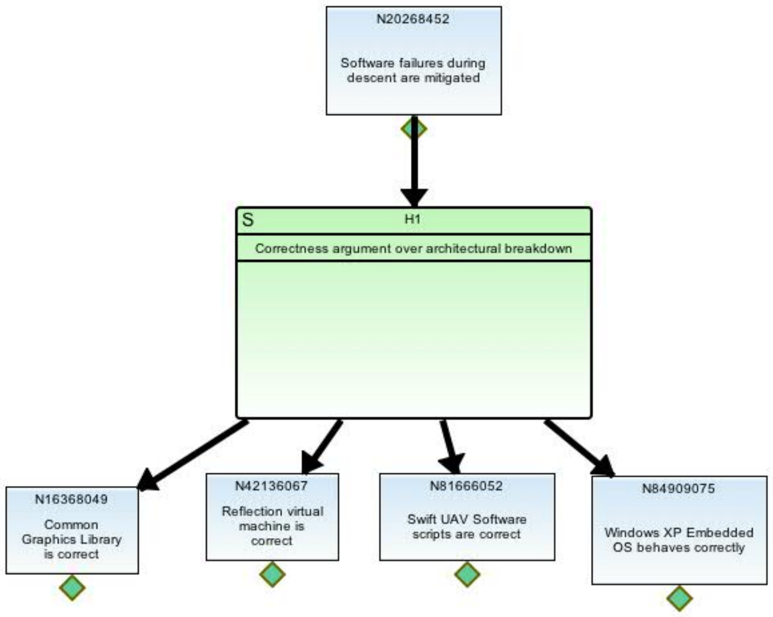

N20268452
Software failures during descent are mitigated
S
H1
Correctness argument over architectural breakdown
N16368049
Common Graphics Library is correct
N42136067
Reflection virtual machine is correct
N81666052
Swift UAV Software scripts are correct
N84909075
Windows XP Embedded OS behaves correctly

Figure 8: A hierarchical strategy in the Swift safety case, viewed as a *blackbox*

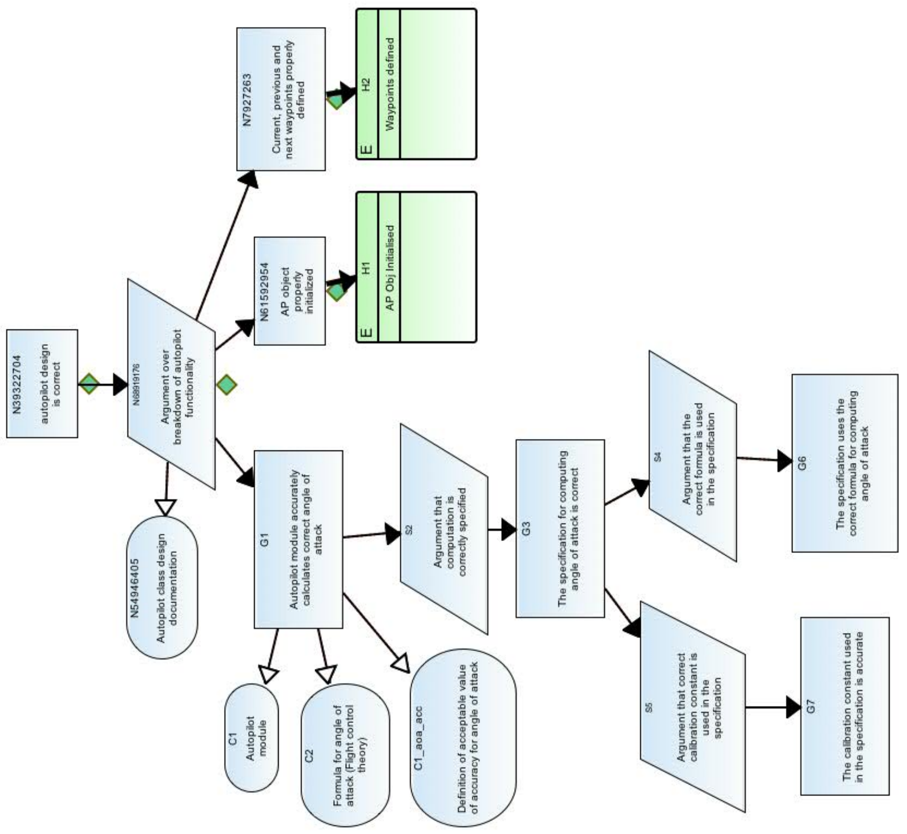

N39322704
autopilot design is correct
N6919176
Argument over breakdown of autopilot functionality
N61592954
AP object properly initialized
N7927263
Current, previous and next waypoints properly defined
E
H1
AP Obj Initialised
E
H2
Waypoints defined
N54946405
Autopilot class design documentation
G1
Autopilot module accurately calculates correct angle of attack
C1
Autopilot module
C2
Formula for angle of attack (Flight control theory)
C1_aoa_acc
Definition of acceptable value of accuracy for angle of attack
S2
Argument that computation is correctly specified
G3
The specification for computing angle of attack is correct
S4
Argument that the correct formula is used in the specification
S5
Argument that correct calibration constant is used in the specification
G6
The specification uses the correct formula for computing angle of attack
G7
The calibration constant used in the specification is accurate

Figure 9: An argument with two trivial subgoals ('AP object properly initialised' and 'Current, previous, and next waypoints properly defined'), which can be hidden

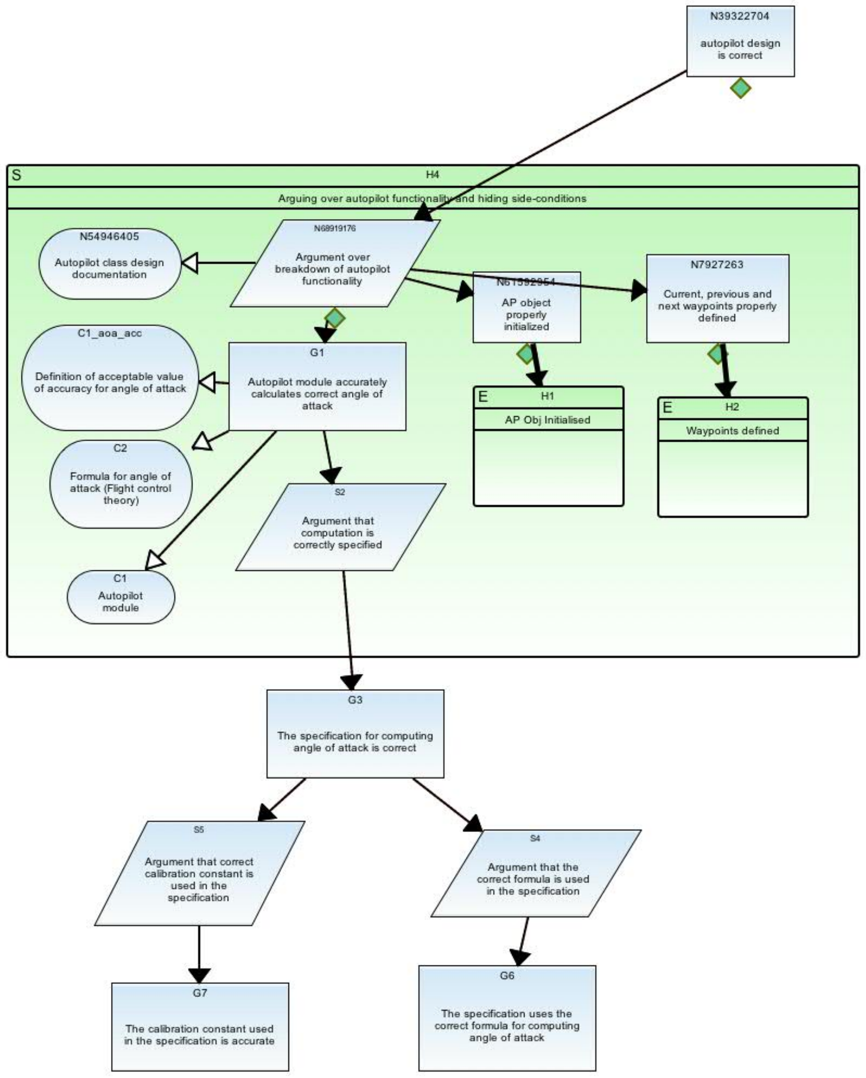

N39322704
autopilot design is correct
S
H4
Arguing over autopilot functionality and hiding side-conditions
N54946405
Autopilot class design documentation
N68919176
Argument over breakdown of autopilot functionality
N61592954
AP object properly initialized
N7927263
Current, previous and next waypoints properly defined
C1_aoa_acc
Definition of acceptable value of accuracy for angle of attack
G1
Autopilot module accurately calculates correct angle of attack
E
H1
AP Obj Initialised
E
H2
Waypoints defined
C2
Formula for angle of attack (Flight control theory)
S2
Argument that computation is correctly specified
C1
Autopilot module
G3
The specification for computing angle of attack is correct
S5
Argument that correct calibration constant is used in the specification
S4
Argument that the correct formula is used in the specification
G7
The calibration constant used in the specification is accurate
G6
The specification uses the correct formula for computing angle of attack

Figure 10: ‘Boxing up’ the trivial subgoals

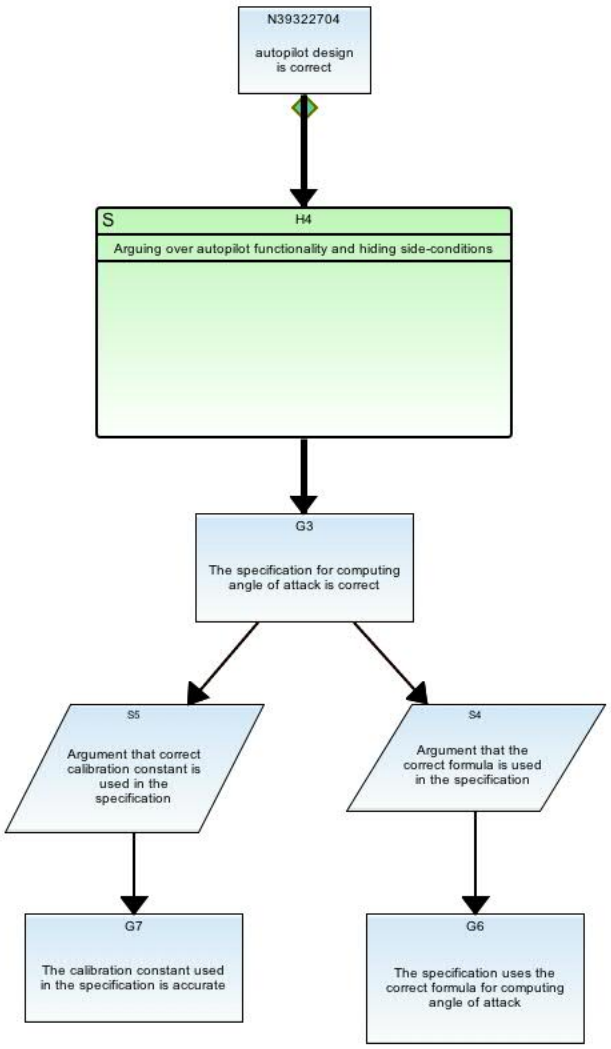

N39322704
autopilot design is correct
S
H4
Arguing over autopilot functionality and hiding side-conditions
G3
The specification for computing angle of attack is correct
S5
Argument that correct calibration constant is used in the specification
S4
Argument that the correct formula is used in the specification
G7
The calibration constant used in the specification is accurate
G6
The specification uses the correct formula for computing angle of attack

Figure 11: When the hierarchical strategy is closed, the trivial subgoals are hidden

3.3 Abstract Goals

By symmetry of strategies and goals, we have the dual notion of an *abstract goal*. In our studies of real-life safety cases, this construct is not as common as abstract strategies, but nevertheless it can be used to provide a high-level view of the safety case. As a concrete example, consider this hierarchical ‘chunking’ of a safety case derived automatically from hazard and requirements tables, Figure 12. We utilise hierarchical goals to group the functional requirements.

3.4 Case Studies

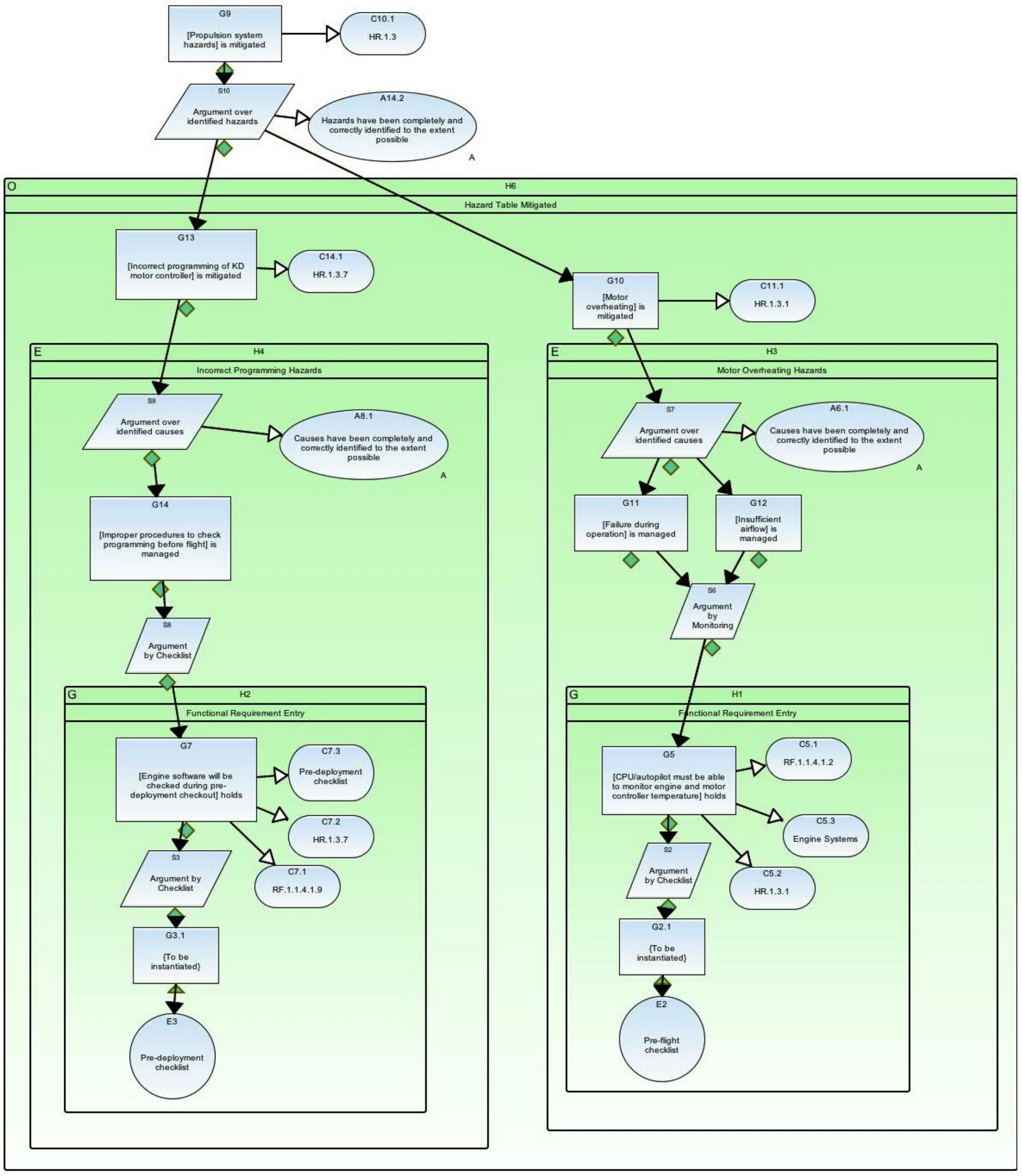
We have also performed case studies introducing structure on the full Swift safety case. In particular, we have utilised hinodes to make clear the structure of a detailed proof generated by AutoCert and make the argument structure of the Swift safety case explicitly using hinodes. We describe both in the following sections.

3.4.1 Aileron Correctness Proof

The AutoCert tool produced a very detailed proof of the correctness property of the aileron control sequence in the autopilot. The proof proceeds by proving correct the individual computations that are performed in controlling the aileron. The computation structure is shown in Figure 13. Previously, the proof was very detailed and unreadable for humans. However, we were able to use hinodes to hide the trivial side-conditions and make clear the architecture of the proof. We show the top fragment of the resulting hicase in Figure 14. The goal $AC1$ represents proving the correctness of the Aileron control computed variable in Figure 13. This is because the AutoCert proof is a backwards proof common in automated theorem proving, where the stated goal is broken into simpler sub-goals. The hierarchical strategy $H16$ performs this decomposition, leaving a single output goal $AC98$, which states that we have to prove the desired roll variable is correct (the next goal up in the computation structure) and so the high-level proof continues with the next goal corresponding to the desired heading variable. For an expert familiar with the system, the hierarchical safety case makes it clearer that this sequence of computations is correct.

3.4.2 Swift High-level Structure

If we look at Figure 15, we see that it has a natural high-level structure. First the argument proceeds by looking at the various sub-systems of Swift: the ground station, the aircraft etc. Then, by arguing over the *phases of operation*, such as Taxi, Cruise, Descent etc. For large safety cases, this inherent structure can become almost completely obscured by the detail in the safety case; however, we have experimented with hicases and discovered that it can be made clear through the addition of hinodes. As an example, Figure 16 shows the hierarchical safety case that we constructed to show the high-level structure. The hierarchical evidence node, $H56$, details the proof technique and encloses the rest of the safety case. Then, each of $H3$–$H6$, represent the hierarchical evidence nodes that contain the proofs of safety for each individual system. The hierarchical structure keeps the high-level structure visible.



G9
[Propulsion system hazards] is mitigated
C10.1
HR.1.3
S10
Argument over identified hazards
A14.2
Hazards have been completely and correctly identified to the extent possible
A
O
H6
Hazard Table Mitigated
G13
[Incorrect programming of KD motor controller] is mitigated
C14.1
HR.1.3.7
G10
[Motor overheating] is mitigated
C11.1
HR.1.3.1
E
H4
Incorrect Programming Hazards
E
H3
Motor Overheating Hazards
S9
Argument over identified causes
A8.1
Causes have been completely and correctly identified to the extent possible
A
S7
Argument over identified causes
A6.1
Causes have been completely and correctly identified to the extent possible
A
G14
[Improper procedures to check programming before flight] is managed
G11
[Failure during operation] is managed
G12
[Insufficient airflow] is managed
S8
Argument by Checklist
S6
Argument by Monitoring
G
H2
Functional Requirement Entry
G
H1
Functional Requirement Entry
G7
[Engine software will be checked during pre-deployment checkout] holds
C7.3
Pre-deployment checklist
C7.2
HR.1.3.7
C7.1
RF.1.1.4.1.9
G5
[CPU/autopilot must be able to monitor engine and motor controller temperature] holds
C5.1
RF.1.1.4.1.2
C5.3
Engine Systems
C5.2
HR.1.3.1
S3
Argument by Checklist
S2
Argument by Checklist
G3.1
{To be instantiated}
G2.1
{To be instantiated}
E3
Pre-deployment checklist
E2
Pre-flight checklist

Figure 12: Hierarchical groupings for hazard table construction

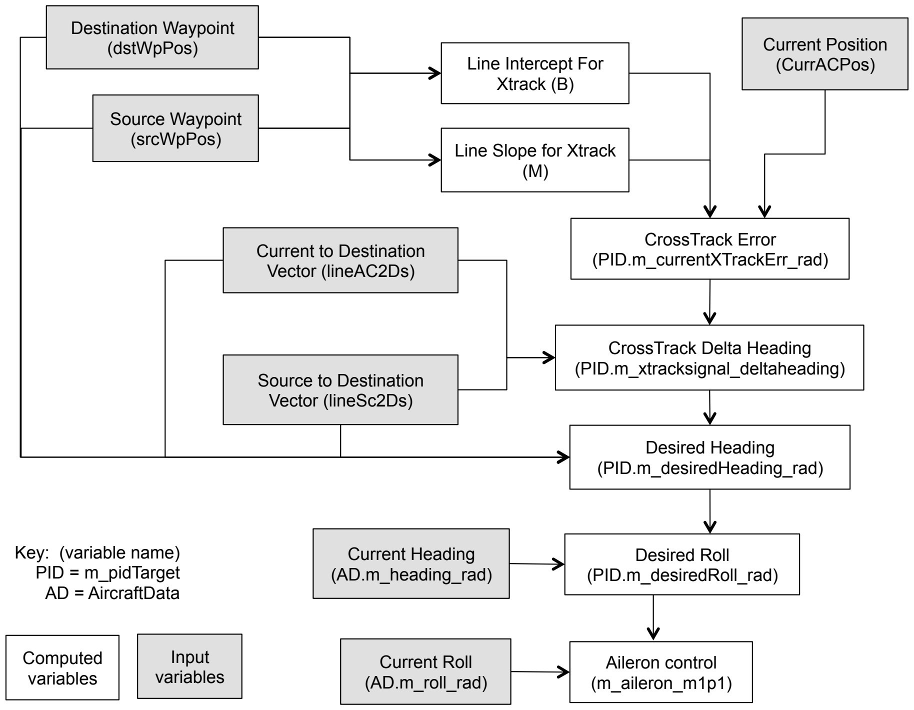

Destination Waypoint (dstWpPos)
Source Waypoint (srcWpPos)
Line Intercept For Xtrack (B)
Line Slope for Xtrack (M)
Current Position (CurrACPos)
CrossTrack Error (PID.m_currentXTrackErr_rad)
Current to Destination Vector (lineAC2Ds)
Source to Destination Vector (lineSc2Ds)
CrossTrack Delta Heading (PID.m_xtracksignal_deltaheading)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

Figure 13: Aileron computation structure (white portions)

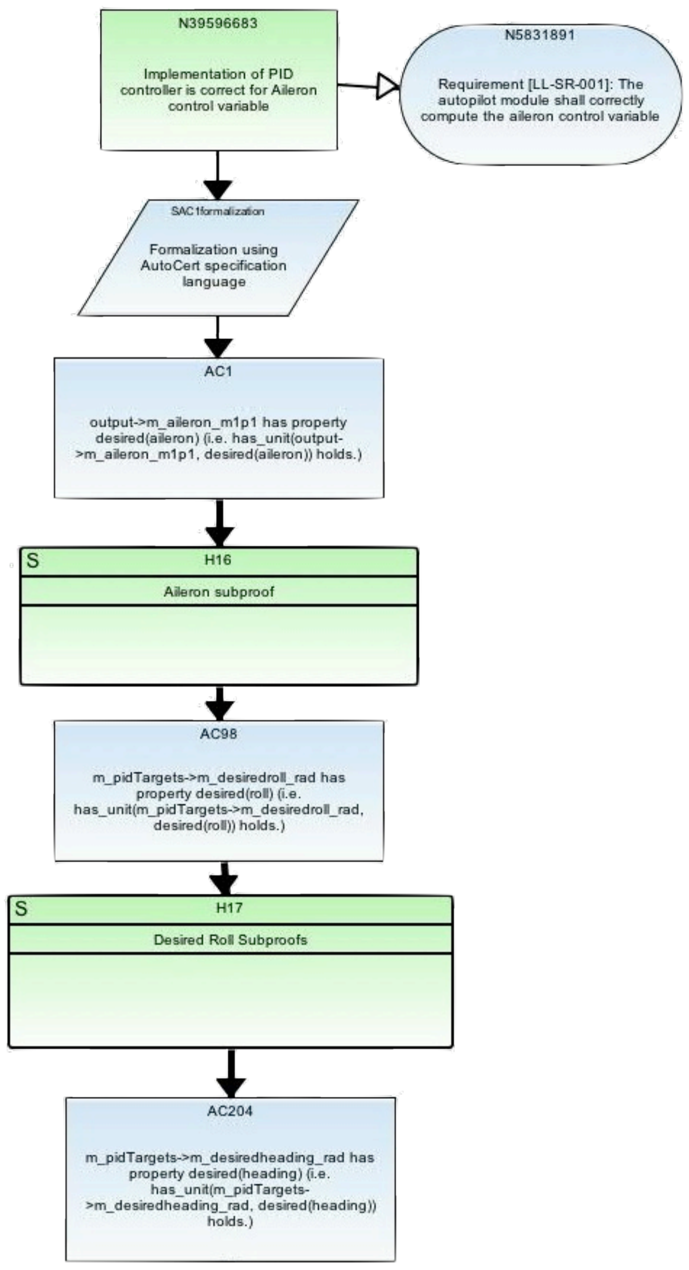

N39596683
Implementation of PID controller is correct for Aileron control variable
N5831891
Requirement [LL-SR-001]: The autopilot module shall correctly compute the aileron control variable
SAC1formalization
Formalization using AutoCert specification language
AC1
output->m_aileron_m1p1 has property desired(aileron) (i.e. has_unit(output->m_aileron_m1p1, desired(aileron)) holds.)
S
H16
Aileron subproof
AC98
m_pidTargets->m_desiredroll_rad has property desired(roll) (i.e. has_unit(m_pidTargets->m_desiredroll_rad, desired(roll)) holds.)
S
H17
Desired Roll Subproofs
AC204
m_pidTargets->m_desiredheading_rad has property desired(heading) (i.e. has_unit(m_pidTargets->m_desiredheading_rad, desired(heading)) holds.)

Figure 14: Hierarchical presentation of the aileron computation proof

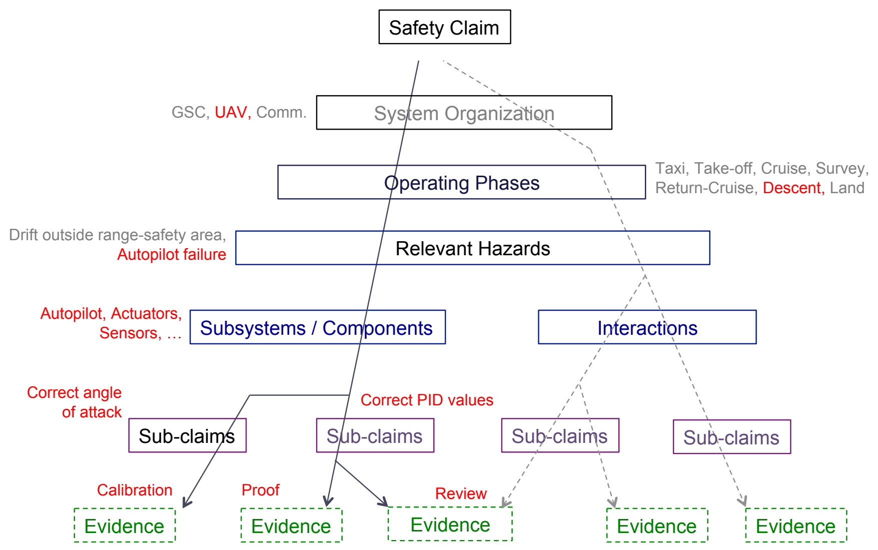


Figure 15: Presentation of the argument structure of the Swift safety case

3.5 Non-examples of Abstraction

There are restrictions on what can be abstracted inside a hinode. Firstly, to preserve the well-formedness of safety cases we need to ensure that the input and output node types are consistent. For example, a hierarchical strategy, like in Figure 7, must have a goal as an incoming node and goals as outgoing nodes, just like an ordinary strategy. We do not currently allow so-called strategicals, which would mix the types of input and output, although we may in future relax this restriction to allow for more compact hicases (since we can hide more contiguous chains).

Furthermore, we cannot abstract disconnected fragments of a safety case as there would be no path from the input goal to all the outputs. It is important to note that this restriction does not force each hinode to have only one input. The disconnectedness is with respect to the input, so multiple connections can enter a hinode[3].

Finally, we make the design decision to place any context, justification, and assumption nodes inside the hierarchical safety case, thus hinodes cannot have *IsContextOf* links connected to them.

3.6 Summary

In this section, we have demonstrated informally where hierarchy can be introduced and motivated the benefits. In summary, we have three different hinodes:

- Hierarchical strategies: abstracting a chain of related strategy applications.

[3]This property, desirable for a hierarchical node, is actually banned for non-hierarchical nodes in our formal definitions (to follow); however, we believe that it makes sense for hierarchical strategies to combine ‘in parallel’.

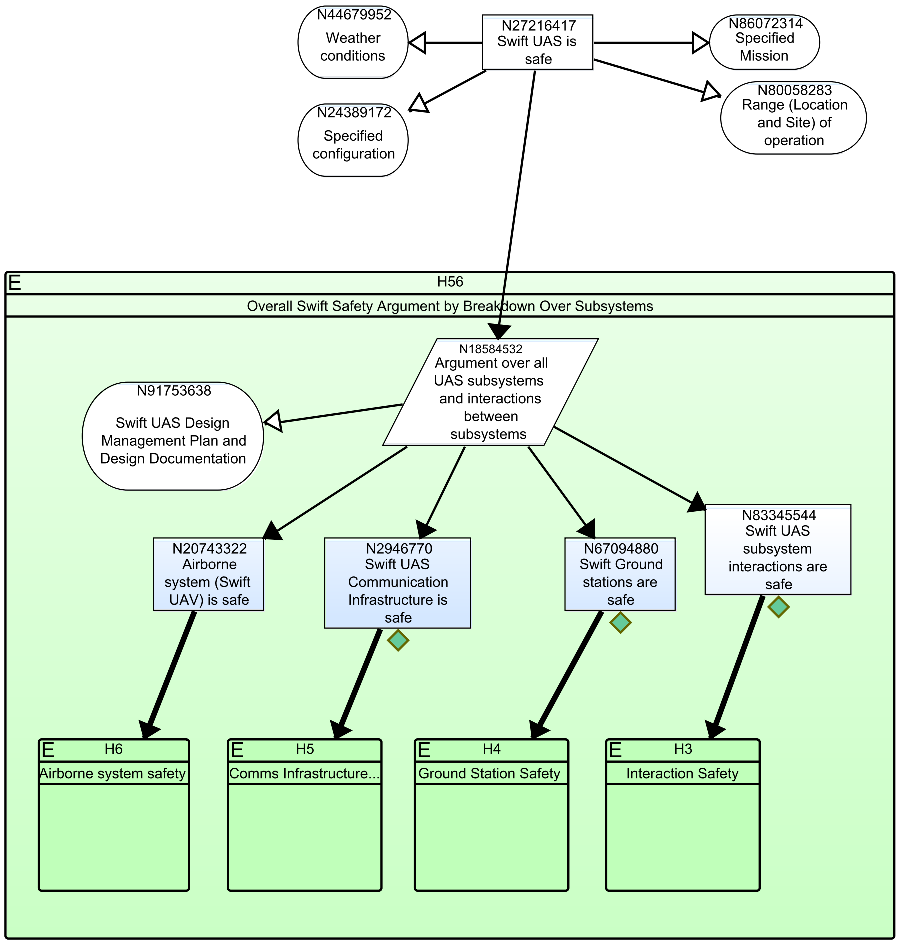

N44679952
Weather conditions
N27216417
Swift UAS is safe
N86072314
Specified Mission
N24389172
Specified configuration
N80058283
Range (Location and Site) of operation
E
H56
Overall Swift Safety Argument by Breakdown Over Subsystems
N18584532
Argument over all UAS subsystems and interactions between subsystems
N91753638
Swift UAS Design Management Plan and Design Documentation
N20743322
Airborne system (Swift UAV) is safe
N2946770
Swift UAS Communication Infrastructure is safe
N67094880
Swift Ground stations are safe
N83345544
Swift UAS subsystem interactions are safe
E
H6
Airborne system safety
E
H5
Comms Infrastructure...
E
H4
Ground Station Safety
E
H3
Interaction Safety

Figure 16: Hierarchical Swift safety case, with only top-level node open

- Hierarchical evidence: abstracting a *fully developed* chain of related strategy applications.
- Hierarchical goals: abstracting a chain of goals.

Hierarchical strategies and evidence have been implemented in AdvoCATE and hierarchical goals will in the near future. Hierarchical nodes are currently constructed manually in AdvoCATE by choosing the *input* and *outputs* for a particular node. If no output is that entire path is enclosed in the hinode. This approach ensure there is no ambiguity about the limits of the hinode. Although hinode construction is done by the user at present, we note that, especially in Example 3.1, the hierarchical presentations can be constructed automatically by any tool that is generating the safety case fragment. We would like to extend our safety case generation tools to automatically construct hinodes at appropriate locations. In the next section, we formalise the intuitions that we have given in this section.

4 Safety Cases

In this section, we give a mathematical account of standard safety cases and hierarchical safety cases by representing them formally as labelled tree — where the labelling function distinguishes the types of nodes — subject to some intuitive well-formedness conditions. In fact, we give definitions for a *partial safety case*[4] which can represent a safety case that is under construction.

4.1 Safety Cases

Definition 4.1 (Partial Safety Case)**.** A *partial safety case* is a triple $\langle N, l, \rightarrow\rangle$, consisting of nodes, a labelling function, and a connector relation, respectively. The labelling function $l : N \rightarrow \{s, g, e, a, j, c\}$ gives the type of the node in the safety case. The connector relation is defined on nodes: $\rightarrow$: $\langle N, N\rangle$. We define the transitive closure, $\rightarrow^*$: $\langle N, N\rangle$, in the usual way. We require the connector relation to form a *finite forest* with the operation $isroot_N(r)$ checking if the node r is a root in some tree[5]. Furthermore, the following conditions must be met:

1. Each part of the partial safety case is rooted by a goal $isroot_N(r) \Rightarrow l(r) = g$.
2. Arrows only leave strategies or goals: if $n \rightarrow m$, then $l(n) \in \{s, g\}$.
3. Strategies cannot connect to other strategies or evidence: if $n \rightarrow m$ and $l(n) = s$, then $l(m) \in \{g, a, j, c\}$.
4. Goals cannot connect to other goals: if $n \rightarrow m$ and $l(n) = g$, then $l(m) \in \{s, e, a, j, c\}$.

[4]Corresponding to the *core* GSN model.

[5]Safety cases are the case where we have a single root.

By virtue of forming a tree, we ensure that nodes cannot connect to themselves, that there are no cycles and, finally, that two nodes cannot connect to the same child node[6]. Additionally, we see that the GSN standard's requirement of two arrow-types (*IsSolvedBy* and *InContextOf*) has no semantic content, but rather provides an informational role.

Definition 4.2 (A goal-strategy restriction)**.** The safety case $\langle N', l\restriction_{N'}, \to\restriction_{N'}\rangle$ is a *goal-strategy* restriction of $\langle N, l, \to\rangle$ where $N' = \{n \mid n \in N \mathit{\ and\ } l(n) \in \{s, g, e\}\}$.

That is, we remove the context, justification, and assumption elements. By definition, these nodes are always leaves and alternatively could be understood as attributes of the strategy and goal nodes. We also say that a (total, i.e., unique root) safety case is *fully developed* if, for every goal n i.e., $l(n) = g$, we have $g \to^* n'$, with $l(n') = e$. That is, all paths lead to evidence.

4.2 Hierarchical Safety Cases

We define partial hierarchical safety cases, hicases, extend this model with an additional relation representing the hierarchical structure. We'll represent it as a partial order symbol $\leq$ where $n < n'$ means that the node n is inside n'. We wish to define hicases in such a way that we can always *unfold* all the hierarchy to regain an ordinary safety case. Thus:

Definition 4.3 (Hicases)**.** A partial *hierarchical safety case* is a tuple $\langle N, l, \to, \leq\rangle$. The set of nodes and labelling function are as above. The forest $\langle N, \to\rangle$ is subject to the same conditions as Definition 4.1. The hierarchical relation fulfills the axioms of a partial order and can thus also be viewed alongside N as a forest. Finally, we impose conditions on the interaction between the two relations:

1. If v is a local root (using $\to$) of a higher-level node w (i.e., $v < w$), then

$$l(w) = \begin{cases} g & \text{if } l(v) = g \land \forall v'\, v''.\, (v' < w \land v' \to v'' \land v'' \not< w) \Rightarrow l(v'') = s \\[1em] s & \text{if } l(v) = s \land \\ & (\forall v'\, v''.\, (v' < w \land v' \to v'' \land v'' \not< w) \Rightarrow l(v'') = g \\ & \ \lor \text{ subtree rooted at } v \text{ is not fully developed}) \\[1em] e & \text{if } l(v) = s \land \\ & (\nexists v'\, v''.\, (v' < w \land v'' \not< w \land v' \to v'') \\ & \ \land \text{ subtree rooted at } v \text{ is fully developed}) \end{cases}$$

[6]This last condition (two nodes can't have the same child) is actually a restriction of the GSN standard, which doesn't explicitly disallow this, but we believe this to be reasonable property to have. The AdvoCATE tool doesn't currently support this restriction: in Figure 12, we see, for example, a goal with two incoming strategies.

That is:

- A hierarchical goal node must be rooted by a goal and any nodes immediately outside the hierarchical goal must be strategy nodes.
- A hierarchical strategy must be rooted by a strategy and either a) any nodes immediately outside the hierarchical strategy must be goals; or, b) the subtree rooted at v inside is not fully developed. The latter case catches the possibility that there are no outgoing goals, but the node is not evidence.
- A hierarchical evidence node is the special case of a hierarchical strategy with no outgoing goals, but where the subtree rooted at v is fully developed.

2. The connectors will target the outer nodes: $v \to w_1$ and $w_1 < w_2$ then $v < w_2$.
3. Connectors come from inner nodes: if $v \to w_1$ and $w_1 \leq w_2$ then $v = w_1$.
4. Hierarchy and connection are mutually exclusive: $v \leq w$ and $v \to^* w$ means $v = w$.
5. two nodes which both at the top level (or immediately included in some node) means at most one has no incoming $\to$ edge. That is:

$$siblings_i(v_1, v_2) \wedge isroot_s(v_1) \wedge isroot_s(v_2) \implies v_1 = v_2.$$

The first condition above formalises our intuition that a hierarchical strategy must take goals as an input and return goals as an output (possibly in the form of a non-fully developed tree), hierarchical evidence must have no outputs and enclose a fully-developed safety case, and hierarchical goals must take strategies as inputs and outputs. We phrase it in terms of the immediately enclosed items: that they must be of the same type as the hinode. Hierarchical evidence is subject to stricter conditions: it must not have any outputs and the safety case it encloses must be fully developed. We can view abstract evidence as an abstract strategy without outgoing goals just as evidence is an axiomatic strategy.

The latter conditions (2–5)[7], are designed to produce a mapping from a hierarchical safety case to its ordinary safety case unfolding: its *skeleton*. We show below that safety cases can be viewed as (trivial) hicases and that the skeleton operation unfolds into an ordinary safety case

4.3 Relating Safety Cases and Hicases

4.3.1 Safety Cases as Hicases

Before defining the more complex transformation of a hicase into its *skeleton* we briefly note that a safety case $\langle N, l, \to \rangle$ can be mapped to a hicase $\langle N, l, \to, id_V \rangle$ where id_V is the trivial partial order with only reflexive pairs. This ordering trivially satisfies all the well-formedness properties of a hicase.

[7]These are identical to the well-formedness conditions for Hierarchical Proofs (or hiproofs) that inspired this work [8]. For more discussion about hiproofs, see Section 7.1.1.

4.3.2 Skeleton of a Hicase

We define an operation *sk*, mapping hicases into ordinary safety cases and show that the tuple it constructs is indeed well-formed w.r.t the safety case conditions.

Theorem 1 (Skeleton)**.** *The operation sk which maps a hierarchical safety case $\langle N, l, \rightarrow, \leq\rangle$ to $\langle N', l', \rightarrow'\rangle$, where N' is the set of leaves of $\leq$, l' is the restriction of the labelling function l, and $v_1 \rightarrow' v_2$ iff $\exists w \in N$ such that $v_2 \leq w$ and $v_1 \rightarrow w$ maps a well-formed hierarchical safety case to an ordinary safety case.*

Proof sketch. The relationship between Hiproofs and Hicases (as well as the corresponding relationship between safety cases and proofs) allows us to claim that the mapping constructs the appropriate forest structure on $\langle N', \rightarrow'\rangle$. We simply need to show that the well-formedness conditions 2–4 of Definition 4.1. Condition 2, for example, (if $v_1 \rightarrow v_2$ then $l(v_1) \in \{s, g, e\}$) comes for free since if $v_1 \rightarrow w$ then it already has this property for $v_1 \rightarrow' v_2$.

4.4 Extending the Core GSN Model

As discussed in Section 2, the core GSN model is often extended with *entity abstraction* annotations — which can state if a node is *undeveloped* or *uninstantiated* — and with patterns and modules[8]. Additionally, the AdvoCATE tool extends the GSN node notion to include some meta-data about risk, whether the property is a high-level or low-level requirement etc which can, for example, influence the colour of a node. Our core definition of a safety case can be extended to *store* this detail by considering our node set N as a record, with various projections. As an example, here we simply implement the entity abstractions, with predicates: $isundev(N)$ and $isuninst(N)$[9].

Our interest in these extensions comes from their interaction with the hierarchical structure: the hinodes must also have a *developed* and *instantiated* attribute **and** it must be consistent with its contents. We ensure this consistency by providing well-formedness rules connecting the hinode property and its child properties. However, since we do not want to fix the attributes available, we instead give schemas for consistency rules. The general structure is as follows:

$$\frac{f_p(v_1, \ldots, v_n, n)}{p(n)}\ v_i < n$$

By this, we mean that a property holds (or has a particular value) if a function of it and it all the contained nodes also has that value. We illustrate this schema with the *undeveloped* property:

$$\frac{(undev(v_1) \wedge l(v_1) = l(n)) \vee \ldots \vee (undev(v_n) \wedge l(v_n) = l(n))}{undev(n)}\ v_i < n$$

[8]In fact, the *undeveloped* entity abstraction is part of the core GSN, but we didn't include it in our original definition for simplicity.

[9]We are cheating a little. Technically, only goals, strategies, and evidence can be abstracted. We assume that contexts, assumptions, and justifications always have these predicates false.

That is, a hinode is undeveloped if any of its contained nodes (of the same type) are also undeveloped.

4.5 Viewing the Hierarchy

It can also be useful to view the hierarchical relation as a tree in its own right. This presentation makes explicit the nesting structure of the individual hinodes: which hinode is enclosed by which other hinode. Figure 17 shows an example of this *hierarchy tree* for the hicase in Figure 10. The definition of the hierarchy tree is simply a pair $\langle N, \leq\restriction_{\neg leaf}\rangle$. That is, all the nodes in the hierarchy that are not leaves. This is precisely the hinodes as no other types of nodes are allowed to contain elements.

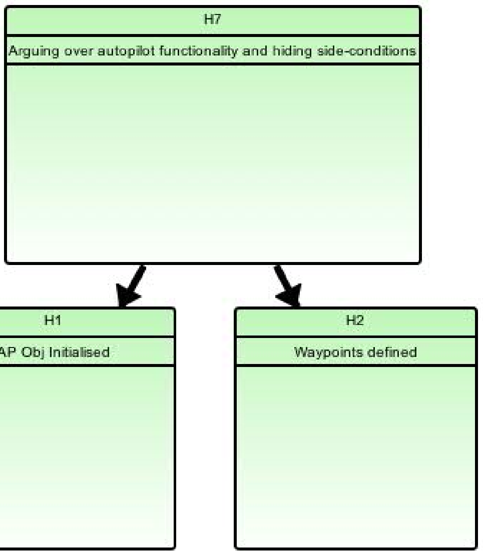

H7
Arguing over autopilot functionality and hiding side-conditions
H1
AP Obj Initialised
H2
Waypoints defined

Figure 17: A presentation of the hierarchy of a safety case

5 An Implementation-focussed Definition

We have implemented hicases in AdvoCATE, which was designed using a different, but equivalent, definition of a safety case[10]. Assuming that we have finite, disjoint sets of goals (G), strategies (S), evidence (E), assumptions (A), justifications (J) and, contexts (K), we give the following definition.

[10]This is a slightly revised definition to that given in [3]. We have restricted each goal/strategy to have one input strategy/goal and generalised what can be attached to goals to also include assumptions and justifications.

Definition 5.1 (Partial Safety Case: Type 2)**.** A partial *safety case*, $\mathcal{G}$, is a tuple

$$\langle G, S, E, A, J, K, sg, gs, gc, ga, gj, sa, sc, sj \rangle$$

with the functions:

- $sg : S \to \mathcal{P}(G)$, the subgoals of a strategy;
- $gs : G \to \mathcal{P}(S \cup E)$, the strategies and/or evidence of a goal;
- $gc : G \to \mathcal{P}(K)$, the contexts of a goal;
- $ga : S \to \mathcal{P}(A)$, the assumptions of a goal;
- $gj : S \to \mathcal{P}(J)$, the justifications of a goal;
- $sa : S \to \mathcal{P}(A)$, the assumptions of a strategy;
- $sj : S \to \mathcal{P}(J)$, the justifications of a strategy;
- $sc : S \to \mathcal{P}(K)$, the contexts of a strategy.

With the properties:

- Each goal has only one input strategy. That is, for every $g \in G$ if $g \in sg(s)$ and $g \in sg(s')$ then $s = s'$.
- Each strategy has only one input goal. That is, for every $s \in S$ if $s \in gs(g)$ and $s \in gs(g')$ then $g = g'$.

We say that g' is a subgoal of g whenever there exists an $s \in gs(g)$ such that $g' \in sg(s)$. We can then define the descendant goal relation, $g \to g'$ off g' is a subgoal of g or there is a goal g'' such that $g \to g''$ and g' is a subgoal of g''. We require that the $\to$ relation is a DAG with roots R. We define a similar relation for sub-strategies, including evidence as an axiomatic strategy. For simplicity, we overload the same symbol.

Translating Between Definitions. We can define a mapping from safety cases (which we will call type-1 safety cases) to type-2 safety cases and vice versa. This mapping has the property that it preserves node connections and well-formedness conditions for either definition. Informally, we can map a safety case of type-2 to a safety case by collapsing the node sets and constructing the labelling function in the obvious way: if $n \in G$ in the type-2 safety case, then $l(n) = g$ in the type-1 safety case. We construct the connector relation by flattening the target power sets for each function. The resulting structure satisfies the properties in Definition 4.1.

In the other direction, we can translate a type-2 safety case to a type-1 safety case by means of a partition of the node set N using the labelling function. We can then construct each mapping function by partitioning the connector, again using the labelling function.

5.1 Hierarchical Extension

Formally, we extend the type-2 description of a safety case to include a set H of hinodes, finite and disjoint from the other node types. Thus, we extend a safety case as follows[11]:

Definition 5.2 (Hierarchical Safety Case)**.** A partial *hierarchical safety case*, $\mathcal{G}$, is a tuple:

$$\langle G, S, E, A, J, K, H, sg, gs, gc, ga, gj, sa, sc, sj, hi, ho \rangle,$$

extending hicases with the additional functions:

- $hi : H \to G$, the input goal of a hinode;
- $ho : H \to \mathcal{P}(G)$, the set of output goals of a hinode.

We ensure that the following properties about the hierarchical links hold:

1. Inputs and outputs are connected: $\forall\ h \in H$, if $g_{out} \in ho(h)$ and $hi(h) = g_{in}$ then $g_{in} \to g_{out}$.
2. Outputs are disconnected: $\forall g_1, g_2 \in ho(h),\ g_1 \to g_2 \Rightarrow g_1 = g_2$.
3. Hinodes cannot overlap: if $h_1, h_2 \in H$ and $hi(h_1) \to hi(h_2)$ then for **every** $g \in ho(h_1)$ such that $hi(h_2) \to g$ we must have $g' \in ho(h_2)$ such that $g' \to g_{h_1}$. We say h_1 *encloses* h_2.
4. Additionally, if $hi(h1) = hi(h2)$ then either for every $g \in ho(h1)$ there is a $g' \in ho(h2)$ such that $g' \to g$ **or** for every $g \in ho(h2)$ there is a $g' \in ho(h1)$ such that $g' \to g$. That is, one must wholly enclose the other.

Note that the complicated property about overlapping comes for free in our partial order version of this definition. Intuitively, the overlap property basically says that every pair of hinodes must either be independent or any interaction must be a complete nesting. For example, in Figure 18, we cannot construct a hinode containing $S2$ and $S3$ only, because that would break the overlap property since that hinode, let's call it $H2$, would require $hi(H2) = G2$ and $ho(H2) = \{G4\}$. However, since $G1 \to G2$ and $ho(H1) = G3$ and $hi(H2) \to G3$, we trigger the condition so we must have an element, g, of $ho(H2)$ such that $g \to G3$, but this is not the case as the only output of $H2$ is $G4$. Graphically, we see that this would mean the hinodes would intersect.

We highlight two special cases of hicases:

- When $ho(h) = \{\}$ and the hicase rooted at $hi(h)$ is fully developed, the hierarchical strategy becomes a hierarchical evidence node. That is, it encompasses the rest of the safety case tree rooted at $hi(h)$. If it is not fully developed, then it is still a hierarchical strategy.
- For a hierarchical strategy h, with $hi(h) = g$ and

$$ho(h) = \{g' \mid g' \text{ is a subgoal of g } \}$$

the hierarchical strategy is *degenerate*. It only covers one strategy.

[11]Note that this (preliminary) definition does not account for hierarchical goals, but these can be added similarly to Definition 4.3.

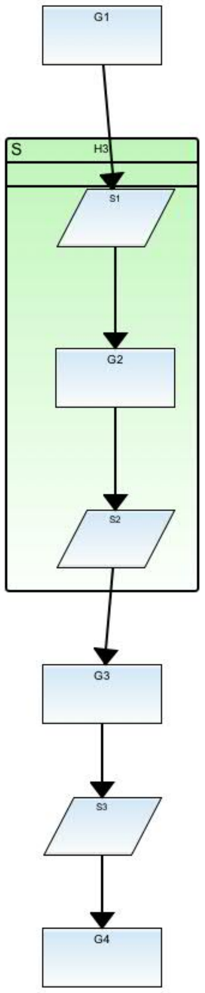


Figure 18: A hicase with the potential for overlapping hierarchy

6 Implementation

We have implemented hicases in AdvoCATE, providing basic features for constructing, modifying, and viewing hinodes. For a complete description of the features of AdvoCATE, we refer the reader to [5]. The main hierarchical features provided are:

- The ability to modify an existing safety case to add hierarchical structure;
- Two ways to view hierarchical objects: an *open* and *closed* view[12];
- The ability to modify the contents of a hierarchical safety case: abstracting more or less of the safety case;
- The ability to view a tree representation of the hierarchical structure.

In this section, we show how each of these features can be used, based on a simple safety case, which we have constructed as an exemplar, given in Figure 19. It is *fully developed* and the root goal is $G1$. First the strategy $S1$ breaks $G1$ into two simpler goals: $G2$ and $G3$. $G3$ is solved directly using an evidence node ($E1$) and $G2$ requires an additional strategy application.

6.1 Creating Hierarchical Nodes

In order to create a hierarchical node, the user will select the goals (in the case of constructing a hierarchical strategy) that they wish to delimit the hinode from outside. Once selected, the user can use the 'Abstract Nodes' feature to construct the hinode. The selected nodes are used to construct the hi and ho functions in Definition 5.2. The set of nodes is analysed to ensure that:

- One node is suitable as an input: that is, there is a path from it to all other nodes;
- All the other nodes (the outputs) are disconnected;
- This new hinode will not overlap with any pre-existing strategies.

Once the well-formedness checks are performed, the node is created in the *open* view, with no description and coloured green by default. The user can then click to add a meaningful name to the hinode. If, for example, the user selected $G1$ and $G4$ as the delimiting goals, then the resulting safety case is shown in Figure 20. Note, that in accordance with the interpretation the non-selected path is completely enclosed. In this open view, all the details of the safety case are seen within a hierarchical box.

6.2 Viewing Hierarchical Details

The default view for a hinode is called the *open* view. In this view we display both the hinode and its contents, with incoming links going directly to and from the enclosed nodes. We offer one core additional view: the *closed* view, which is where all the contained nodes are hidden and the only links shown are the hierarchical links. It is

[12]Actually in the implementation we provide three, but only two are available in the interface.

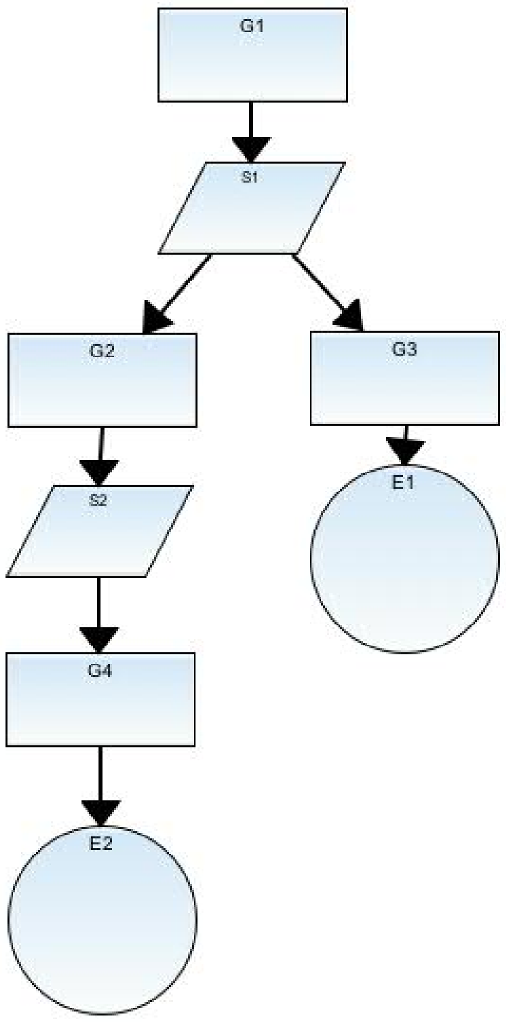


Figure 19: Original safety case in AdvoCATE

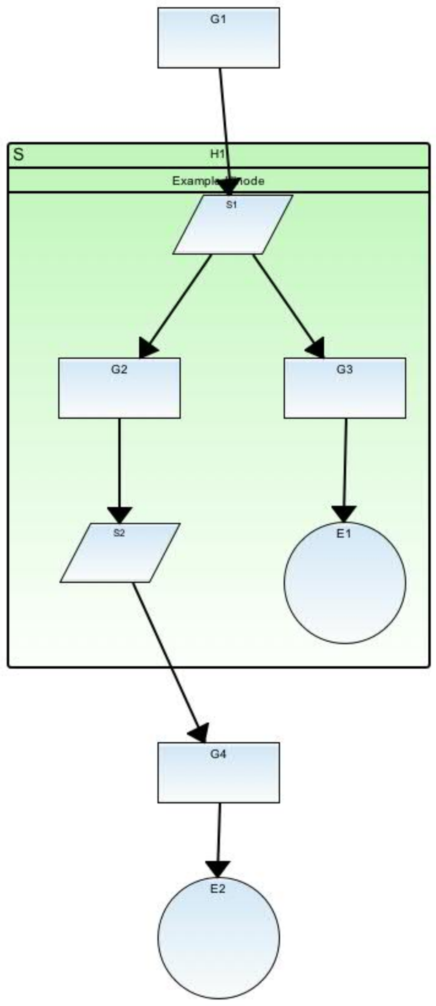


Figure 20: Safety case with hierarchy created

in this view that we see the advantages of hierarchical structure: since the resulting diagram contains fewer visible elements, it becomes easier to view. Additionally, there is also a view that we call the *flat* representation of a hierarchical node. This view can be considered as the raw representation of the definition: with both hierarchical links and non-hierarchical links visible.

Nested hierarchical nodes will be visible only when their parent hinode is in the open state and nested hinodes have their states preserved when a parent hinode is modified. Since the opening and closing of hinodes affects their size and shape, the system will automatically re-layout the changed parts of the diagram. This is necessary, particularly when the diagram becomes large, in order to ensure that nodes are not accidentally hidden.

6.3 Modifying a Hinode

Once a hinode has been created, we may wish to change its contents. It can be modified in three atomic ways[13]:

- We can delete it entirely;
- We can enclose less;
- We can enclose more.

We provide a **delete** operation for hinodes, which will remove the hinode and place all its contents at the level directly below (which is where they would have been had the hinode being deleted not existed).

In order to change the contents of a hinode, we need to change either the input goal or the set of output goals. We have not yet implemented this behaviour but it is expected that the user can click and drag goals in and out of a hinode. The tool will then calculate whether such a move is legal (preserving the well-formedness conditions for hicases) and perform the change. As a simple example: consider the hicase in Figure 20. If we wanted to remove G3 and E1 from the hinode H2, we could click and drag G3 outside the hinode.

7 Conclusions

In this report, we have introduce hierarchical safety cases as an approach to help improve maintainability, understandability, and checking of safety cases. We first motivated the work with examples derived from real safety cases then gave a theoretical account of the GSN notation for safety cases before extending it with a hierarchical interpretation. We then described our prototype implementation of hierarchical safety cases in the AdvoCATE safety case editor. In this section, we highlight some important related work and hint at directions for future development of our work.

[13]Note that only the delete fragment is fully implemented at present.

7.1 Related Work and Concepts

7.1.1 Hiproofs

Hiproofs, introduced in [8], are the immediate inspiration for hicases. In particular, we follow the hiproof notation for the graphical representation of hierarchical nodes. Hiproofs *could* be viewed as a more general model (for hierarchical trees) than safety cases, without the particular node typing present in hicases. An alternative understanding would be to consider a hiproof as the strategy/evidence subset of the hicases representation (where goals flow is represented by the connections). Hiproofs are simplifications of proof trees in real-life systems (which often consist of meta-data), being intended for theoretical study of proof systems. This contrasts slightly from our approach as we also attempt to account for meta-data in our definitions.

7.1.2 Safety Case Modules

Safety cases have a built-in *module system*, which is described in [1, 9]. As in programming languages, the module system is designed to allow *reuse* as well as limiting the changes required to the safety case when a particular aspect of the underlying system is changed. Part of the motivation for safety case modules lies in the movement in industry towards more modular systems. Safety case modules are designed to naturally represent modular systems. Modular safety cases can be easier to understand and justifiably add some hierarchical structure to a safety case, so we offer a comparison here. We must be clear, though, that we do not claim to subsume the features of modules with hicases; rather, we see these as being complementary.

A closed hierarchical goal can mimic the appearance of an *away goal* reference (and many of them); however, there is an important difference: the away objects are simply references to a separate safety case fragment; whereas, for hierarchical goals, it is simply an additional node enclosing structure that is there (and is also repeated if existing more than once in a safety case). Additionally, hicases offer the notion of a hierarchical strategy — an enclosure of (possibly) a complex (unfinished) safety case fragment — which does not have an equivalent notion in the module system. Modules, however, cannot be nested in the way that hicases can, thus offering only one abstract view. Additionally, one can see the module system as working at a larger level: a module is typically a large segment of a safety case, but we view hinodes as being viable at all scales. Modules also have informal *contracts* that they must fulfill to be well-formed, but hinodes do not enforce any *semantic* properties.

7.1.3 Hierarchy in Safety Cases

Hierarchical safety cases have been proposed before, in [11], where Stone proposed and built a hierarchical presentation of a safety case for a frigate upgrade. In this safety case, hierarchy was represented simply by indentation in the spreadsheet safety case and showed the basic hierarchical decomposition of the safety case: arguing over subsystems, hazards etc. In our terminology, we would say they can only construct abstract evidence. Their tool doesn't offer the flexibility of abstraction given by abstract strategies. In fact, Denney et al. have also presented a tabular formulation of safety cases in [6] and it would be interesting to see if we could extend it to represent hierarchical safety cases.

7.2 Future Work

We address the avenues for future work in the theoretical account of safety cases and in further development of AdvoCATE separately.

7.2.1 Hicases Development

There are a few possible threads for future development:

Extending the Hicases Definitions Our current definition for safety cases and hicases only accounts for the core GSN language and potential meta-data extensions. Most safety cases in practice make use of either (or all) of the modular extension to safety cases and the pattern language for safety cases. We would like to give an account for each of these within our model. The module language, in particular, would require careful thought to ensure no inconsistencies are introduced. Additionally, we would like to further explore our definitions for hicases and formally relate the two hierarchical definitions that we currently have[14]. A detailed account of the skeleton operation is also required.

We have not carefully considered assumptions and contextual nodes. Checking the *consistency* of assumptions and contexts is an important part of (manually) verifying a safety case. This can be a difficult and confusing task, particularly with large safety cases, since assumptions and contexts for a goal are inherited from their parents. It would be fruitful to provide a clear understanding to these notions. In particular, we could formalise contexts, which will allow us to *transform* contexts/assumptions. One possibility is to provide a formal logic (propositional should be expressive enough) for writing assumptions, then giving some logical rules describing how context and assumption lists are propagated.

We would also like to investigate the formal notion of a hicase *view*: a slice through the hierarchy which is actually a normal safety case; and, *refinement* of hicases: providing a mathematical meaning for well-formed changes to the hicase. Both of these exist informally in the implementation and it is important to capture it formally.

Hierarchy Construction via Patterns Safety cases are often developed for *patterns*: commonly used specifications for safety case fragments. Given input data — from a requirements table, for example — the pattern can be instantiated to a safety case fragment. We can extend the safety case pattern language to also contain hierarchical constructs. In particular, we can contain the whole pattern in a hierarchical structure. For example, this pattern can be enclosed in a hierarchical goal structure that explains that this section of the safety case provides the hazard decomposition. In this way, the hierarchy can be seen as providing a record of pattern instantiation.

[14]We believe them to be equivalent, but have not yet proved it.

We would like to give an account for *pattern instantiation*. One interesting possibility is to provide a term language for writing patterns. These terms, when applied to an input could evaluate (possibly partially) to construct safety cases. This approach is loosely analogous to writing tactics to construct proof trees and we could look to the hitac tactic language for hierarchical proofs [2].

Extending Existing Safety Case Tools We plan to extend our existing tools for automatically constructing safety cases so that they can construct hierarchical safety cases. For example, we could extend the AutoCert tool to construct a hicase like we presented in Section 3.4. Denney et al. also developed a tool for constructing safety cases from sets of hazards and requirements tables [3]. Again, this tool could be usefully extended to construct hicases.

Utilising Meta-data for Hierarchical Presentation An example application of hierarchy is to give a method to highlight different technical regions of a safety case. For example, we could specify an argument, within an abstract strategy or goal, as requiring an avionics expert to verify correctness of the argument. The EGSN model, however, has a concept of meta-data which can be used to annotate nodes. We plan to investigate in the future what potential consequences for hierarchy this can have. It can certainly help deal with regions requiring an expert verification. Additionally, meta-data could help provide information for a *Google maps*-style view of a safety case: where major goals (cities) are visible when *zoomed out* and gradually more detail is fleshed out[15].

7.2.2 Improving AdvoCATE

There are many interesting avenues for future development of the AdvoCATE tool: our prototype implementation has really only scratched the surface. Here, we confine ourselves to the developments specifically related to hierarchy. Most immediately required are interface modifications (and the required changes will become clear with use) to improve the users experience with hierarchical proof. Currently, for example, the laying out algorithm does not behave optimally for presenting hierarchical nodes and (minimal) manual adjustment is typically required.

Firstly, we wish to utilise the meta-data that is central to the system. Currently, hinodes can be manually annotated, but we would like to be able to derive this data from the enclosed elements. For example, we can mark a hinode as uninstantiated if any of its elements are uninstantiated. Secondly, we would like to utilise AdvoCATE's transformation system, described in [5], to collect information about hierarchy and possibly learn potential hierarchical structure. For example, one might be able to notice and abstract patterns from existing safety cases. Finally, we would like to implement the hierarchical patterns that have been discussed above.

[15]Buggy versions of this may be known as the Apple Maps view.

References

[1] GSN community standard version 1. Technical report, Origin Consulting (York) Limited, 2011.

[2] David Aspinall, Ewen Denney, and Christoph Lüth. A tactic language for hiproofs. In *Proceedings of the 9th AISC International Conference, the 15th Calculemus Symposium, and the 7th International MKM Conference on Intelligent Computer Mathematics*, pages 339–354, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Ewen Denney and Ganesh Pai. A lightweight methodology for safety case assembly. In Frank Ortmeier and Peter Daniel, editors, *SAFECOMP*, volume 7612 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2012.

[4] Ewen Denney, Ganesh Pai, and Ibrahim Habli. Perspectives on software safety case development for unmanned aircraft. In Robert S. Swarz, Philip Koopman, and Michel Cukier, editors, *DSN*, pages 1–8. IEEE Computer Society, 2012.

[5] Ewen Denney, Ganesh Pai, and Josef Pohl. AdvoCATE: An assurance case automation toolset. In Frank Ortmeier and Peter Daniel, editors, *SAFECOMP Workshops*, volume 7613 of *Lecture Notes in Computer Science*, pages 8–21. Springer, 2012.

[6] Ewen Denney, Ganesh Pai, and Josef Pohl. Formal verification methodology for automated construction of software safety cases. AFCS Milestone Report SSAT.1.3.VVFCS.4.05V.4.SI.12.03, NASA Ames Research Center, Apr. 2012.

[7] Ewen Denney, Ganesh Pai, and Josef Pohl. Heterogeneous aviation safety cases: Integrating the formal and the non-formal. In *17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, Paris, France, Jul. 2012.

[8] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci.*, 155:341–359, 2006.

[9] Tim Kelly. Concepts and principles of compositional safety case construction. Technical Report COMSA/2001/1/1, University of York, 2001.

[10] Kevin Kinsella. UK offshore safety lessons learned. ERM Risk Practice. SPE Seminar, Houston, TX, 2010.

[11] Gordon Stone. On arguing the safety of large systems. In *Tenth Australian Workshop on Safety-Related Programmable Systems*, volume 162 of *ACM International Conference Proceeding Series*, pages 69–75, 2006.