

# Querying Safety Cases

Ewen Denney<sup>1</sup>, Dwight Naylor<sup>2</sup>, and Ganesh Pai<sup>1</sup>

<sup>1</sup> SGT / NASA Ames Research Center  
Moffett Field, CA 94035, USA.

{`ewen.denney`, `ganesh.pai`}@nasa.gov

<sup>2</sup> Rensselaer Polytechnic Institute  
Troy, NY 12180, USA.  
`naylod@rpi.edu`

**Abstract.** Querying a safety case to show how the various stakeholders' concerns about system safety are addressed has been put forth as one of the benefits of argument-based assurance (in a recent study by the Health Foundation, UK, which reviewed the use of safety cases in safety-critical industries). However, neither the literature nor current practice offer much guidance on querying mechanisms appropriate for, or available within, a safety case paradigm. This paper presents a preliminary approach that uses a formal basis for querying safety cases, specifically Goal Structuring Notation (GSN) argument structures. Our approach semantically enriches GSN arguments with domain-specific metadata that the query language leverages, along with its inherent structure, to produce *views*. We have implemented the approach in our toolset AdvoCATE, and illustrate it by application to a fragment of the safety argument for an Unmanned Aircraft System (UAS) being developed at NASA Ames. We also discuss the potential practical utility of our query mechanism within the context of the existing framework for UAS safety assurance.

**Keywords:** Safety cases, Queries, Views, Formal methods, Automation

## 1 Introduction

A safety case essentially provides an audit trail, which can assist in convincing the various stakeholders of a system, including regulators, that the system is acceptably safe [1]. One of the motivations to use structured arguments in developing a safety case is to provide a means to explicitly justify safety considerations from concept, through requirements, to the evidence of risk mitigation/control. Additionally, argument structures are intended to make a safety case easier to comprehend and, thereby, more efficient to review critically [2]. To improve clarity in presenting the underlying reasoning, the Goal Structuring Notation (GSN) [3] provides a graphical syntax with which to specify the appropriate argument structures.

Previously [4], we identified the need to present role-specific information to subject-matter experts to improve the comprehensibility of a safety argument. Furthermore, as a system evolves through its lifecycle, so should its safety case, i.e., system changes, assumptions that are validated/invalidated, and observations of safety performance, for example, should translate into updates of the safety case so that the system and its safety

case are mutually consistent. We believe that one of the first steps to address these needs is through an approach for safety case *queries*. Although the potential to query a safety case has been put forth previously as one of the benefits of using safety cases, and as a way for stakeholders to understand how safety concerns have been addressed [5], to the best of our knowledge there is scant guidance on a principled way for querying safety cases.<sup>3</sup>

The application domain motivating our work is Unmanned Aircraft Systems (UASs). We are interested in creating a framework for argument-based assurance of airworthiness and flight safety of UAS, which augments the existing processes and reuses the artifacts produced to the extent possible, so as to ease its adoption in practice. A broad goal is to be able to address the requirements from the relevant regulations/standards. An additional goal is to be able to support safety case development. In general, safety engineers and system developers need to understand and communicate what they (or others) have already done, that which remains to be done, and how different parts of the argument may relate to each other and to the system.

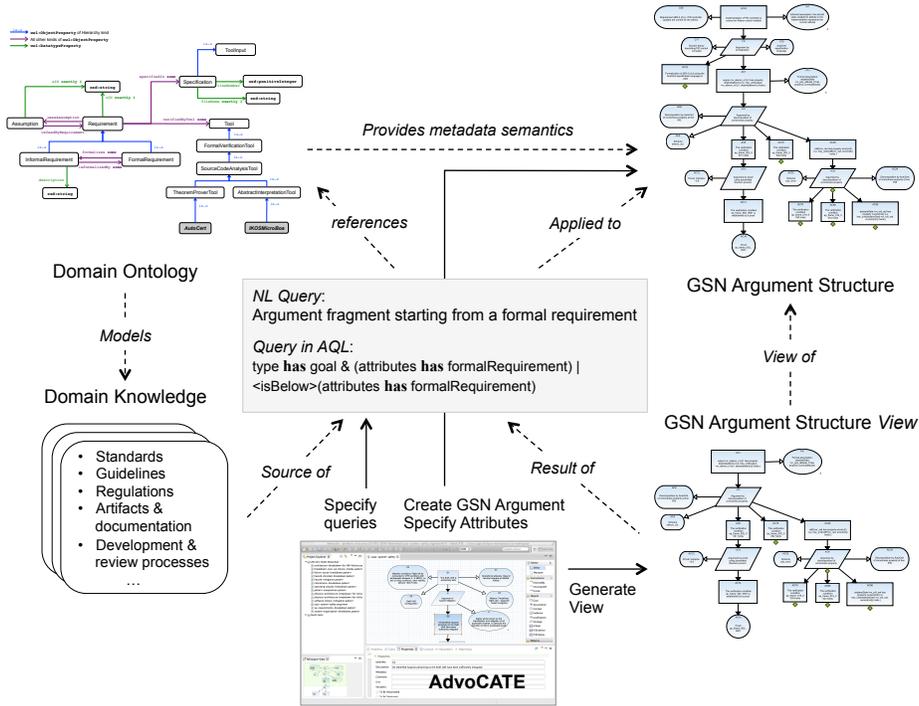
These issues are also critical for safety/assurance case assessors. To determine safety case fitness for purpose, it is necessary to involve all the relevant stakeholders so that they may understand the (safety) claims made, and challenge the reasoning and evidence presented. However, safety cases typically contain heterogeneous reasoning [6] and a wide variety of evidence, e.g., the mandated work products which show compliance to the relevant regulations and standards, the results of analyses (safety, system, and software), various inspections, audits, reviews, simulations, verification activities including various kinds of subsystem/system tests, and, if applicable, also the evidence of safe performance from prior operations. In other words, safety cases can easily amass a large amount of information. For example, the preliminary safety case for ADS-B airport surface surveillance applications [7] is about 200 pages long. Thus, partly due to the size and diversity of information contained in a safety case it may not be straightforward (or possible) for all stakeholders to locate and/or understand all the arguments presented along with their different elements.

In this paper, our main contribution is a preliminary approach (Section 2), and a formal basis for querying GSN safety case argument structures (Section 3). We define queries as properties of GSN nodes, constructed from unary and binary relations, and take the result of executing a query to be an *argument structure view*, rather than simply the list of nodes which satisfy the query. We also describe the *Argument Query Language*, AQL<sup>4</sup>, and give formal semantics for both queries and views based on an earlier semantics for argument structures. We have implemented the approach in our toolset AdvocATE [8], and illustrate its application on a fragment of a safety case argument structure for the Swift UAS, under development at NASA Ames (Sections 4 and 5).

In this first implementation, we have limited ourselves to querying the argument structure, rather than the entire assembly of artifacts comprising a safety case. Next, we describe our approach for querying and how it can help to address the problem of accessing (and understanding) the rich variety of information contained in a safety case.

<sup>3</sup> However, we acknowledge that existing safety case tools may provide a search functionality to locate the information of interest.

<sup>4</sup> In Islamic philosophy, *'aql* is the use of logical inquiry as a basis for law.



**Fig. 1.** Methodology for querying safety cases using AdvOCATE: Enrich GSN argument structures with metadata drawn from domain ontologies and use Argument Query Language (AQL) queries to create views. The dotted lines give the role of each element in relation to the others, while the solid lines give the role of the tool.

## 2 Methodology

We describe our approach mainly with respect to the GSN argument structures created using our tool AdvOCATE (although the principles can be applied more generally). AdvOCATE already offers several features: filtering and searching the argument, and showing/hiding sub-arguments relative to a node. The search mechanism allows a string search on different node fields (e.g., identifier, description, etc.), which can also be filtered by node type. However, our requirement is to develop a mechanism to query arguments in a much richer way, making use of both syntactic (i.e., structural) and semantic information. Fig. 1 shows our methodology for using the Argument Query Language (AQL) to query safety case argument structures and create views.

### 2.1 Semantic Enhancement

The main idea is first to semantically enrich the GSN nodes in the argument structure. Thus, in addition to the descriptive text, e.g., the actual claim for a goal node, we associate nodes with *metadata*, given as a set of *attributes*.

For example, we can use metadata to relate nodes containing informal claims, with those containing the formal equivalents. We can also use metadata to indicate an association between an instance node in an argument, and the source node in the pattern from which it was generated. Another type of metadata can be used to indicate that a node is linked to some external artifact(s). More generally, we can use metadata to give provenance information, such as representing how a node was constructed (e.g., via some formal method) or tracing information (e.g., to a system or standard).

In general, metadata are meant to reflect a variety of domain knowledge. Thus, we use different domain ontologies, which capture the relevant concepts and their interrelations in a domain, to give the semantics of the attributes. For example, from a requirements ontology, we can provide attributes to goal nodes that reflect not only concepts such as *requirement*, *formal requirement*, *safety requirement*, etc., but also relations such as *formalizes* or *is allocated to*. Then, by drawing from a system organization ontology, we can add more information about the specific system, subsystem or component to which the requirement applies. In the absence of an ontology, we can rely on terminological information, such as a glossary from a standards document, or procedural guidance documents.

## 2.2 Sources of Queries

We see queries and views as a means to express, respectively, specific questions relevant for argument structure creation, review, or modification, and their responses. Potential sources of queries, besides the experience of the safety engineer or the assessor, includes domain knowledge, such as that contained in regulations, standards, guidance documents, artifact documentation, documentation for processes and procedures, etc. To illustrate, we give some scenarios:

*Supporting Safety Argument Development and Change:* When developing a safety argument for a complex system, arguments addressing all parts of the system may not all be created at the same time. These present some simple query needs, e.g., determining the claims that remain to be supported or how/if high-risk hazards have been addressed. Similarly, a developer may want to view specific fragments, e.g., how a formal method was applied to develop a claim or how a specific pattern has been instantiated. Furthermore, when redesign/replacement of some components is required, we can use queries also to identify those argument fragments that ought to be updated to reflect the revised safety analysis, and, in turn, to understand the impact of those changes on the overall safety argument.

*Addressing Traceability Concerns:* In general, demonstrating traceability is a requirement during certification, e.g., as part of the software approval process [9]. An important form of traceability is to show how requirements from regulations, standards and other relevant guidance documents are linked to the appropriate evidence items. For instance, item 5.b.(5) of the safety checklist in FAA Order 8130.34B, Appendix D [10], requires describing how software requirements are validated and the means for software verification. In addition to providing descriptive text—as is the case in practice—we believe that an informative response also could include an appropriate slice or *view* of the airworthiness assurance argument structure, showing the claims relevant to software requirements, the applicable context under which validity can be claimed, the relevant

assumptions and justifications, the strategies for verification and validation (e.g., formal verification, and inspection against domain knowledge, respectively), and how these have been applied to refine the claims made.

*Supporting Assessment and Review:* As part of the different milestones of the systems engineering process [11], engineering artifacts (as well as the safety case) are to be reviewed and accepted before development proceeds. Simple queries on the safety argument can be used to determine whether the relevant obligations have been met. For example, during a Preliminary Design Review (PDR), we can query the safety case to establish whether or not all the identified safety requirements have been allocated.

### 2.3 Components of Queries and Views

For this paper, we mainly focus on queries that operate on GSN arguments although, eventually, we want to expand the scope of queries to include the entire safety case.

Conceptually, queries in AQL comprise a combination of properties of both the semantic and the syntactic information in the argument structure. As described earlier (Section 2.1), metadata, i.e., attributes on nodes, provide a way to access the semantic information. To access the structure, AQL queries contain expressions referencing GSN syntax. The language (described subsequently in Section 3) itself consists of a selection of *atomic* queries that can be grouped with the usual logical connectives (and, or, xor, not), as well as the *path quantifiers*  $[]$  (all), and  $\langle \rangle$  (some), to specify query relations. The language also contains constructs to access structure in terms of the relative arrangement of nodes, e.g., *above*, *below*, *directly above*, etc.

Taken together, we can express some relatively complex queries in the form of concepts that AdvoCATE can understand, so that an informal, natural language query in the domain can be expressed as a formal AQL query over the GSN argument structure. Here, we note that the translation of an informal query into a formal one, and the resulting view generated, depends on the purpose of the query. For instance, consider querying for an *incomplete* argument. If the purpose were simply to locate a set of undeveloped nodes, so that the safety case author(s) can further develop them, it suffices to specify a formal query whose result is exactly the set of undeveloped nodes of interest, e.g., goals nodes marked *to be developed*. Alternatively, if the purpose were to assess, say, whether or not a claim has been supported by evidence, or the extent to which it has been developed, then a view containing greater details is more useful. Then, we can specify an appropriate formal query in AQL which will result in an argument structure view containing any goal or strategy not immediately (or eventually) followed by other goals, strategies or evidence (See Section 5 for a concrete example).

The outcome of executing a query on an argument structure is an argument structure *view*. A view is a diagram showing the fragment(s) of the (source) argument that satisfy the query. In our implementation, we collapse those nodes that do not satisfy the query into *concealment nodes* (*C*-nodes, for short), which we annotate with the number of hidden nodes. A *C*-node can be (temporarily) expanded to show the corresponding fragment in the source argument. To reduce visual clutter, by default we only show *C*-nodes that appear between two regular nodes. So, for example, if a context node does not satisfy a query, it does not appear in the view. One consequence is that if the root

node does not satisfy the query, the view will consist of several unconnected fragments, though this preference can be changed.

We allow multiple views for a given argument structure, reflecting the application of different queries. The views and the queried structure are kept consistent with each other in our implementation, so that a change in any one of the views/argument structure is either propagated to the rest or, in the case of an inconsistent change (due to independent unsaved edits, say), the user is alerted.

### 3 Foundations

#### 3.1 Metadata

Metadata is associated with individual nodes (rather than globally with the entire argument). Each node has a set of associated attributes, which are declared and can be parameterized over parameters of specific types. Nodes have instances of attributes with values that comply with the type of the parameter (which can itself depend on the node). In general, we draw these parameter values from a domain ontology (See Fig. 3 in Section 5, for an example). The grammar of an *attribute declaration* is:

```
attribute ::= attributeName param*
param ::= String | Int | Nat | nodeID | sameNodeTypeID | goalNodeID | strategyNodeID |
        evidenceNodeID | assumptionNodeID | contextNodeID | justificationNodeID |
        userDefinedEnum
```

The type of a parameter can either be:

- a basic type, i.e., a string (String), an integer (Int), or a natural number (Nat)
- a *node type*, which can be used as parameters in three different ways:
  - NodeID: any kind of node
  - sameNodeTypeID: the parameter must be the identifier of a node of the same type as the node with the attribute.
  - Specific node parameter types, which allow specification of a node of a given type: assumptionNodeID, contextNodeID, evidenceNodeID, goalNodeID, justificationNodeID, strategyNodeID.
- A user-defined enumeration (userDefinedEnum): for example, we can define the parameter types
 

```
severity ::= catastrophic | hazardous | major | minor | noSafetyEffect
likelihood ::= frequent | probable | remote | extremelyRemote |
             extremelyImprobable
```

to define the parametrized attribute risk(severity, likelihood). Then, we can give an *attribute instance* as: risk(severity(catastrophic), likelihood(extremelyImprobable)). We will just use “attribute” when it is clear from the context whether we mean attribute instance or attribute declaration. Note that we do not force the values of different enumerations to be distinct.

#### 3.2 Syntax and Semantics

**Query Syntax** Queries are defined with respect to a signature given by the declared metadata. Henceforth, we will assume that this signature is fixed, and let  $A$  range over

attribute instances and  $N$  range over node identifiers. We will also use  $\mathcal{F}$  to indicate a *node field* and write  $\mathcal{F}$  **has**  $v$  where  $\mathcal{F}$  is one of the fields *id*, *type* ( $t$ ), *description* ( $d$ ), *attributes* ( $m$ ), *status* ( $s$ ), and  $v$  is an appropriately typed concrete value.

The node identifier (*id*) and description (*description*) must be strings; node type is one of goal  $g$ , strategy  $s$ , evidence  $e$ , context  $c$ , assumption  $a$ , and justification  $j$ ; attributes takes an attribute instance( $A$ ), and status takes *tbd* (to be developed). For the fields *id* and *type*, **has** means equality; for the field *description*, **has** means sub-string, and for the fields *attributes* and *status*, **has** means set membership.

**Definition 1 (Pre-query).** A pre-query is a term constructed according to the following grammar:

$$Q ::= true \mid \mathcal{F} \text{ has } v \mid \text{isAbove} \mid \text{isBelow} \mid \text{isDirectlyAbove} \mid \text{isDirectlyBelow} \mid Q(N) \mid \text{not } Q \mid Q \text{ and } Q' \mid \langle Q \rangle Q'$$

Now we define well-formedness rules on pre-queries, which will allow us to define queries. We give these as inference rules for the *arity* of a query:

$$\frac{\overline{\mathcal{F} \text{ has } v : 1} \quad v \text{ well-typed for } \mathcal{F}}{\text{isBelow} : 2} \quad \frac{}{\text{isAbove} : 2} \quad \frac{}{\text{isDirectlyBelow} : 2} \quad \frac{}{\text{isDirectlyAbove} : 2}$$

$$\frac{Q : 2}{Q(N) : 1} \quad \frac{Q : n}{\text{not } Q : n} \quad \frac{Q : n \quad Q' : n}{Q \text{ and } Q' : n} \quad \frac{Q : 2 \quad Q' : n}{\langle Q \rangle Q' : n}$$

Here  $Q : n$  means that query  $Q$  is well-formed and represents a property of  $n$  node arguments. An inference rule states that if the hypotheses hold (i.e., the queries above the line are well-formed with the specified arities) then the conclusion holds (i.e., the stated query below the line is well-formed with given arity).

**Definition 2 (Query).** We define a query to be a pre-query  $Q$  such that  $Q : 1$  according to the pre-query well-formedness rules.

We do not need to supply all the parameters in an attribute instance in a query. For example, we can write **attributes has risk(likelihood(probable))** to mean: find a node with an attribute risk whose likelihood is probable and with *any* severity. We can abbreviate this further by writing **attributes has risk(probable)**, which will look for any parameter with value probable, or even **attributes has risk** with which to find nodes tagged with any risk values. We can omit the second argument of a top-level quantifier, in which case it is taken to be true. For example, a root node can be queried by **not(isBelow)**, which is equivalent to **not(isBelow>true)**. A derived syntax for queries is given as:

$$\begin{aligned} false &= \text{not } true \\ (\text{or}) \quad Q \text{ or } Q' &= \text{not } (\text{not } Q \text{ and } \text{not } Q') \\ (\text{xor}) \quad Q \text{ xor } Q' &= (Q \text{ and } \text{not } Q') \text{ or } (\text{not } Q \text{ and } Q') \\ (\text{all}) \quad [Q] Q' &= \text{not } \langle Q \rangle \text{not } Q' \end{aligned}$$

**Semantics of Queries** In order to give semantics to queries, we first give semantics to argument structures.

**Definition 3 (Safety Case Argument Structure).** A safety case argument structure is a 3-tuple  $\langle N, f, \rightarrow \rangle$  where  $N$  is a set of nodes;  $f_X$  (where  $X \in \{t, d, m, s\}$ ) gives the node fields: type, description, attributes, status; and  $\rightarrow$  is the connector relation between nodes. Various restrictions<sup>5</sup> must be placed on  $\rightarrow$  to ensure that an argument structure is well-formed. We have  $f_t : N \rightarrow \{s, g, e, a, j, c\}$  gives node types,  $f_d : N \rightarrow \text{string}$  gives node descriptions,  $f_m : N \rightarrow A^*$  gives node instance attributes, and  $f_s : N \rightarrow \mathcal{P}(\{tbd\})$  gives node development status.

Note that, here, we equate nodes with their identifiers. Also, it is possible to give many variants on this definition (as we have done previously [12], [13]), depending on the information that we want to associate with the argument. Here, we include all information that is relevant to the definition of the queries. Next, we give semantics to queries, as:

$$\begin{array}{ll}
N \models \text{true} & \\
N \models \text{id has } v \iff N = v & N, N' \models \text{isAbove} \iff N \rightarrow^+ N' \\
N \models \text{type has } v \iff f_t(N) = v & N, N' \models \text{isBelow} \iff N' \rightarrow^+ N \\
N \models \text{description has } v \iff v \text{ substring } f_d(N) & N, N' \models \text{isDirectlyAbove} \iff N \rightarrow N' \\
N \models \text{attributes has } v \iff v \in f_a(N) & N, N' \models \text{isDirectlyBelow} \iff N' \rightarrow N \\
N \models \text{status has } v \iff v \in f_s(N) & N \models Q(N') \iff N, N' \models Q
\end{array}$$

For compound query terms, we need to give rules for either one or two nodes. Write  $\bar{N}$  to mean either  $N_1$  or  $N_1, N_2$ . Then,

$$\begin{array}{l}
\bar{N} \models \text{not } Q \iff \bar{N} \not\models Q \\
\bar{N} \models Q \text{ and } Q' \iff \bar{N} \models Q \text{ and } \bar{N} \models Q'
\end{array}$$

For quantifiers, it is simpler to give the two cases separately:

$$\begin{array}{l}
N \models \langle Q \rangle Q' \iff \exists N' \text{ such that } N, N' \models Q \text{ and } N' \models Q' \\
N, N' \models \langle Q \rangle Q' \iff \exists N'' \text{ such that } N, N'' \models Q \text{ and } N'', N' \models Q'
\end{array}$$

**Semantics of Views** There are two (equivalent) ways to define views, depending on whether we treat concealment nodes as a special kind of node or as part of a link. Though each definition has some advantages, the simplest is to use nodes.

**Definition 4 (Argument View).** An argument view is a 5-tuple  $\langle N, \mathbb{C}, f, \gamma, \rightarrow \rangle$  where  $N$  is the set of argument nodes,  $\mathbb{C}$  is the set of  $\mathcal{C}$ -nodes,  $f$  gives node fields for  $N$ ,  $\gamma : \mathbb{C} \rightarrow \text{nat}^+$  gives  $\mathcal{C}$ -node counts, and the connector relation  $\rightarrow$  is subject to the same restrictions as in Definition 3 (that is, if  $x, x' \in N$  and  $x \rightarrow x'$ , then there are restrictions on the types of  $x, x'$  to prevent illegal links). Moreover, if  $x' \in \mathbb{C}$  then  $f_t(x) \in \{g, s\}$ ; if  $x \in \mathbb{C}$ , then  $x'$  can have any type; we cannot have both  $x, x' \in \mathbb{C}$ .

In practice, to reduce clutter, the tool allows additional restrictions to be placed when creating  $\mathcal{C}$ -nodes. For the examples in this paper (Section 5), we also require that  $\forall \gamma \in \mathbb{C}. \exists n, n' \in N. n \rightarrow \gamma \rightarrow n'$ . We can also relax this condition so that  $\mathcal{C}$ -nodes are added as root. The last condition prevents  $\mathcal{C}$ -nodes at the edge of the view, since they can only appear between regular nodes (as mentioned in Section 2.3). Next, we relate views to arguments.

<sup>5</sup> See [12] for details.

**Definition 5 (A-view).** Let  $A = \langle N_A, f_A, \rightarrow_A \rangle$  and  $V = \langle N_V, \mathbb{C}, f_V, \gamma, \rightarrow_V \rangle$  be an argument and a view, respectively. We say that  $V$  is an  $A$ -view if  $N_V \subseteq N_A$ ,  $N_A \cap \mathbb{C} = \emptyset$ ,  $f_V = f_A \upharpoonright N_V$ ,  $\rightarrow_A \upharpoonright N_V \subseteq \rightarrow_V$ , and there exist mappings  $f : N_A \rightarrow N_V \cup \mathbb{C}$  and  $g : N_V \cup \mathbb{C} \rightarrow A$  such that  $g; f = id$ , and  $f; g(x) \rightarrow_A^* x$ .

The latter condition forces the map from a  $\mathbb{C}$ -node to be to the root of a concealed sub-DAG. Note that, in general,  $f$  is partial and so, therefore, is  $f; g$ . However,  $g$  and  $g; f$  are total.

Now, we define the views which result from queries. First, we need to define those fragments of an argument which are concealed by a query. Let  $Q$  be a query and define  $S_Q = \{n \in N \mid n \not\models Q \text{ and } \exists n_1, n_2. (n_1 \rightarrow n \rightarrow n_2)\}$ . A path,  $p$ , is a loop-free sequence of connected nodes. If  $p$  connects nodes  $n$  and  $n'$  we write  $p : n \rightarrow^* n'$ . Then, define the relation  $R_Q$  as  $n R_Q n' \iff \forall p : n \rightarrow^* n'. \forall n'' \in p. n'' \not\models Q$ .

$R$  relates nodes which are in the same concealed fragment. It is easily seen that  $R_Q$  is an equivalence relation, and so we can form the partition  $S_Q \setminus R_Q$ , i.e., the set of concealed fragments.

**Definition 6 (Q-view).** Given argument  $A = \langle N, f, \rightarrow \rangle$ , and query  $Q$ , we define the  $Q$ -view of  $A$  as  $\langle N_v, \mathbb{C}, f_v, \gamma, \rightarrow_v \rangle$ , where the components are defined as follows:

- (a)  $N_v = \{n \in N \mid n \not\models Q\}$
- (b) Let  $S_Q \setminus R_Q = \{H_1, \dots, H_m\}$ , and define  $C$  as a fresh set of elements  $\{c_1, \dots, c_m\}$ .
- (c)  $f_v = f \upharpoonright N_v$
- (d)  $c(c_i) = |H_i|$
- (e)

$$n \rightarrow_v n' \iff \begin{cases} n, n' \in N & \text{and } n \rightarrow n', \text{ or} \\ n \in N, n' = c_i \in C & \text{and } \exists n'' \in H_i. n \rightarrow n'', \text{ or} \\ n = c_i \in C, n' \in N & \text{and } \exists n'' \in H_i. n'' \rightarrow n' \end{cases}$$

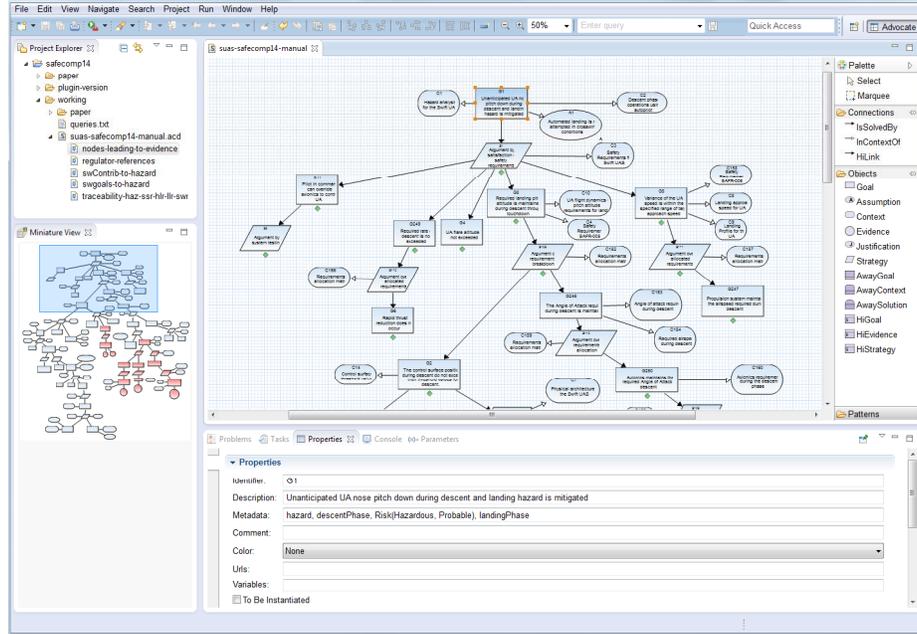
We now state (without proof) that queries give rise to well-formed views. Recall that we assume that queries and arguments are defined over a common attribute signature.

**Theorem 1.** Let  $Q$  and  $A$  be a query and argument, respectively. Then, the  $Q$ -view of  $A$  is an  $A$ -view.

## 4 Implementation

We have implemented the query/view mechanism in our toolset, AdvoCATE [8]. The tool stores the views associated with a diagram as special properties of the diagram, in particular as two lists in the diagram file itself: (a) all the view names associated with the diagram, and, (b) correspondingly, the query that maps to each name. In the interface, views appear by name as sub-items in the project explorer, under the corresponding diagram, e.g., as shown in Fig. 2. The figure also shows how node attributes are displayed (in the properties panel) along with other node fields. We draw the attributes from a domain-specific grammar, an excerpt of which is given in Fig. 3.

Although not shown in Fig. 2, we have implemented some additional usability features, such as the ability to open multiple views simultaneously in separate *tabs*, i.e., multiple canvases. The end-user can save changes either to the argument structures,



**Fig. 2.** Screenshot of AdvocaTE: (a) queries appear as sub-items of the argument structure file, in the project explorer panel on the left (b) the canvas is used to create GSN argument structures using the palette on the right, which provides the different GSN nodes and links (c) queries are run by entering them in the query text-box in the toolbar.

the queries, or both. Users will also be shown the current query, and can edit it further before saving. When any change is made either to the source diagram or a view, it is reflected in all views and the original diagram.

Due to space limitations, we only briefly describe the algorithm underlying the query/view mechanism. Let  $A$  be an argument structure,  $Q$  be a well-formed query,  $N$  be a node in  $A$ , and  $\tau$  be a table of query results. If  $\tau$  contains the result of applying  $Q$  to  $A$  then, using the function  $\text{computeView}(A, Q)$ , create a view as the conjunction of the nodes and links in  $\tau$ . Then, according to the restrictions of Definition 4, create and link  $C$ -nodes to hide all nodes in  $A$  absent in  $\tau$ . Otherwise use the function  $\text{satisfiesQuery}(A, Q, N)$  on all nodes of  $A$  to locate those nodes that satisfy  $Q$ , store the result to  $\tau$ , and call  $\text{computeView}(A, Q)$  to create the view as earlier. The  $\text{satisfiesQuery}(A, Q, N)$  function recursively evaluates the syntax tree of  $Q$ , iteratively locating the nodes in  $A$  such that the function returns *true*. We now state the correctness of the algorithm (without proof) as:

If  $\text{satisfiesQuery}(A, Q, N)$  returns *true* then  $N \models Q$     and  
 If  $\text{computeView}(A, Q) = V$  then  $V$  is the  $Q$ -view of  $A$

Query execution is reasonably fast, taking under a second to process large diagrams containing upwards of 500 nodes.

```

requirement(id, hierarchyLevel, assuranceConcern)
formalClaim(id), informalClaim(id), hazard(id)
id ::= int | string
hierarchyLevel ::= highLevel | lowLevel
assuranceConcern ::= functional | safety | reliability | availability | maintenance
requirementAppliesTo(elementLevel, elementType, element)
elementLevel ::= system | subsystem | component | module | function | model | signal
elementType ::= hardware | software
element ::= aileron | elevator | flaps | propulsionBattery | avionicsBattery | actuatorBattery |
           avionics | autopilot | FMS | AP | aileronPIDController | elevatorPIDController |
           propulsion | engine | propeller | engineMotorController | actuator |
           flightComputer | wing | actuatorMotorController pilotReceiver | IMU
references(variable)
variable ::= aileronValue | pitchAttitude | flareAltitude | vRef | vNE | thrust | vS1
regulation(part)
part ::= 14CFR23.73 | 14CFR23.75
risk(severity, likelihood)
severity ::= catastrophic | hazardous | major | minor | noSafetyEffect
likelihood ::= frequent | probable | remote | extremelyRemote | extremelyImprobable
isFormalizedBy(sameNodeTypeID)

```

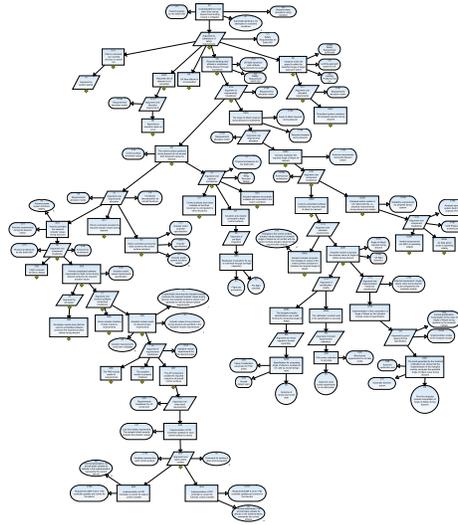
**Fig. 3.** Excerpt of domain specific grammar for metadata.

## 5 Application

We illustrate our query mechanism and its utility by application to a fragment of the Swift UAS safety argument (See Fig. 4a for a bird’s eye view): in particular, we describe some queries based on the motivating scenarios described earlier (Section 2.2) and show the resulting views. The argument structure (in Fig. 4a) concerns, in brief, the mitigation of a specific safety hazard—*unanticipated nose pitch down during descent and landing*—that can result in a loss of the aircraft and damage to the runway. The argument develops the root claim of hazard mitigation into sub-claims concerning the various contributory system functions, including software/hardware, components, and operations, which are then linked to the evidence, e.g., available from experimental data, procedures, and verification activities. In preparation for querying the argument, we added metadata to the nodes using user-defined enumerations (see Section 3) and a domain-specific grammar (Fig. 3).

Requirements address an assurance concern at a particular level of hierarchy, and can be applied to system elements of various types. As motivated earlier, (Section 2.2), an assessor might want to examine whether traceability exists from hazards to all relevant system safety requirements, high-level and low-level requirements, down to software requirements. We can specify such a traceability query in AQL in a straightforward way, as shown in Fig. 4b. As mentioned earlier (Section 3.2), some of the parameters of the metadata can be omitted in the query. The resulting (bird’s eye) view (Fig. 4c) contains goal nodes with metadata about the hazard and requirements to which they are related. These goal nodes, in turn, are linked using  $\mathcal{C}$ -nodes. Fig. 4d shows the top right leg of the view (Fig. 4c), showing traceability from a high-level requirement on the avionics system, to the high-level and low-level avionics software requirements relevant for the mitigation of the descent phase hazard.

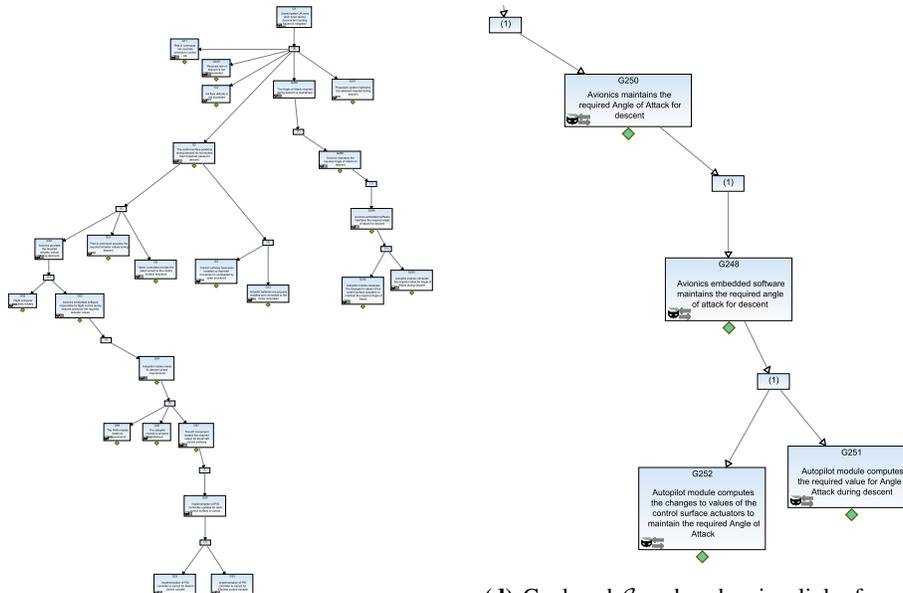
Note that we can create the view shown in Fig. 4d by constraining the traceability query, e.g., by including attributes about the avionics software. We can further constrain the query to only consider hazards with a certain risk level. For example, by including



(a) Bird's eye view of a fragment of the Swift UAS safety case in GSN.

type **has** goal and (attributes **has** hazard or (attributes **has** requirement(safety) and attributes **has** requirementAppliesTo(system)) or attributes **has** requirement(highLevel) or attributes **has** requirement(lowLevel) or attributes **has** requirementAppliesTo(software))

(b) AQL traceability query.



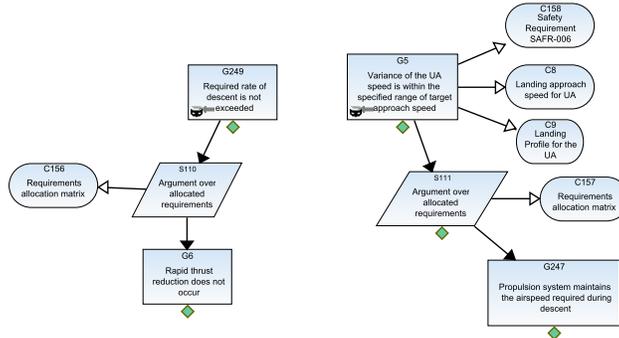
(c) Bird's eye view showing the result of a traceability query applied to Fig. 4a.

(d) Goal and C-nodes showing links from a high-level requirement to software requirements for the avionics.

**Fig. 4.** GSN argument fragment for the Swift UAS, AQL query and the resulting view.

(type has goal) and (attributes has regulation(14CFR23.73) or attributes has regulation(14CFR23.75)) or <isBelow>(attributes has regulation(14CFR23.73) or attributes has regulation(14CFR23.75))

(a) Query in AQL to locate references to regulatory requirements.

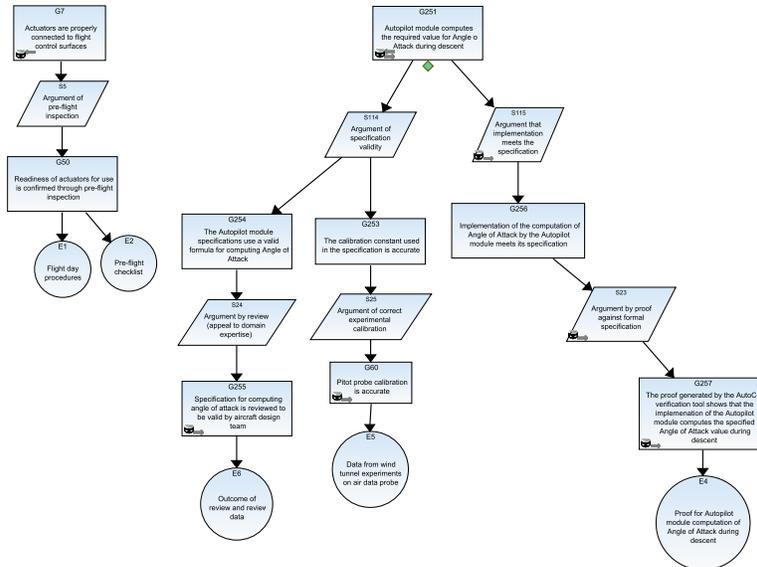


(b) View resulting from the query in Fig. 5a, showing disconnected argument structure fragments.

**Fig. 5.** AQL query and view showing those parts of the argument fragment of Fig. 4a referencing regulatory requirements.

[isAbove] (not (type has goal or type has strategy) or <isAbove>(type has goal or type has strategy or type has evidence)) and (<isAbove> type has evidence or type has evidence)

(a) AQL query using only structural references.



(b) View produced by applying the query in Fig. 6a to the fragment in Fig. 4a.

**Fig. 6.** Query and View: All nodes from which all paths lead to evidence.

the AQL expression attributes **has** risk(severity(catastrophic), likelihood(remote)) in the query of Fig. 4b, we can generate a view (not shown here) of traceability to only those hazards whose likelihood of occurrence is remote, and whose severity is catastrophic.

Another query on the argument structure can be to identify those parts that address concerns from regulations, standards, and guidance documents. For example, Part 23 of the Federal Aviation Regulations (FARs) specifies requirements concerning aircraft performance during landing, e.g., approach speeds (14 CFR §23.73), the conditions to be met for accomplishing a safe landing within the required landing distance (14 CFR §23.75), etc. To formulate an appropriate query, first we locate all goal nodes with the attributes `regulation(14CFR23.73)` or `regulation(14CFR23.75)`, using the grammar of Fig. 3. Then, to determine the extent to which the corresponding claims have been addressed in the argument, we locate those fragments whose roots are the located goal nodes and show the entire structure to highlight the relevant context, assumptions and justifications, if any, and the reasoning used. Fig. 5a and Fig. 5b show the relevant AQL query and its corresponding view respectively. From the latter, we can infer that there are claims in the structure that reference the regulatory requirements, but that they are yet to be fully developed. To determine the exact extent of how the regulations are met, an assessor could navigate to, and examine, the external documentation referenced from the nodes shown in the view.

Thus far, our queries have shown how we use simple combinations of structure and metadata to produce views that address domain specific scenarios: namely, establishing if and how some regulatory requirements have been addressed, and showing the traceability concerns that may be required by assurance standards. However, we can also use AQL to specify more complex queries that produce meaningful views and operate on the structure alone. One such example concerns querying for those fragments which are *completely developed*, i.e., all nodes from which all paths lead to an evidence node.

In fact, this query gives a way to determine the *internal completeness* of an argument structure from a purely structural standpoint. That is, the property that—assuming valid reasoning from premises to conclusions, and not considering the confidence needed to accept a claim/argument—there exists no claim (i.e., goal node) in the argument such that a path from it does not end in evidence. We specify this query in AQL as given in Fig. 6a, and the resulting view is shown in Fig. 6b. Thus, an argument structure that is identical to the view produced by applying this query is internally complete.

To understand this query (Fig. 6a), we include some basic notions for explanation purposes: An *end node* here is a goal or strategy with no goals or strategies beneath it, effectively making it the end of an *is supported by* chain. A *middle node* here is a node with goals or strategies beneath it; since only goals or strategies satisfy this condition, all middle nodes are goals or strategies). There are three types of nodes the query seeks to find: (i) end nodes that have some evidence node beneath them; (ii) middle nodes that only have other middle nodes and nodes of type (i) beneath them; (iii) the evidence nodes at the end of the argument. To express these three possibilities, we combine the three using the `or` operator, and simplify. Note that the simplification includes facts (expressed in AQL), e.g., an evidence node can never be above a goal.

Queries can also be used to identify parts of argument that, though complete, might not engender sufficient confidence. For example, goals associated with high risk may

need to be supported by particular forms of evidence. We can then use an appropriate query to identify those argument fragments that do not meet these criteria.

## 6 Concluding Remarks

We have described a methodology and a formal foundation to query safety cases. Specifically, we have described how to enrich GSN argument structures with domain-specific metadata, and how to produce argument structure views by querying arguments using the Argument Query Language (AQL). We have implemented and tested a prototype of our query/view mechanism in our toolset, AdvoCATE. Using a fragment of the Swift UAS safety case argument structure as a driving example, we have demonstrated the creation of a simple set of domain concepts, its use in querying an argument, and how queries can produce specific perspectives on that argument.

The closest counterpart to our work proposes *multi-view safety cases* [14] and also operates on GSN argument structures. Here, a view is produced from, effectively, an *a priori* encoding of the elements of the argument that correspond to a specific, static, stakeholder viewpoint, e.g., a process view. In contrast, our notion of view is dynamic since it is determined upon evaluating the query applied to the argument structure. Queries have been used in safety-critical applications by [15], wherein *visual queries* are applied to *traceability information models* to show traceability. Our work is comparatively much broader in scope and considers a variety of queries (Section 2.2) including, and in addition to, traceability in safety assurance. For instance, a useful perspective to present during software approval would be showing, say, only the software aspects in an argument, or the software contributions to different hazards. We can specify these kinds of queries and generate the relevant views in a straightforward way.

Query languages exist for a variety of frameworks, e.g., databases, knowledge bases, ontologies, etc. For example, SQWRL [16] is an *ontology query language*, which offers richer logical expression than AQL (currently) does, but is more generic. PrQL [17] is a specialized *proof query language* with some similarities to AQL, though it targets a different domain. The data comprising a safety case, potentially, can be organized into a (relational) database and then queried. However, we are unaware of approaches/tools that either query argument structures in this manner, or are similar to ours.

We believe that our approach for querying safety cases can be useful to address stakeholder-specific concerns, and can help in argument comprehension by locating and displaying the relevant information of interest. However, we can do more to further improve the practical usefulness of our approach. For instance, currently we specify attributes through an interface in which the end-user relies on an external ontology or glossary of terms. We plan to integrate an ontology tool and *import* the relevant ontologies. In addition to enriching the underlying domain theory, we can enhance the query language in several ways: First, we can make several simple extensions to the atomic predicates on which the query language is built, e.g., distinguishing the different link types, *in context of* and *is supported by*. Next, a more significant extension would be the implementation of named queries, i.e., allowing queries (as opposed to views) to be saved to a library and referenced by name within other queries. This would greatly simplify the use of larger, more complex queries. We will also provide a range of use-

ful library queries by default, e.g., finding undeveloped nodes. The current query/view mechanism is limited to the core GSN, and it works primarily on the argument structure. We intend to develop suitable interfaces to linked artifacts, which will allow us to query the entire assembly of artifacts comprising a safety case.

**Acknowledgement.** This work has been funded by the Assurance of Flight-Critical Systems element of the SSAT project in the Aviation Safety Program of NASA ARMD.

## References

1. UK Ministry of Defence (MOD): The ‘White Booklet’: An Introduction to System Safety Management in the MOD. Issue 3 (January 2011)
2. Hawkins, R., Habli, I., Kelly, T.: Principled Construction of Software Safety Cases. In: 2013 SAFECOMP Workshops–Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR). (September 2013)
3. Goal Structuring Notation Working Group: GSN Community Standard Version 1. (2011)
4. Denney, E., Habli, I., Pai, G.: Perspectives on Software Safety Case Development for Unmanned Aircraft. In: Proc. 42nd Annual IEEE/IFIP Intl. Conf. Dependable Systems and Networks (DSN 2012) pp. 1–8 (June 2012)
5. Bloomfield, R., Chozos, N., Embrey, D., Henderson, J., Kelly, T., Koornneef, F., Pasquini, A., Pozzi, S., Sujan, M., Cleland, G., Habli, I., Medhurst, J.: Evidence: Using Safety Cases in Industry and Healthcare. The Health Foundation (December 2012)
6. Denney, E., Pai, G., Pohl, J.: Heterogeneous Aviation Safety Cases: Integrating the Formal and the Non-formal. In: 17th IEEE Intl. Conf. Engineering of Complex Computer Systems (ICECCS) pp. 199–208 (July 2012)
7. EUROCONTROL: Preliminary Safety Case for ADS-B Airport Surface Surveillance Application. PSC ADS-B-APT (November 2011)
8. Denney, E., Pai, G., Pohl, J.: AdvocATE: An Assurance Case Automation Toolset. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012 Workshops. LNCS, vol. 7613, Springer (September 2012)
9. FAA: Software Approval Guidelines. FAA Order 8110.49 Chg 1 (September 2011)
10. U.S. Dept. of Transportation, FAA: Airworthiness Certification of Unmanned Aircraft Systems and Optionally Piloted Aircraft. FAA Order 8130.34B (November 2011)
11. Denney, E., Ippolito, C., Lee, R., Pai, G.: An Integrated Safety and Systems Engineering Methodology for Small Unmanned Aircraft Systems. In: Infotech@Aerospace. AIAA 2012-2572. (June 2012)
12. Denney, E., Pai, G.: A Formal Basis for Safety Case Patterns. In: Bitsch, F., Guiochet, J., Kaniche, M. (eds.) Computer Safety, Reliability and Security (SAFECOMP 2013). LNCS, vol. 8153, pp. 21–32 Springer (September 2013)
13. Denney, E., Pai, G., Whiteside, I.: Hierarchical Safety Cases. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods. LNCS, vol. 7871, pp. 478–483 Springer (May 2013)
14. Flood, M., Habli, I.: Multi-view Safety Cases. In: 6th IET Intl. Conf. System Safety. pp. 1–6 (September 2011)
15. Maeder, P., Jones, P.L., Zhang, Y., Cleland-Huang, J.: Strategic Traceability for Safety-critical Projects. IEEE Software 30(3), pp. 58–66 (2013)
16. O’Connor, M., Das, A.: SQWRL: A query language for OWL. In: Proc. 6th International Workshop on OWL: Experiences and Directions (OWLED 2009). (2009)
17. Aspinall, D., Denney, E., Lüth, C.: Querying proofs. In: 18th Intl. Conf. Logic for Programming Artificial Intelligence and Reasoning (2012)