AdvoCATE: An Assurance Case Automation Toolset

Ewen Denney, Ganesh Pai, and Josef Pohl

SGT / NASA Ames Research Center
Moffett Field, CA 94035, USA
{ewen.denney,ganesh.pai,josef.pohl}@nasa.gov

Abstract. We present **A**dvo**CATE**, an **A**ssurance **C**ase **A**utomation **T**ools**E**t, to support the automated construction and assessment of safety cases. In addition to manual creation and editing, it has a growing suite of automated features. In this paper, we highlight its capabilities for (i) inclusion of specific metadata, (ii) translation to and from various formats, including those of other widely used safety case tools, (iii) composition, with auto-generated safety case fragments, and (iv) computation of safety case metrics which, we believe, will provide a transparent, quantitative basis for assessment of the state of a safety case as it evolves. The tool primarily supports the Goal Structuring Notation (GSN), is compliant with the GSN Community Standard Version 1, and the Object Modeling Group Argumentation Metamodel (OMG ARM).

Keywords: Assurance cases, Safety cases, Metrics, Safety management, Safety process, Safety toolset, Formal methods.

1 Introduction

Structured, evidence-based arguments are increasingly being adopted as a means for assurance, e.g., as dependability or assurance cases [15], and more popularly as *safety cases* [18], for safety assurance in several domains including automotive, medical devices, and aviation. Safety cases have already been in use for some time in the defense, rail, and oil & gas sectors. The practitioner has a broad choice of tools, e.g., [1, 13, 14, 17], to use in creating structured safety assurance arguments (manually) in a variety of notations such as the Goal Structuring Notation (GSN) [10], and the Claims Argument Evidence (CAE) notation. This is, by no means, a comprehensive list of available safety case construction tools, each of which have different foci, e.g., linking to type theory, use of different notations for graphical representation of safety cases, etc. However, common to all the tools is manual safety case creation with limited support for auto-generation or automatic assembly. Creating safety cases manually can be time consuming and costly.

Our goal is to develop a framework for the automated creation and assembly of assurance cases, using model-based transformation. In particular, we want to (i) leverage our earlier work on using the output of formal methods to create auto-generated safety case fragments [3], and additionally (ii) automatically combine them with the results of traditional safety analyses, also transforming these into safety case fragments [8]. The latter is aimed at supporting lightweight, automatic assembly and integration of safety

F. Ortmeier and P. Daniel (Eds.): SAFECOMP 2012 Workshops, LNCS 7613, pp. 8–21, 2012.

cases into traditional safety, and development processes [6]. We aim to support the more general notion of assurance cases, although in this paper we focus on safety cases.

We present AdvoCATE, the **A**ssurance **C**ase **A**utomation **T**ools**E**t, a suite of tools and applications based on the Eclipse platform[1], to build and transform safety cases. The core of the system is a graphical safety case editor, integrated with a set of model-based transformations that provide functionality for translating and merging pre-existing safety cases from other formats, and for incorporating automatically generated content from external formal verification tools. The tool metamodel (Section 2.1) extends the GSN, e.g., through the inclusion of metadata (Section 2.2). The tool (Section 3) supports basic manual creation and editing (Section 4.1), and interoperability with other safety case tool formats (Section 4.2). The metadata supports automation in safety case creation, for assembly of safety case fragments that have themselves been auto-generated using formal methods (Section 5), generation of safety case metrics (Section 6), and transformations to generate "to-do" lists, textual narratives, and tabular representations (Section 7).

We are using AdvoCATE in the ongoing construction of a safety case for the Swift Unmanned Aircraft System (UAS), under development at NASA Ames.

2 Extended Goal Structuring Notation

2.1 Metamodel

In AdvoCATE, we have defined and implemented an Extended Goal Structuring Notation (EGSN) metamodel, to extend "traditional" GSN with additional information, e.g., node metadata, to be used to define more features and operations. The EGSN metamodel has been developed as a combination of several different safety and assurance case models; it is compatible both with the GSN standard [10], and the Argumentation Metamodel (ARM)[2], from the Object Management Group. There is a mapping from any EGSN-based model to ARM, and vice versa; the major difference is that EGSN explicitly contains all of the standard constructs, and only the two relationships as defined in the GSN. This is, of course, extensible and extra relationships and/or reasoning elements (in ARM terminology), can be added as needed.

The top level of any safety case model based on the EGSN metamodel (Fig. 1) is a *SafetyCase* element. Essentially, this is the container that holds all elements of the safety case; it has no attributes, and children of *SafetyCase* can be concrete instances of either of the the abstract elements *Node* or *Link*. A *Node* generalizes the different types of GSN elements, i.e., *Goal*, *Strategy*, *Assumption*, *Justification*, *Context*, and *Evidence*[3]. The attributes of a *Node* are:

- *identifier*, which holds a unique name for a given node.
- *description*, which is user-supplied content describing/defining the node.

[1] `http://www.eclipse.org`

[2] GSN is itself in compliance with ARM, which is available at: `http://www.omg.org/spec/ARM/`

[3] Strictly speaking, the GSN uses the term *Solution*. We use the term *Evidence* interchangeably with *Solution*.

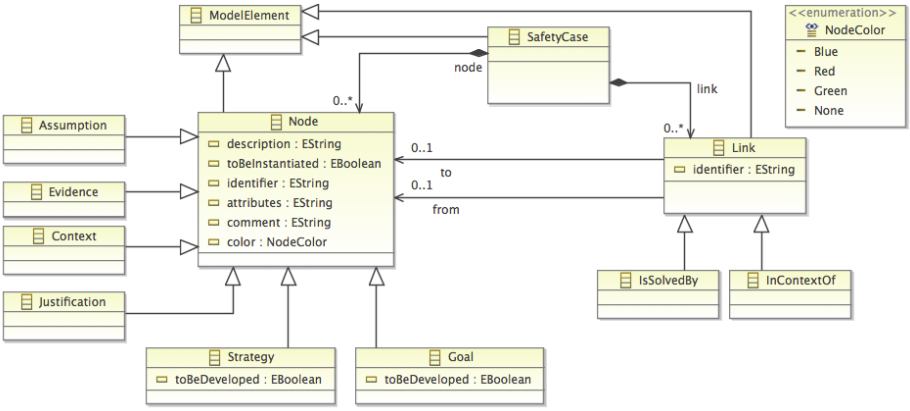

Fig. 1. EGSN metamodel in UML, representing model elements, attributes and relationships

- *color*, an attribute which is meant to indicate the color used for display; informally, we use it to convey the relative importance, source, or node state.
- *comment*, which we use to give informal information about a node.
- *toBeInstantiated*, which we use to denote abstract GSN elements that require further instantiation of specific content within the *description*, and
- *attributes*, which are used to hold extra metadata about the node, e.g., classification of the node as a high- or low-level requirement, and merging points with auto-generated content.

We can modify the attributes (above) as required and they are inherited by each node specialization. The *Goal* and *Strategy* elements also contain an additional attribute, *toBeDeveloped*, which denotes that the elements are yet to be developed, e.g., by using strategies to connect them to sub-goals/solutions. The GSN standard limits applying the *toBeDeveloped* annotation to only *Goal* and *Strategy* elements, whereas the attribute *toBeInstantiated* applies to any node.

A *Link* is also an abstract entity and contains the attribute *identifier*. Links have containment relationships, which relate the *Node*, *from* which the link comes, and *to* which it goes. These relationships refer to the abstract entity *Node* and not directly to the derived entities. The *InContextOf* link, represents a one-way association between the *Goal* (*Strategy*) element, and the *Context*, *Assumption* or *Justification* elements respectively; the *IsSolvedBy* link, denotes a one-way association between *Goal*, *Strategy*, and *Evidence* elements.

2.2 Node Metadata

We tag nodes with metadata to convey meaning about the significance or provenance of particular nodes in a safety case, such as whether they relate to the mitigation of a specific hazard, or whether they represent requirements that can be formally verified using external tools.

Node metadata is expressed as a set of attributes associated with each node. We use metadata to define transformations on the safety case and during metrics computation. At present, we have a pre-defined list of attributes that may be used. Eventually, this will be replaced with a user-definable dictionary of attributes based on an ontology. There is a strict syntax for defining attributes, as below, and multiple attributes are comma-separated.

1. High-level and Low-level requirements
   - High-Level Requirement
   - Low-Level Requirement
2. Risks
   - Risk[Likelihood,Severity]

   where
   - Likelihood ::= Extremely Improbable | Extremely Remote | Remote | Probable
   - Severity ::= Catastrophic | Hazardous | Major | Minor | No Safety Effect
3. Hazards
   - Hazard[Identifier]

   where identifier is a string giving a reference identifier in a hazard table.
4. Provenance
   - autocert:n

   where n is a number giving an AUTOCERT [9] requirement. The auto-generated fragment produced by verifying the formal requirement number n, will be merged into the safety case at this node (described in more detail in Section 5).

The tool has been designed so that different attributes affect the display (color) of the nodes, e.g., the attributes *Risk*, *Hazard*, and *Requirement* affect node color. The idea is to provide a visual indicator to the user to convey specific semantics.

For instance, for requirement attributes, *High-Level Requirement* will assign a red node color, whereas *Low-Level Requirement* will assign a green color. The *Hazard* attribute will turn a node red as well. For the *Risk* attribute, the color scheme is dependent on the combination of *Severity* and *Likelihood*, and is based on a risk categorization matrix, e.g., such as the one defined in [19]. Node color will turn red, green, or remain blue depending on whether the risk region in the risk categorization matrix is *high*, *medium*, or *low*. Once a color has been set by an attribute, it cannot be manually changed. In the case of multiple attributes, the color set by the first attribute takes precedence. The rules used to determine node colors are currently hard-coded, but we plan to make it user-definable in future.

3 Tool Chain Architecture and Implementation

In this section, we briefly describe the different frameworks and components that comprise the AdvoCATE tool chain (Fig. 2), and their integration.

Eclipse. AdvoCATE is distributed as a set of plug-ins to the Eclipse platform. Eclipse uses a number of utilities of the underlying frameworks, namely the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). AdvoCATE

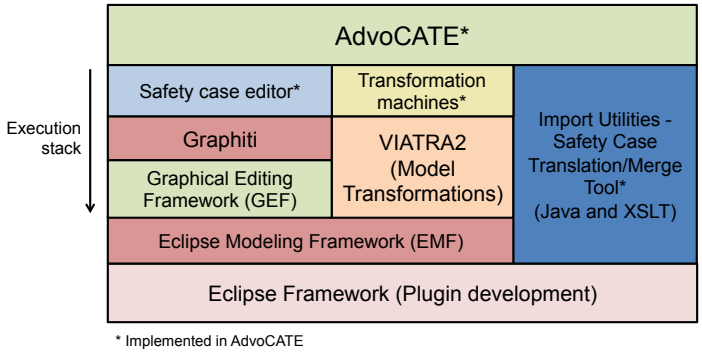

Fig. 2. Frameworks in the AdvoCATE tool chain architecture

uses the generated EMF editing tools. In principle, we could provide *extension points*[4] to extend AdvoCATE as well.

Graphical Editing. The graphical component, at the core of the safety case editor, permits the addition and manipulation of elements of a safety case. It also provides a visual representation of the relationships between the safety case model elements. The safety case model created is maintained as a separate resource from its visual representation, and the diagram. In this way, the model can be used and manipulated separately without affecting the graphical representation. Similarly, none of the information of the graphical representation affects the model except when explicitly specified, e.g., *color* can be stored as part of the properties. The two files are combined to create the diagram that is rendered on the screen and is editable by the user. Both representations are contained inside an Eclipse project. The model data file is connected to the Ecore metamodel, i.e., it must be a well-formed representation of that metamodel.

Graphiti. We built the graphical component in the Graphiti framework, an application interface (API) built upon the GEF. As shown in Fig. 2, the GEF is itself built on top of EMF. Graphiti simplifies the development of graphical tools for editing and displaying models, by automating much of the low-level implementation used to manipulate graphical objects such as rendering, moving, selecting, etc.

Translation. The tool uses XSLT to convert external data into the appropriate XML format (such as the AUTOCERT-generated XML, Section 5), which can be merged with a pre-existing assurance case. The file formats for assurance cases developed in other tools, such as ASCE, are parsed using Java DOM XML libraries.

VIATRA2. *VIATRA*2 (*VI*sual *A*utomated model *TRA*nsformations) [20], a project developed within the Generative Modeling Technology (GMT) framework, is a toolset designed for engineering life-cycle support from specification to maintenance. In the scope of AdvoCATE, it is used to hold intermediate model representations (such

[4] Plug-ins typically will provide extension points, by connecting to any of which we gain access to their functionality, e.g., the context menus and diagram creation utilities are extended from the core Eclipse user interface.

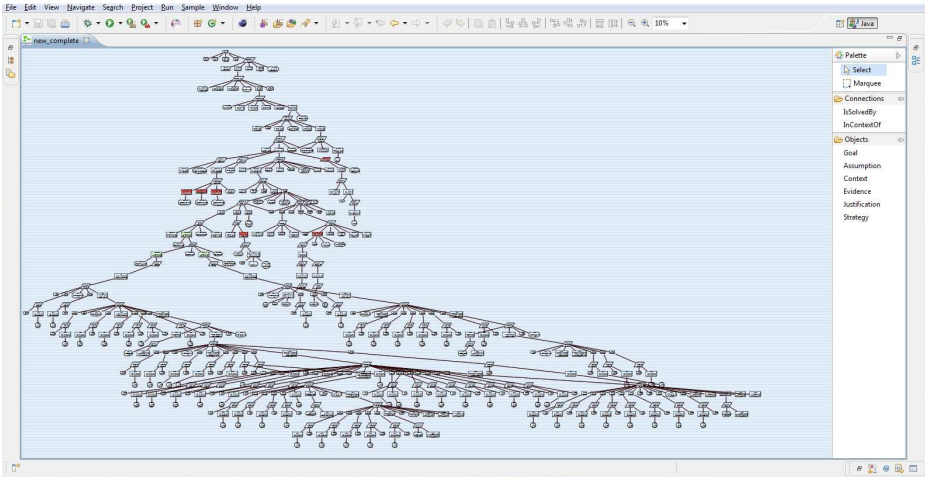

Fig. 3. AdvoCATE screenshot displaying auto-layout on the auto-assembled Swift UAS safety case fragment, which contains both manually created and auto-generated fragments

as the EGSN Ecore representation) and enact transformations on those models. Through the transformation system, we can manipulate and transform safety case models into other models (such as text, a CSV table, or a modified safety case).

4 Basic Functionality and Interoperability

Although the primary goal of AdvoCATE is to support automation, it also contains the basic manual functionality that one would expect from a safety case tool, i.e., creating and editing. In this section, we describe this manual functionality, giving an overview of some basic use-cases for the tool: to create a new safety case diagram/model, and to open a pre-existing model (as a diagram) for further editing or manipulation. Additional basic functionality includes saving, printing, translating from/to other formats, and merging external data.

4.1 Creating and Editing Safety Cases

Fig. 3 shows a bird's eye view of the structure of an end-to-end safety case fragment for the Swift UAS. It has been automatically assembled/composed from a manually created fragment and an auto-generated fragment, after which it underwent auto-layout.

Editing a diagram typically takes place within a more detailed view (Fig. 4) that shows more node information, as well as the editing features. We can select, move and resize nodes as required; node descriptions are editable either directly on the canvas, or in the properties panel, whereas attributes are edited only via the latter. Edits are reflected in the diagram in real-time, through automatic refresh. In Fig. 4, the canvas shows that the goal with identifier N48087573 (at the top right of the canvas) is selected and being edited. The properties panel beneath the canvas shows the corresponding

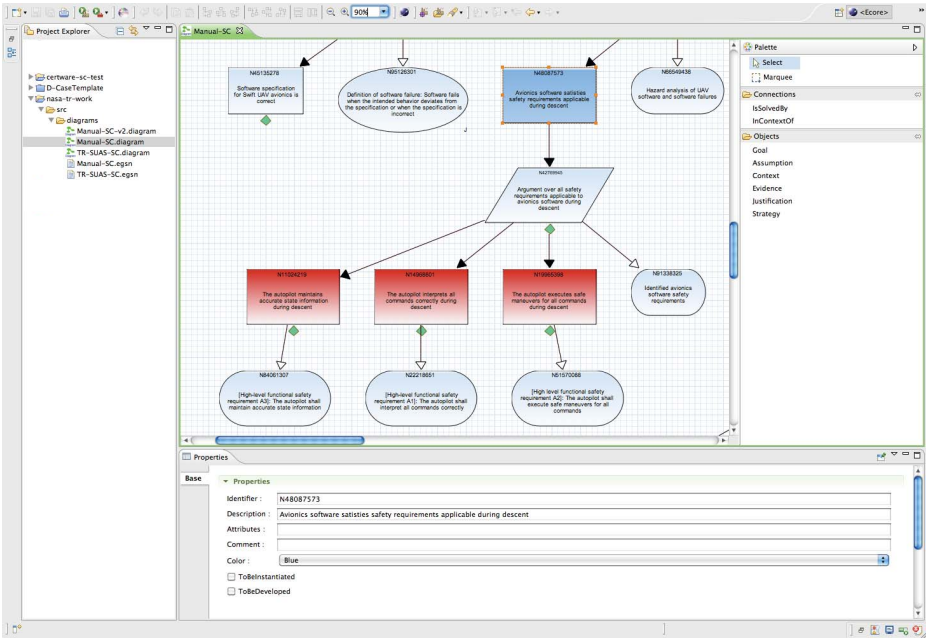

Fig. 4. AdvoCATE screenshot showing a zoomed-in view for editing, with a *Properties* panel underneath the canvas, the *Project explorer* as the left panel, and the *Palette* containing EGSN constructs in the right panel

attributes that can be edited, and we can add new values directly, as required. Attribute editing uses specialized syntax (Section 2.2) to include pre-defined node metadata.

Safety cases can be split up into separate interconnected diagrams using the *Goal Developed Elsewhere* symbol ([16], p. 66). Note that this does not provide true modularity in the sense of the modular GSN notation [10], which we do not yet support, but it does make large safety cases more manageable.

We can link to other documents such as webpages, spreadsheets, or text documents. These documents can either be local to the system or remotely stored, e.g., on a web server. We provide a specific syntax to make references to external documentation: in the *description* attribute for a node, the reference to the external document is specified as a fully qualified URL in the properties panel. The resource will be displayed in a web browser or the user will be prompted to open/save the resource.

Diagrams can be exported as an image in scalable vector graphics (SVG) format, and subsequently converted[5] to portable document format (PDF).

4.2 Interoperability

AdvoCATE supports the import/export of a variety of safety case formats—currently those produced from the ASCE [1], CertWare [13], and D-Case [14] tools. A translation

[5] Using the Batik toolkit: `http://xmlgraphics.apache.org/batik/`

engine acts as an import/export utility translating file formats from these tools into EGSN, and vice versa. If an EGSN file already exists it can be imported directly.

The translation works by using ARM as the interoperability metamodel, i.e., there are bidirectional translations between ARM and the different safety case formats. Consequently we only need to define a translation from each format to/from ARM rather than defining point-to-point connections between each tool. The ARM is also convenient for merging external information.

One of the challenges in model transformation, as between EGSN and other safety case formats, is that each metamodel has different attributes, and sometimes differing model elements. To preserve information between translations, we annotate the information as comments in the EGSN metamodel. The annotations indicate the source metamodel and what the information actually represents. For instance, in EGSN, we label nodes with a user-modifiable identifier. In ASCE, there is both a unique "reference" and a user specified identifier. If, in ASCE, the user specified identifier is not provided, the unique reference is used instead. This information is preserved by storing it in the EGSN metamodel, tagging it as being from ASCE and by labeling it as the "user-id". This way, if translating back to ASCE, the information is preserved and the ASCE model can be regenerated without information loss. There are a number of such cases which we handle in a similar way.

The one exception to this is layout data; in most cases, the model is stored separately from the layout. As a design decision, we decided not to preserve layout information.

5 Automated Assembly

AdvoCATE can automatically assemble safety cases by combining manually created fragments with content produced by external tools. Currently, this is limited to the formal verification tool AUTOCERT [9], though we plan further tool integrations in future.

Rather than perform formal verification itself, AdvoCATE integrates results from formal verification or formal methods with safety case construction. In general, there are two ways to achieve this: (i) the output of a tool can produce evidence or, depending on the level of detail it provides, be transformed into an actual argument fragment of a safety case [8], and (ii) safety case fragments can be transformed into formal specifications that are then input to a tool.

An AUTOCERT specification formalizes software requirements that we derive from system safety requirements, during safety analysis. Formal verification takes place in the context of a logical domain theory, i.e., a set of axioms and function specifications. To verify the software, we use formal verification of the implementation against a mathematical specification and test low-level library functions against their specifications.

5.1 From Formal Proofs to Safety Cases

AUTOCERT generates a document (in XML) with information describing the formal verification of requirements. The core of this is the chain of information relating requirements back to assumptions.

Each step is described by (i) an annotation schema for the definition of a program variable [4], (ii) the associated verification conditions (VCs) that must be shown for

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

During the process of merging the manually created and the auto-generated safety cases, we replace specific nodes of the manually created safety case with the tree fragments generated from AUTOCERT; specifically, the top-level goals of the latter are grafted onto the appropriate lowest-level nodes of the former. These nodes are denoted with unique attributes, `autocert:n`, relating the node to a tree in the automatically created file, meaning that the goal with tag `n` is to be solved with AUTOCERT. Additionally, these nodes are formal equivalents of informally stated goals, developed through an explicit strategy of formalization, though the formalization at this stage is both performed and checked manually.

5.2 From Safety Cases to Formal Specifications

Often, a safety case fragment may be created before the software verification is completed. In this case, we can use the `autocert:n` annotations on the nodes to generate a formal specification. Based on the type of node in which the identifier occurs, the tool infers whether the labeled node is a requirement or an assumption. Thereafter, we can transform and graft back onto the safety case the proofs that result after running AUTOCERT on the generated specification.

6 Generation of Safety Case Metrics

6.1 Metrics Derivation

There has been some criticism of safety cases as lacking a measurement basis and, therefore, impeding systematic, repeatable evaluation [21]. We attempt to address this weakness of safety cases, using AdvoCATE, by defining and implementing a (preliminary) set of safety case metrics. Our goal is to create a transparent, quantitative foundation for assessment/review. It is worth noting that metrics alone (including those given here) do not necessarily constitute an assessment; rather, together with a model for interpretation, they can provide a convenient mechanism for decision-making by summarizing the state and key properties of a safety case during its evolution.

We distinguish between (i) *base* metrics, which express a direct measurement of, or value assignment to, a safety case property, e.g., the number of claims in a safety argument, and (ii) *derived* metrics, which are an analytical combination of base metrics, expressing a measure of, or a value assignment to, a safety case property that is not directly measurable, e.g., coverage.

Table 1. (Excerpt) GQM based derivation of safety case metrics and their specification

Goal G1. Coverage of Claims: *Analyze* the argument structure *for the purpose of* establishing the extent of coverage *with respect to* the claims made and the evidence presented *from the viewpoint of* the assessment team *in the context of* the safety case of the Swift UAS.	
Questions	**Metrics**
Q1.1. What is the total number of claims made?	BM1.1. Total #(Claims)
Q1.2. What is the total number of claims that end in evidence, i.e., developed claims?	DM1.2. Total #(Developed claims)
Q1.3. What fraction of the total number of claims are developed claims?	DM1.3. Coverage (Claims)
Specification: – BM1.1. Total #(Claims) = C, $C \geq 1$. – DM1.2. Total #(Developed claims) = C_D, $C_D \geq 0$. – DM1.3. Coverage (Claims) = Fraction of developed claims = $COV_C = \frac{C_D}{C}$.	

We consider an underlying process for safety case assessment, e.g., based on inspections [11], or reviews [12] to get insights into where metrics can be useful for decision making during assessment, and the interpretation models required. Thereafter, we use the Goal-Question-Metric (GQM) method [2] to define appropriate measurement goals, identify questions that characterize the goal, and specify the relevant metrics.

For instance, in a staged argument review [12], quantitative measures applied at the step of checking well-formedness can summarize the relevant properties, e.g., the number of goals with missing evidence/strategies. This can be useful when assessing large argument structures, where manual review of the entire structure, for well-formedness, can be time consuming. Similarly, during the argument criticism and defeat step, coverage of the top-level claim by evidence is a property for which metrics can be defined.

Table 1 gives an example of how GQM has been used to define metrics that, we believe, meet the goal of analyzing claims coverage. We state the measurement goal by instantiating the GQM template (the *italicized* text in Table 1), identify questions that characterize the goal, and define the metrics that answer the questions quantitatively. In Table 1, the base and derived metrics are distinguished by the prefixes BM and DM respectively. In this way, by defining additional measurement goals, we have specified[6] a preliminary set of safety case metrics (Table 2). For this paper, we have mainly focused on metrics that address the structural and syntactical properties of argument structures described using the GSN.

Note that although tool support can also be used to highlight violations, e.g., of well-formedness properties, this is mainly useful during argument development, where the intent would be to "find and fix". From the perspective of an assessor, however, the broad intent is to evaluate the argument for essential qualities [11]. When properly defined and interpreted, we hypothesize that metrics can be indicators of these qualities.

6.2 Metrics Implementation

The generation of the safety case metrics, as given in Table 2, is an automated operation, which uses some of the node metadata (Section 2.2) to count the nodes in the

[6] The full GQM-based derivation of the metrics, and their formal specifications, are out of the scope of this paper.

Table 2. Safety case metrics, with their valid values

Metric	Symbol	Type	Valid Values
Measures of Size			
Total #(Hazards considered in the safety case)	H	Base	≥ 0
Total #(Hazards identified in hazard analysis)	HI	Base	≥ 0
#(High-level safety requirements per hazard H_i)	$r(H_i)$	Base	≥ 0
Total #(High-level safety requirements)	R_{HL}	Base	≥ 0
Total #(Low-level safety requirements)	R_{LL}	Base	≥ 0
#(Developed claims per hazard H_i)	$c_D(H_i)$	Base	≥ 0
#(Claims per high-level safety requirement HLR_i)	$C(\text{HLR}_i)$	Base	≥ 0
#(Claims per low-level safety requirement LLR_i)	$C(\text{LLR}_i)$	Base	≥ 0
Total #(Claims)	C	Base	≥ 1
Total #(Developed claims)	C_D	Derived	≥ 0
Total #(Undeveloped claims)	C_{UD}	Derived	≥ 0
Total #(Uninstantiated claims)	C_{UI}	Derived	≥ 0
Total #(Strategies)	S	Base	≥ 0
Total #(Undeveloped strategies)	S_{UD}	Derived	≥ 0
Total #(Uninstantiated strategies)	S_{UI}	Derived	≥ 0
Total #(Contexts)	K	Base	≥ 0
Total #(Assumptions)	A	Base	≥ 0
Total #(Justifications)	J	Base	≥ 0
Total #(Evidence)	E	Base	≥ 0
Measures of Coverage			
Coverage (Claims)	COV_C	Derived	$[0, 1]$
Coverage (High-level safety requirements)	$COV_{R_{HL}}$	Derived	$[0, 1]$
Coverage (Low-level safety requirements)	$COV_{R_{LL}}$	Derived	$[0, 1]$
Coverage (Hazards considered)	COV_{CH}	Derived	$[0, 1]$
Coverage (Hazards Identified)	COV_{HI}	Derived	$[0, 1]$

EGSN-based safety case model, e.g., counting the nodes containing "high-level requirement" as an attribute gives the value assignment for the metric R_{HL}. Presently, only certain node types can be distinguished based on node attributes and metadata. Consequently, only a subset of the metrics identified in Table 2 have been implemented.

Fig. 5 shows the implemented metrics and the computed values when applied to the Swift UAS safety case fragment [8] (also shown as a bird's eye view in Fig. 3). As we define more expressive/detailed node metadata, we can implement the remainder of the metrics from Table 2, as well as additional metrics such as "confidence in a claim" [7].

7 Transformation Operations

We describe three automated operations defined in AdvoCATE, for generating artifacts that support safety case development and assessment:

To-do Lists. One simple form of assessment is determining those parts of the safety case that need further development. AdvoCATE uses a *Model2Text* transformation to create a simple to-do list, listing the undeveloped and uninstantiated nodes. Fig. 6 shows an excerpt of such a to-do list, for the Swift UAS safety case fragment.

```
---SIZE METRICS---
Goals: 220
        Developed: 157
        Undeveloped: 63
        Uninstantiated: 6
Strategies: 107
        Undeveloped: 13
        Uninstantiated: 0
Contexts: 133
Assumptions: 5
Justifications: 3
Evidence: 65
TOTAL NODES: 533

R_HL : Number of High-Level Requirements = 3
R_LL : Number of Low-Level Requirements = 2
R1_HL : Number of claims (High-Level Requirement 1) = 182
R2_HL : Number of claims (High-Level Requirement 2) = 1
R3_HL : Number of claims (High-Level Requirement 3) = 1
R1_LL : Number of claims (Low-Level Requirement 1) = 32
R2_LL : Number of claims (Low-Level Requirement 2) = 122

--- COVERAGE METRICS ---
COV_C : Developed claims to total claims = 0.71
COV_R_HL : Coverage of High-Level Requirements = 0.8
COV_R_LL : Coverage of Low-Level Requirements = 0.88
```

Fig. 5. AdvoCATE calculation of metrics for the Swift UAS safety case fragment

```
Undeveloped Goals To Do:
        ID:N43752193 :: Failure hazards during Cruise phase are mitigated
        ID:AC486 :: srcWpPos is a position in the NE frame (i.e. has_unit(srcWpPos, pos(ne)) holds.)
        ID:N63112384 :: Modem interface is correct ID:N11943209 :: FMS design is correct
        ...
Uninstantiated Goals To Do:
        ID:N87102962 :: Autopilot module satisfies {Higher-level Requirement X}
        Derived from parent ID: N92654598 :: Argument that Autopilot module satisfies higher level
                                             requirements
        ID:N59408212 :: {Subsystem X} failure hazard during descent is mitigated
        Derived from parent ID: N3143972 :: Argument over all Swift UAV subsystems (identified
                                             failure hazards)
```

Fig. 6. (Excerpt) To-do list generated by AdvoCATE, for the Swift UAS safety case fragment

Narrative Form. The generation of a safety case narrative form, i.e., a structured document providing the content of the safety case in a readable form, uses an intermediate tree model. The safety case can then be flattened into a sequence that is a pre-order traversal of the tree, giving a description of the content of the safety case without the diagrammatic form.

Tabular Form. We generate a comma separated value (CSV) format of the document (Fig. 7) using an intermediate model for the transformation. The CSV template relates a goal with an arbitrary number of contexts and strategies. The strategies are further related to any number of assumptions, contexts, justifications, and sub-goals. For each goal the operation generates these relationships. The operation then repeats the process for each sub-goal related to each strategy. The rationale for a specific CSV format of a safety case, and the resulting tabular form, is based on the experiences gained [5] from the ongoing creation of the Swift UAS safety case.

PARENT GOAL	CONTEXT	STRATEGY				SUBGOAL/SOLUTION
		Strategy Type	Context	Assumptions	Justifications	
N27216417: SWIFT UAS is safe	N80058283: Range (Location and Site) of operation	N18584532: Argument of safety over all UAS subsystems and interactions between subsystems	N91753638: SWIFT UAS Design Management Plan and Design Documentation			N2946770: SWIFT UAS Communication Infrastructure is safe
	N24389172: Specified configuration					N20743322: Airborne system (SWIFT UAV) is safe
	N44679952: Weather conditions					N83345544: SWIFT UAS subsystem interactions are safe
	N86072314: Specified Mission					N67094880: SWIFT Ground stations are safe
N2946770: SWIFT UAS Communication Infrastructure is safe						
N20743322: Airborne system (SWIFT UAV) is safe		N70618522: Argument of hazard mitigation over all identified SWIFT UAV hazards	N49558532: Definition of acceptable risk and risk categories			N44519454: Interaction hazards
			N2965510: Identified hazards and hazard categories during Swift UAV Hazard analysis			N69623828: SWIFT UAV failure hazards are mitigated
			N84863913: Definition of hazard from MIL-STD-882D			N40609843: Hazards arising from the operating environment of SWIFT UAV are mitigated

Fig. 7. (Excerpt) CSV format of the Swift UAS safety case generated using AdvoCATE, subsequently imported into a spreadsheet, resulting in a tabular view

8 Conclusion

In this paper, we have described AdvoCATE, an Eclipse-based toolset that uses model-based transformation and extended GSN to support the automated construction and assessment of safety cases.

We have just begun to develop the wealth of functionality for automated construction that can be implemented using transformations, e.g., a simple extension will be the generation of traceability matrices linking requirements, hazards and evidence. A more involved transformation will be argument refactoring. Our next step will be to include modular extensions to GSN, patterns, and to provide automated features for their use. To support safety case assessment, we have defined and implemented a preliminary set of metrics based on the syntactic/structural properties of argument structures documented using GSN. As future work, we intend to define integrated measures that combine the metrics based on both syntactic and semantic properties, building on our previous work on confidence quantification [7]. We will also define interpretation models based upon which metrics can be used, during assessment, for decision making.

For tool validation, we continue regression testing of the interface and transformations, and we also plan to verify the algorithms that AdvoCATE implements. We believe that the capabilities of AdvoCATE, highlighted in this paper, are promising steps towards cost-effective safety assurance, and transparency during assessment and certification. Eventually, our goal is to support "round-trip engineering" of safety cases, linking safety-relevant, operational, and development artifacts.

Acknowledgements. This work was funded by the VVFCS element under the SSAT project in the Aviation Safety Program of the NASA Aeronautics Mission Directorate. We also thank Ábel Hegedüs and Michael Wenz for their help with VIATRA2 and Graphiti, respectively, and Corey Ippolito for access to the Swift UAS data.

References

[1] Adelard LLP: Assurance and safety case environment (ASCE), http://www.adelard.com/asce/ (last accessed May 2011)

[2] Basili, V., Caldiera, G., Rombach, D.: Goal question metric approach. In: Encyclopedia of Software Engineering, pp. 528–532. John Wiley (1994)

[3] Basir, N., Denney, E., Fischer, B.: Deriving Safety Cases for Hierarchical Structure in Model-Based Development. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 68–81. Springer, Heidelberg (2010)

[4] Denney, E., Fischer, B.: Generating customized verifiers for automatically generated code. In: Proc. Conf. Generative Programming and Component Eng., pp. 77–87 (October 2008)

[5] Denney, E., Habli, I., Pai, G.: Perspectives on software safety case development for unmanned aircraft. In: Proc. 42nd Intl. Conf. Dependable Systems and Networks (June 2012)

[6] Denney, E., Pai, G.: A Lightweight Methodology for Safety Case Assembly. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 1–12. Springer, Heidelberg (2012)

[7] Denney, E., Pai, G., Habli, I.: Towards measurement of confidence in safety cases. In: Proc. 5th Intl. Symp. Empirical Soft. Eng. and Measurement, pp. 380–383 (September 2011)

[8] 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 (July 2012)

[9] Denney, E., Trac, S.: A software safety certification tool for automatically generated guidance, navigation and control code. In: IEEE Aerospace Conf. Electronic Proc. (2008)

[10] Goal Structuring Notation Working Group: GSN Community Standard Version 1 (November 2011), http://www.goalstructuringnotation.info/

[11] Graydon, P., Knight, J., Green, M.: Certification and safety cases. In: Proc. 28th Intl. System Safety Conf. (September 2010)

[12] Kelly, T.P.: Reviewing Assurance Arguments - A Step-by-Step Approach. In: Proc. Workshop on Assurance Cases for Security - The Metrics Challenge, Dependable Systems and Networks (July 2007)

[13] Kestrel Technology LLP and NASA Langley Research Center: CertWare tool, http://nasa.github.com/CertWare/ (last accessed May 2011)

[14] Matsuno, Y., Takamura, H., Ishikawa, Y.: Dependability case editor with pattern library. In: Proc. 12th IEEE Intl. Symp. High-Assurance Systems Eng., pp. 170–171 (2010)

[15] National Research Council Committee on Certifiably Dependable Software Systems: Software for Dependable Systems: Sufficient Evidence? National Academies Press (2007)

[16] Spriggs, J.: GSN - The Goal Structuring Notation. Springer (2012)

[17] Steele, P., Collins, K., Knight, J.: ACCESS: A toolset for safety case creation and management. In: Proc. 29th Intl. Systems Safety Conf. (August 2011)

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

[19] U.S. Department of Transportation, Federal Aviation Administration: System Safety Handbook. FAA (December 2000)

[20] Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68(3), 214–234 (2007)

[21] Wassyng, A., Maibaum, T., Lawford, M., Bherer, H.: Software Certification: Is There a Case against Safety Cases? In: Calinescu, R., Jackson, E. (eds.) Monterey Workshop 2010. LNCS, vol. 6662, pp. 206–227. Springer, Heidelberg (2011)