Ontology-integrated Model-based Assurance Cases

Anonymous Author(s)

ABSTRACT

Assurance cases (ACs) are increasingly being championed in emerging autonomy safety standards as a preferred means of providing confidence that an autonomous system is sufficiently safe. We have substantially extended an open-source AC toolkit with a variety of models to capture the diverse facets of assurance; namely models of: system hazards and requirements recording an *assurance basis*, risk scenarios and mitigations describing an *assurance architecture*, *structured arguments* capturing safety assurance rationale, and *evidence*. This paper describes how we: 1) embed these core assurance models in a user-extensible ontology to facilitate domain modeling, and 2) use an ontology-backed query language to analyze the resulting, semantically enhanced assurance model. These extensions provide system stakeholders with a capacity to specify queries that encode domain- and role-specific assurance concerns, and will eventually facilitate graphical views that communicate query results. So far as we are aware, these innovations set our framework apart from the state of the art and the prevailing practice in AC development. We illustrate the utility of our framework by using examples from an AC for an autonomous underwater vehicle system.

CCS CONCEPTS

• **Software and its engineering** → **Domain specific languages**; **Integration frameworks**; • **Computer systems organization** → *Robotic autonomy*.

KEYWORDS

Assurance cases, Autonomy, Model-based assurance, Ontologies, Queries

ACM Reference Format:

Anonymous Author(s). 2020. Ontology-integrated Model-based Assurance Cases. In *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, October 18–23, 2020, Montreal, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/xxxyyyz.xxxyyyz

1 INTRODUCTION

An *assurance case* (AC) is a risk management artifact used to justify to stakeholders, e.g., regulators, that a system or service will function as intended for a defined application and operating environment. ACs have been successfully used for dependability assurance of novel safety-critical applications where regulations and standards continue to be under development, e.g., unmanned aircraft systems [5]. Increasingly, emerging autonomy safety standards [14, 21] are recommending the use of ACs to engender trust in machine learning (ML) based autonomous systems.

A core component of an AC is a *structured argument* that captures (often diagrammatically) the evidence and rationale for confidently relying upon a system or service. Thus, the prevailing approaches to AC development include either informal arguments whose content is largely given in descriptive natural language, fully formal arguments, e.g., [13], or a combination of the two, e.g., [7].

Although readily comprehensible by human stakeholders, evaluating informal arguments requires careful inspection by competent domain experts. Conversely, formal arguments are amenable to automated analysis but are limited in their scope: not all assurance concerns of a system can be fully, or consistently formalized. Additionally, experience reports of creating real-world ACs that have successfully undergone regulatory scrutiny [3] suggest that a richer, multi-faceted notion of assurance may be more appropriate. Practically, moreover, providing assurance entails allaying stakeholder-specific concerns, and can involve mechanisms other than assurance argumentation [11].

For this work, we have adopted AdvoCATE [9], an open-source, model-based toolkit for structured arguments and their abstractions, i.e., *patterns* [8]. Over the past several years, we have extended AdvoCATE with a number of supplementary models beyond those that it provides for assurance rationale capture (arguments and patterns), to construct, analyze, and maintain the following additional elements: an *assurance basis*, an *assurance architecture*, and *evidence* (Section 2.2 elaborates these in more detail). In application to real-world systems, we have found these to be practically useful in the provision of assurance and, together with assurance rationale, they constitute the core components of useful ACs.

This paper presents our vision for how ontologies can further enhance this model-based approach to assurance, and ongoing work on its implementation in AdvoCATE, in particular: 1) formulating domain-specific extensions to the underlying models, 2) querying the resulting extended models[1], and 3) instantiating argument patterns with artifacts from the extended models to automatically generate instance arguments, with consistency between patterns and their instances being maintained using bidirectional transformations (BX). We illustrate these enhancements using excerpts of the components of an AC for an autonomous underwater vehicle, showing how ontologies can enrich assurance modeling.

Our goal in integrating ontologies into model-based assurance is to provide the benefits of formalism while retaining the key communicative purpose of ACs, without sacrificing their comprehensibility. By mapping assurance case AC components to a domain-specific ontology we facilitate AC *validation*, and by applying domain- and stakeholder-specific queries to core AC components that have been

MODELS '20, October 18–23, 2020, Montreal, Canada

ACM ISBN 978-1-4503-XXXX-X/XX/XX…$15.00
https://doi.org/10.1145/xxxyyyz.xxxyyyz

[1] This paper does not address *views*, although it is part of the broader scope of enhancements planned as future work (see Section 4).

semantically enriched using ontologies, we provide additional stakeholder insights.

General purpose query languages for assurance arguments and associated models have been investigated [18], as have languages more targeted at assurance arguments [6], though neither has exploited integrations with ontologies. Ontologies have been widely used in requirements development, e.g., [12], though less so for ACs [16]. As such, so far as we are aware, ontology-integrated model-based ACs represent a novel extension to the state of the art and prevailing practice of AC development.

The rest of our paper is organized as follows: Section 2 presents the core components of a model-based AC, while Section 3 presents the ontology-based enhancements. Section 4 concludes, discussing additional related work and future directions.

2 BACKGROUND

2.1 Preliminaries

As previously mentioned, we have extended AdvoCATE with additional components, towards an integrated assurance model. In this section, we describe these concepts and exemplify (some of) them with excerpts from an AC for a running example: an autonomous underwater vehicle (AUV) tasked with performing a long duration mission in which it is to provide surveillance of underwater relief and other objects (e.g., mines).

The AUV payload is an imaging sonar that, together with a forward scanning sonar, provides sensor information about the operating environment to an onboard reinforcement learning-based controller. This controller is itself embedded within an autonomous planner component responsible for path planning. Two amongst the main applicable assurance concerns are *safety* (avoiding collisions, e.g., with static and dynamic obstacles) and *mission continuity* (continuing to operate despite degradations).

2.2 Core Assurance Case Components

2.2.1 Assurance Basis. An assurance basis records the risks posed by/to a system, and the corresponding risk management and mitigation objectives. We have extended AdvoCATE with tabular models to capture both these aspects, and Figure 1 shows an excerpt of the assurance basis for the AUV AC.

The *hazard log*, which captures the risks posed (Figure 1, top), comprises a collection of *tabular hazard models* that record hazard conditions, precursors, effects, and associated level of risk posed, together with high-level mitigation mechanisms. There exists a hazard table for each operational context that is characterized by the combination of AUV hazardous activities, system states, and environmental conditions. A *tabular requirements model* (Figure 1, bottom left) reflects the related *assurance requirements* associated with the mitigations identified in the hazard log. The hazard log also references these requirements (shown by the highlighted cells) although the requirements model effectively captures different information relevant for the wider AC, e.g., requirement allocation to system functions and components, verification methods, etc.

An example hazard, as shown, concerns a deviation from the expected output (vehicle heading) of the autonomous planner AUV component, and the resulting assurance requirements call for monitoring and failover functionality.

Also shown in Figure 1 (bottom right) is the AUV physical decomposition used to allocate hazards and requirements.

2.2.2 Assurance Architecture. An assurance architecture details (typically operational) risk scenarios showing the system capabilities that participate in risk mitigation. Built compositionally, it is an abstraction of how the system architecture contributes to risk reduction and, in turn, to dependability assurance.

Barrier models represented using *bow tie diagrams* (BTDs) have been shown to be useful for this purpose [10], and we have adopted, added, and extended them in AdvoCATE. An example operational scenario (not shown here due to space constraints) consistent with the hazard identified in Figure 1 would contain, for instance, the chain of events beginning from the deviation in the autonomous planner output and terminating in a collision, also would also show when the identified mitigation measures—such as runtime monitoring and failover mechanisms—are employed.

2.2.3 Assurance Rationale. Assurance rationale is the justification for trusting that a system is fit for purpose. As indicated earlier, AdvoCATE natively supports assurance rationale capture using models of both structured arguments and their patterns.

An argument contains explicit assurance claims substantiated by (typically diverse) evidence, where reasoning steps elaborate why the evidence supplied entail the claims made. Arguments can be specified graphically, in a textual form, or using a combination of the two. AdvoCATE supports the graphical *Goal Structuring Notation* (GSN) [20] for this purpose.

Figure 2 shows a GSN argument fragment substantiating a claim of mitigating a lower-level hazard related to a deviation in the heading output of the autonomous planner component (shown by the rectangular *goal* node G550). This hazard is itself a *cause* of a higher-level hazard (see Figure 1). The argument shows three complimentary legs of reasoning; i.e., that: 1) the identified hazard mitigation constraints and requirements have been met; 2) operational mitigations can effect recovery; and 3) the identified causal factors of the hazard have been managed. These correspond, respectively, to the *strategy* nodes (shown as parallelograms in Figure 2) S13, S14, and S15, and their children nodes.

The diamond annotation on nodes indicates incompleteness. The oval and racetrack shaped nodes are *assumption* and *contextual* elements, respectively, that add clarifying detail to the argument node to which they are attached. For instance, the claim in goal node G552, rests on the assumption stated in the assumption node A2. Other relevant argument nodes (not shown or discussed further here, see [20] for details) provide *justifications* and *solutions*, the latter of which refer to evidence items. Links between nodes with solid arrowheads represent inferential relations, interpreted as "is supported by", while those with hollow arrowheads, interpreted as "in context of", are contextual relations.

Argument patterns are reusable abstractions of arguments, using which the latter can be gradually developed through automated instantiation and composition. AdvoCATE implements GSN patterns, which include notational elements for abstraction via parameterization, multiplicity, choice, and recursion. We defer further discussion of patterns to Section 3.4, where we also present the ontology extensions to the same.

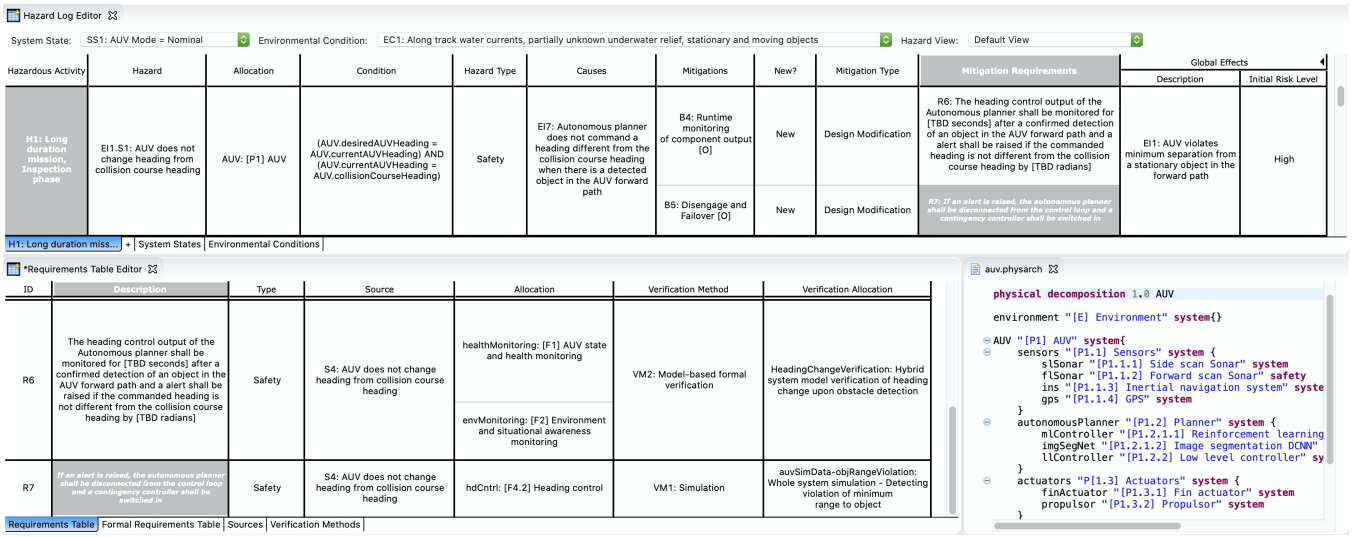

Figure 1: AdvoCATE screenshot showing an excerpt of the assurance basis component of an AUV assurance case.

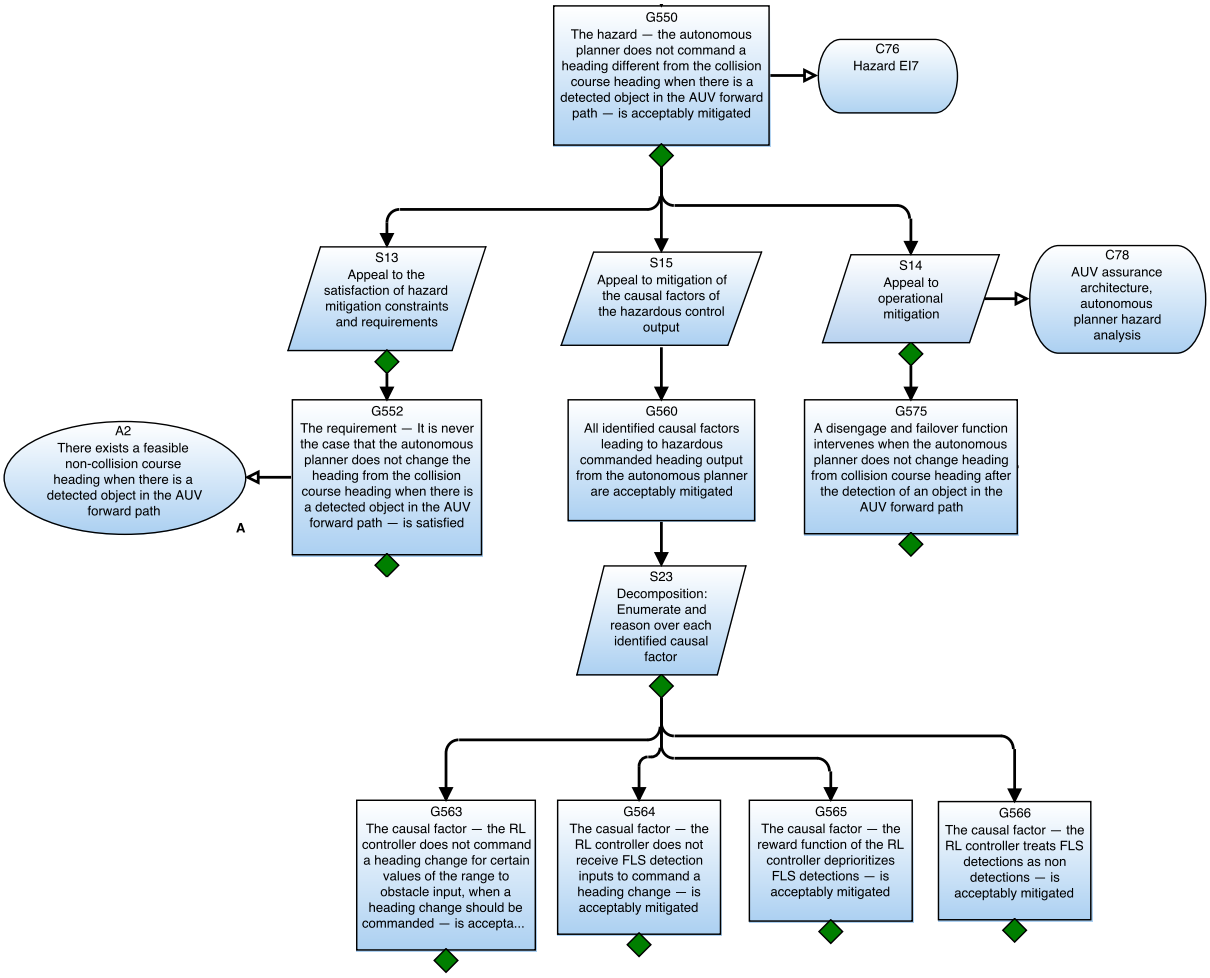

Figure 2: Fragment of an assurance argument, in GSN, from the AUV assurance case.

2.2.4 Evidence. Evidence comprises, among other things, development and operational data and artifacts that—together with the above core components—concretely corroborate assurance claims, thus underpinning an AC. The evidence model in our framework captures the relationships not only between the various evidence items (to be) used in an AC, but also to core AC components. Additionally, this model records *evidence provenance*, e.g., whether it is (or will be) generated by a verification tool, and *evidence assertions*, i.e., the concrete conclusions that can be drawn from a given item of evidence.

3 APPROACH

3.1 Overview

Each of the components presented in the preceding section has a model-based representation, which the tool user interface displays using a variety of formats—each component has a domain specific language (DSL), and some also have tabular or graphical representations. DSLs are built with Xtext[2], tables with NatTable[3], and graphical diagrams using Sirius[4]. We refer to the collection of inter-related models as the *integrated assurance case model*, or simply AC model. We employ model transformations to generate artifacts from the AC model, in particular, assurance arguments.

Though formal approaches have been taken to the construction of ACs, either incorporating formal reasoning [7] or integrating with external models with formal semantics [2, 13], this introduces a tension with one of the fundamental purposes of ACs: to *communicate and convince*.[5] We believe ontology-backed ACs can provide the advantages of both informal and formal approaches.

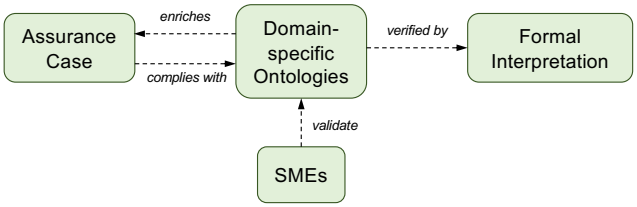

Figure 3: Vision for ontology-backed assurance cases.

Figure 3 illustrates our vision, where the AC model is embedded in a user-extensible ontology that contains information from the assurance case, which can then be extended with domain-specific concepts. The ontology can be validated by subject matter experts (SMEs) and serves as a semi-formal specification of the domain that can, optionally, be mapped to a formal semantics for verification. Elements can, in turn, be used to construct parts of the assurance case through the use of an ontology-backed *structured language*.

The ontology provides a vocabulary for, for example, claims of the assurance arguments. Well-formedness of claims and soundness of some forms of reasoning can be determined by the ontology. It also provides a vocabulary for domain-specific queries that are also used in patterns to generate arguments. We will focus on the queries and patterns here.

3.2 Ontology Extensions

We map elements of the AC model to concepts, relations, and their instances in a *derived ontology* that is user-extensible. This ontology, itself, then forms part of an extended model. Since the ontologies are, in effect, also part of our model, we will sometimes use *core model* to refer to the non-ontological part.

For example, one of the extensions we have made to the core model in AdvoCATE includes a simple notion of physical architecture consisting of a hierarchy of components (see Figure 1, bottom right). In the ontology, we represent this by a concept `Component`, whose instances are the actual components of a given system. A relation `subComponent` represents the containment relation of the architecture. The user can then define new concepts and relations to enrich the model, such as concepts for component input and output, and relations for connections.

Figure 4 illustrates some features of our ontology definition language, which has an object-oriented flavor and is reasonably verbose. We distinguish *conceptual* and *instance* ontologies, where the former defines concepts and their relations, and the latter instantiates them. Concept declarations optionally give super-concepts, attributes, and relations to other concepts. Attributes have types (primitive, enumerated, list, record, and any combination).

In the AUV conceptual ontology example (Figure 4, left) the concept `Actuator` is a sub-concept of `AUVComponent`, with the boolean attribute `isActuated` and the relations `actuates` and `sending`, to the concepts `PhysicalComponent` and `ActuationSignal`, respectively. We can also define concepts from other concepts using union, intersection, negation, and quantification along relations. We can lift attributes from the target concept of a relation to the relation, itself. Here, a `DetachedFin` is defined to be a `Fin` such that every `DegradedFin` it `degradesTo` has no (zero) `liftDragEfficiency` despite being `actuatedBy` every `FinActuator` that `isActuated`.

We have defined our own languages rather than use existing languages such as the Web Ontology Language (OWL)[6] and SPARQL[7] because it enables a tighter integration with our core model and a similar style of DSL.

3.3 Queries

Figure 5 shows example queries over the integrated AC model.

Figure 5a shows a query for goal nodes of arguments in the AC, that contain claims referring to the reinforcement learning controller, and that are eventually *supported* (i.e., followed) by at least one solution node that is related to verification evidence. Here, `eventually` is used to form the reflexive transitive closure of a relation.

The second query, in Figure 5b, looks for requirements allocated to the autonomous planner (`autonomousPlanner`), and that represent the requirements to implement the mitigations of hazards that are, in turn, allocated to the AUV fins (`Fin`) and whose hazard condition involves either a stuck open starboard fin (`stuckOpenStbdFin`) or detached port fin (`detachedPrtFin`). These items correspond to

[2] https://www.eclipse.org/Xtext/
[3] https://www.eclipse.org/nattable/
[4] https://www.eclipse.org/sirius/
[5] That the sources of risk have been identified, are well-understood, and that they have been appropriately managed.
[6] https://www.w3.org/OWL/
[7] https://www.w3.org/TR/sparql11-query/

auvConceptual.ontology

```
define 1.0 auvConceptual           // Declares conceptual AUV ontology
concept AUVDataVar
concept componentStateVar

concept AUVComponent is an Item {   // AUV Component
     hasInput an AUVDataVar
     hasOutput an AUVDataVar
     hasState a componentStateVar }

enum actuationSignalType{ finAngle, rpm }
enum locationType {top, bottom, starboard, port}
enum finActuationType { fullActuation,
     slightUnderActuation, largeUnderActuation}

concept Actuator is an AUVComponent {   // Actuator
     isActuated bool
     actuates a PhysicalComponent, sending an ActuationSignal  }

concept ActuationSignal { actuationSignal actuationSignalType }

concept FinActuator equals Actuator such that actuates all Fin
     and actuationSignal = finAngle { actuationType finActuationType }

concept Fin is a PhysicalComponent {    // AUV Fins
    finLocation locationType
    trueFinAngle real
    finOffset real
    finEfficiency real
    liftDragForce real
    actuatedBy a FinActuator, receiving an ActuationSignal
    degradesTo a DegradedFin }

concept DegradedMode                    // Degraded modes

concept DegradedFin is a DegradedMode { // Degraded fin
     finEfficiency real
     finAngle real
     finOffset real
     liftDragForce real }

concept StuckFin equals Fin such that all degradesTo.finEfficiency < 0.75
concept DetachedFin equals Fin such that all degradesTo.liftDragForce = 0
      and all actuatedBy.isActuated = true
```

auv.ontology

```
instantiate 1.0 auvConceptual as auv   // Instantiate AUV conceptual ontology
import mlcontroller                    // Import ML controller instance ontology

AUVDataVar
     currentAUVHeading, collisionCourseHeading, // Current and Collision headings
     currentAUVSpeed, currentAUVSpeedDVL,
     currentAUVDepth, currentRangeToInf,
     currentInfBearing, timeInfLastSeen,
     maxPredictedSASRange, closestPointOfApproach

Controller autonomousPlanner {
     // Vehicle State inputs to mlController
     hasInput (
          currentAUVHeading, currentAUVSpeed,
          currentAUVSpeedDVL, currentAUVDepth )

     // Surveilled infrastructure information
     hasInput (
          currentRangeToInf, currentInfBearing,
          timeInfLastSeen, maxPredictedSASRange )

     hasInput closestPointOfApproach     // Obstacle Information

     // Autonomous planner outputs are the outputs of mlController
     hasOutput (
          mlcontroller.desiredAUVHeading, mlcontroller.desiredAUVDepth,
          mlcontroller.desiredAUVSpeed )
 }

 Actuator finActuator, propulsor         // Actuators

 Fin TopFin { finLocation = top }        // AUV fins
 Fin BottomFin { finLocation = bottom }
 Fin StbdFin { finLocation = starboard }
 Fin PortFin { finLocation = port }

StuckFin stuckOpenStbdFin {              // Stuck open starboard fin
     finLocation = starboard
     finOffset = 90   }

DetachedFin detachedPrtFin {
     finLocation = port
     finEfficiency = 0 }
```

Figure 4: Fragments of (left) AUV conceptual ontology describing fin degradation modes and (right) AUV instance ontology instantiating the autonomous planner component and AUV fins.

```
ArgumentNode such that type = goal
  and description = "*rlController*" and
    eventually SupportedBy some (ArgumentNode
      such that type = solution and
        some relatedEvidence.type = verification)
```

(a) Querying the structured argument component.

```
Requirement such that allocation contains autonomousPlanner
  and isRequirement for some Mitigation for some
    (Hazard such that allocation contains Fin and
      (condition contains stuckOpenStbdFin or
          condition contains detachedPrtFin))
```

(b) Querying the assurance basis component.

Figure 5: Examples of querying the AUV AC invoking terms declared in ontologies.

the concepts and instances defined in the corresponding ontologies (Figure 4).

Note, here, that we use "allocation" in two distinct ways: in the first part of the query it refers to a *requirements allocation*, which is a responsibility assignment of the requirement to, say, a component in the physical decomposition model, also reflected as an instance in the instance ontology. In the second part of the query, it refers to a *hazard allocation*, that is, the location of the hazard.

3.4 Pattern-based Argument Generation

As previously discussed (Section 2.2.3), argument patterns are an abstraction that can be used to transform data into arguments. Figure 6 shows an argument pattern in GSN, abstracting the (reasoning steps of the) argument structure of Figure 2. In this pattern, the abstractions are *parameters* enclosed in braces "{ }" in the argument nodes, *multiplicity annotations* on the links (shown as '•'), and the *loop* link connecting children nodes to their ancestors. We do not discuss the remaining pattern abstractions [20] here.

We can instantiate such patterns using source data to replace the specified parameters. This data is either external to the AC (e.g., extracted from a collection of test cases), or internal to the AC and extracted from other components of the AC (such as hazard tables; see Section 2.2). Argument patterns are, in effect, a concise domain-specific notation for defining transformations of source data into assurance arguments. To instantiate a pattern, we assign the root parameter to some element of the model (such as a specific hazard table), and queries then navigate from that point through the model to instantiate the remaining parameters in the pattern.

For example, in the pattern root goal node G1 (Figure 6), we query for those hazards allocated to the `autonomousPlanner` component declared in the physical decomposition (Figure 1) and whose hazard condition (that is, the desired heading does not change from the current heading, which is also the collision course heading) contains the statement shown. Here, `allocated` and `condition` are elements of the core model reflecting, respectively, the hazard allocation and the specific state that precipitates the hazard (hazard condition). Also, `desiredAUVHeading`, `currentAUVHeading`, and `collisionCourseHeading` are instances in the AUV instance ontology (Figure 4, right) of the `AUVDataVar` concept declared in the AUV conceptual ontology (Figure 4, left).

We annotate links with constraints between parameters that enable, for example, the requirement claim in goal node G4 to be

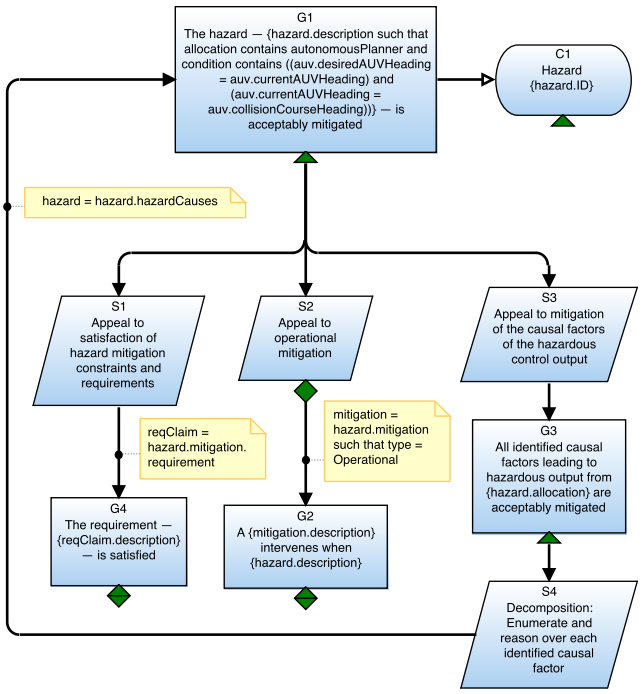

Figure 6: GSN argument pattern abstracting the argument structure of Figure 2, enriched with ontological data.

located in the model, given the hazard in goal node G1. We then instantiate the text in G4 with the description of that requirement in the model. The link from the strategy node S4 to the goal node G1 is a loop, reflecting the recursive nature of this pattern: causes of hazards are, themselves, hazards, and thus amenable to the same reasoning.

The tool generates traceability links allowing navigation between the argument, pattern, and source data, and we use bidirectional transformations to maintain consistency whilst supporting *round-trip persistence*, that is, the ability to edit arguments generated from patterns and for those edits to persist when the argument is re-generated. Users can optionally propagate selected edits to the corresponding artifacts. For instance, we can add an assumption node to the instance argument generated (such as the assumption node A2, as shown in Figure 2), and optionally propagate that change to the pattern of Figure 6, abstracting node content as a parameter whose value is resolved by a query locating the assumption connected to `reqClaim`.

4 CONCLUDING REMARKS

We have described ongoing work on the integration of ontologies into the AdvoCATE model-based AC toolset. Thus far we have primarily used the ontologies for domain and system modeling, and querying the extended model. By tightly coupling the DSLs for ontology definition with the other AC languages in the tool, we offer powerful new mechanisms for AC creation and analysis.

Related Work. The distinction between models and ontologies has been discussed in the literature. Our view is close to that of [1] who consider models to be *prescriptive* (and representing specifications of a system) while ontologies are *descriptive* (and constituting a description of the external world as it is). We think this is a useful distinction, though we also allow ontologies to represent the system under assurance. In practice, this is reflected in extensions to AdvoCATE, where the built-in assurance model comprises those well-defined domain-independent components that we consider key parts of an AC, while the ontologies allow users to flexibly model domain-specific extensions.

Previous work on integrating models and ontologies [1, 19, 22] has discussed how to use ontologies to provide semantics to enrich relations between models, such as model transformations, but has tended to present high-level abstract frameworks, rather than concrete tools and languages.

There has been much work on mediating database access through ontologies. Most relevant to our work is the notion of *ontology-based database access* [4] where a database can be queried using an extended vocabulary that enriches a database with domain-specific semantics. This can use ontology-to-database mappings [15] where declarative rules describe how elements of relational database schemas correspond to the ontology.

Most of this work has been domain-independent and not directly targeting assurance, although [16] explores the use of ontologies for hazard analysis, while [17] explores using ontologies to elicit safety requirements. However, they do not distinguish model and ontology, using ontology to refer to a fixed prescribed structure for representing hazards and their constituent elements (corresponding to the assurance basis component of our model), as well as domain-specific parts that for us would be captured by an ontology.

Future Directions. As mentioned, one of the main motivations for this work is to support validation (and ultimately verification) of the AC. We are doing this by embedding model elements in a domain ontology. A second aspect of this, which is ongoing, is the development of a structured language for expressing claims and other statements in the structured argument component of an AC. We aim to support a range of formality, from free-form natural language to structured statements composed of elements from the ontology, for which well-formedness and correctness of inference rules (with respect to the ontology) would be automatically enforced. The ontology could also be used to automatically generate sub-claims.

Although we have chosen to implement our own ontology definition languages, we do not discount the merits of integrating with other ontology tools, and plan to define import/export mechanisms via the OWL standard. We are also working on a view language that exploits queries to generate user-specified visualizations, both graphical and tabular.

We have described how we use bidirectional transformations to keep several artifacts in an AC consistent, in particular patterns and arguments. We plan to develop similar transformations between the ontology and the other assurance artifacts (e.g., adding a subgoal to an argument could add an instance to an ontology concept).

REFERENCES

[1] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. 2006. *Ontologies, Meta-models, and the Model-Driven Paradigm.* Springer Berlin Heidelberg, Berlin, Heidelberg, 249–273. https://doi.org/10.1007/3-540-34518-3_9

[2] Nurlida Basir, Ewen Denney, and Bernd Fischer. 2010. Deriving Safety Cases for Hierarchical Structure in Model-Based Development. In *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security* (Vienna, Austria) *(SAFECOMP' 10)*, Erwin Schoitsch (Ed.). Springer-Verlag, Berlin, Heidelberg, 68–81.

[3] Randall Berthold, Ewen Denney, Matthew Fladeland, Ganesh Pai, Bruce Storms, and Mark Sumich. 2014. Assuring Ground-based Detect and Avoid for UAS Operations. In *Proceedings of the 33rd IEEE/AIAA Digital Avionics Systems Conference (DASC)*. 6A1–1–6A1–16.

[4] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. 2007. Ontology-based Database Access. In *Proceedings of the 15th Italian Symposium on Advanced Database Systems (SEBD 2007)*, Michelangelo Ceci, Donato Malerba, and Letizia Tanca (Eds.). Torre Canne, Fasano, BR, Italy, 324–331.

[5] Reece Clothier, Ewen Denney, and Ganesh Pai. 2017. Making a Risk Informed Safety Case for Small Unmanned Aircraft System Operations. In *Proceedings of the 17th AIAA Aviation Technology, Integration, and Operations Conference (ATIO 2017), AIAA Aviation Forum*. https://doi.org/10.2514/6.2017-3275

[6] Ewen Denney, Dwight Naylor, and Ganesh Pai. 2014. Querying Safety Cases. In *Computer Safety, Reliability and Security. SAFECOMP 2014.*, Andrea Bondavalli and Felicita Di Giandomenico (Eds.). Lecture Notes in Computer Science, Vol. 8666. Springer, 294–309. https://doi.org/10.1007/978-3-319-10506-2_20

[7] Ewen Denney and Ganesh Pai. 2014. Automating the Assembly of Aviation Safety Cases. *IEEE Transactions on Reliability* 63, 4 (2014), 830–849.

[8] Ewen Denney and Ganesh Pai. 2015. *Safety Case Patterns: Theory and Applications*. Technical Report NASA/TM-2015-218492. NASA Ames Research Center.

[9] Ewen Denney and Ganesh Pai. 2018. Tool Support for Assurance Case Development. *Journal of Automated Software Engineering* 25, 3 (Sep. 2018), 435–499. https://doi.org/10.1007/s10515-017-0230-5

[10] Ewen Denney, Ganesh Pai, and Iain Whiteside. 2017. Model-driven Development of Safety Architectures. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 156–166. https://doi.org/10.1109/MODELS.2017.27

[11] Ewen Denney, Ganesh Pai, and Iain Whiteside. 2019. The Role of Safety Architectures in Aviation Safety Cases. *Reliability Engineering & System Safety* 191 (2019). https://doi.org/10.1016/j.ress.2019.106502

[12] Stefan Farfeleder, Thomas Moser, Andreas Krall, Tor Stålhane, Inah Omoronyia, and Herbert Zojer. 2011. Ontology-Driven Guidance for Requirements Elicitation. In *The Semanic Web: Research and Applications*, Grigoris Antoniou, Marko Grobelnik, Elena Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Pan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 212–226.

[13] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. 2014. Resolute: An Assurance Case Language for Architecture Models. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 19–28.

[14] International Organization for Standardization (ISO). 2019. Road vehicles — Safety of the intended functionality. Standard ISO/PAS 21448:2019.

[15] Kamran Munir and M. Sheraz Anjum. 2018. The use of ontologies for effective knowledge modelling and information retrieval. *Applied Computing and Informatics* 14, 2 (2018), 116 – 126. https://doi.org/10.1016/j.aci.2017.07.003

[16] Abigail Parisaca Vargas and Robin Bloomfield. 2015. Using Ontologies to Support Model-Based Exploration of the Dependencies between Causes and Consequences of Hazards. In *Proceedings of the International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2015)*. Lisbon, Portugal, 316–327. https://doi.org/10.5220/0005618003160327

[17] Luciana Provenzano, Kaj Hänninen, Jiale Zhou, and Kristina Lundqvist. 2017. An Ontological Approach to Elicit Safety Requirements. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. Nanjing, China, 713–718.

[18] Alessio Di Sandro, Sahar Kokaly, Rick Salay, and Marsha Chechik. 2019. Querying Automotive System Models and Safety Artifacts with MMINT and Viatra. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. Munich, Germany, 2–11.

[19] Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras. 2010. *Model Driven Engineering with Ontology Technologies*. Springer Berlin Heidelberg, Berlin, Heidelberg, 62–98. https://doi.org/10.1007/978-3-642-15543-7_3

[20] The Assurance Case Working Group (ACWG). 2018. Goal Structuring Notation Community Standard Version 2. https://scsc.uk/r141B:1

[21] Underwriter Laboratories Inc. 2020. ANSI/UL 4600 Standard for Safety for the Evaluation of Autonomous Products.

[22] Srdjan Živković, Marion Murzek, and Harald Kühn. 2008. Bringing Ontology Awareness into Model Driven Engineering Platforms. In *Proceedings of the 1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering TWOMDE 2008, Toulouse, France, September 28, 2008 (CEUR Workshop Proceedings, Vol. 395)*, Fernando Silva Parreiras, Jeff Z. Pan, Uwe Aßmann, and Jakob Henriksson (Eds.). CEUR-WS.org, 47–54.