

# Semantic Analysis of Assurance Cases using s(CASP)

Anitha Murugesan<sup>1</sup>, Isaac Hong Wong<sup>1</sup>, Robert Stroud<sup>2</sup>, Joaquín Arias<sup>3</sup>,  
Elmer Salazar<sup>4</sup>, Gopal Gupta<sup>4</sup>, Robin Bloomfield<sup>2</sup>, Srivatsan Varadarajan<sup>1</sup> and  
John Rushby<sup>5</sup>

<sup>1</sup>Honeywell Aerospace, USA

<sup>2</sup>Adelard (NCC Group), UK

<sup>3</sup>CETINIA - Universidad Rey Juan Carlos, SPAIN

<sup>4</sup>University of Texas, Dallas, USA

<sup>5</sup>SRI International, USA

## Abstract

The use of assurance cases is gaining popularity, particularly in the safety-critical system industry, as an organized approach to submitting documentation for the safety and security certification of systems. However, these arguments can become overwhelming and complicated, even for moderately complex systems. Therefore, there is a compelling requirement to develop new automation strategies that can aid in creating and assessing assurance cases. Existing assurance-case tools primarily automate syntactic analysis, focusing on structural completeness, while providing limited or no support for semantically evaluating the logical aspects of the assurance case.

In prior work, we introduced a framework called Assurance 2.0, which aims to enhance the rigor of assurance cases by emphasizing the reasoning process, evidence utilized, and explicit identification of counter-claims (defeaters) and counter-evidence. In this paper, we present a new approach to enhancing Assurance 2.0 by incorporating semantic rule-based analysis capabilities. Firstly, we systematically convert the assurance case into Prolog predicates and constraints. Then, leveraging the analysis capabilities of the s(CASP), a goal-directed top-down solver for Constraints Answer Set Programs, we evaluate the semantic properties of assurance cases, including logical consistency, completeness, and indefeasibility. The application of these analyses provides both authors and evaluators with higher confidence when assessing the assurance case.

## Keywords

Assurance Cases, Semantic Analysis, s(CASP)

## 1. Introduction

In recent years, there has been a growing interest in the safety-critical field to develop *assurance cases* as a means of establishing credible confidence in claims made about a system, particularly regarding its safety or security. These assurance cases typically consist of a claim that defines the desired property, evidence supporting the design and construction of the system, and a well-structured and persuasive argument demonstrating that the evidence is sufficient to


---

3rd Workshop on Goal-directed Execution of Answer Set Programs (GDE'23), July 10, 2023

✉ anitha.murugesan@honeywell.com (A. Murugesan); isaachong.wong@honeywell.com (I. H. Wong);  
robert.stroud@nccgroup.com (R. Stroud); joaquin.arias@urjc.es (J. Arias); elmer.salazar@utdallas.edu (E. Salazar);  
gupta@utdallas.edu (G. Gupta); robin.bloomfield@nccgroup.com (R. Bloomfield);  
srivatsan.varadarajan@honeywell.com (S. Varadarajan); Rushby@csl.sri.com (J. Rushby)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

support the claim. Unlike the traditional approach of presenting a laborious checklist of activities performed or not performed for system assurance, assurance cases focus on creating a systematic argumentative record. As a result, they are widely recommended as a practical alternative for organizations seeking safety/security certification from agencies such as the FAA [1].

At its core, a correct assurance case is one in which the top-level claim logically follows from the composition of its sub-claims, arguments, and supporting evidence. However, even for systems of moderate complexity, assurance cases can become exceedingly large, consisting of extensive sets of arguments and a substantial body of evidence, all expressed in natural language. Consequently, both the authors and evaluators of assurance cases must diligently verify the correctness of the arguments and evidence and ensure they logically support the stated claims about the system. Unfortunately, this process is time-consuming, intellectually demanding, and prone to errors. The risk lies in the possibility of approving flawed assurance arguments and consequently certifying systems that are unsafe or insecure.

To address this challenge, researchers and practitioners worldwide have developed numerous tools and techniques aimed at automatically analyzing specific syntactic elements of assurance cases. These include ensuring adherence to notations, assessing structural completeness, and conducting grammar and spell checks within the natural language descriptions [2]. However, there is currently not adequate support for systematically reasoning about the semantics or underlying meaning of the claims, arguments, and evidence presented in assurance cases, as well as verifying the accuracy of the logical relationships established throughout the case. Nevertheless, implementing rigorous automation for semantic analysis would reduce human effort and enhance confidence in the assurance case.

This paper presents our approach to conducting a formal analysis of specific semantic aspects within assurance cases. We have previously introduced Assurance 2.0 [3], a novel framework designed to enhance the rigor of assurance cases by emphasizing the reasoning process, evidence utilized, and explicit identification of defeaters (or counter-claims) and counter-evidence. The accompanying tool, Assurance and Safety Case Environment (ASCE) [4], facilitates the systematic creation of Assurance 2.0 cases and performs structural analysis to ensure their proper structure. The approach detailed in this paper extends this previous work by leveraging the s(CASP), a goal-directed top-down solver for Constraints Answer Set Programs [5, 6], to semantically reason about various properties of the assurance case such as *consistency* (absence of logical contradictions), *indefeasibility* (absence of violations), completeness, etc.

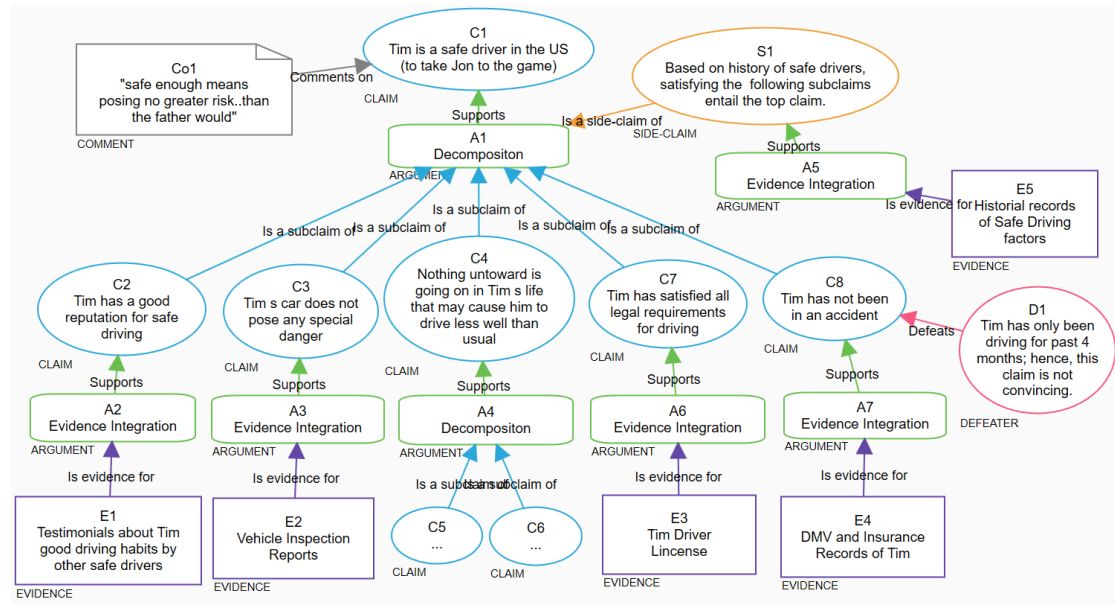
We begin by formally establishing the objects pertaining to the system under consideration, the properties that are asserted, and the operational environment for each claim, argument, and evidence. Although this step currently requires manual effort, we are actively investigating methods to automatically extract this information from natural language descriptions. Subsequently, we automatically convert the entire assurance case into predicates and constraints expressed in Prolog as a logic programming language. We then interface with the s(CASP) reasoning engine to systematically examine the consistency of the predicates, identifying any violations of the constraints, and assessing the completeness of the assurance arguments.

In order to illustrate the concepts of our approach and tool-suite, we will use a generic example [7] that concerns an assurance case built to determine whether the father of Jon (a teenager not yet of driving age) will allow him to ride a car with Tim (a college student known to Jon's family) to a football game. The top-level claim "Tim is a safe enough driver (to take

Jon to the game)" is supported by sub-claims and corroborating evidence that establish Tim's capability to drive safely. While there are more details associated with each claim and argument, we will focus on the high-level particulars solely for the purpose of illustration.

## 2. Assurance 2.0 Methodology: A brief primer

Assurance 2.0 is a modern framework to support reasoning and communication about the behavior and trustworthiness of engineered systems and their certification. This framework adopts a Claims-Arguments-Evidence approach, with an increased focus on the evidence presented, the logical reasoning utilized, and the exploration and assessment of counter-claims [8, 9].



**Figure 1:** Safe Driver Case in Assurance 2.0

Figure 1 shows an Assurance 2.0 case for the safe driver example. It has a top-level *claim* (blue ellipse) that is hierarchically refined into fine-grained *sub-claims* (blue ellipses) via *arguments* (green round-edge rectangles) that are eventually discharged by *evidence* (purple rectangles) at the leaf level. The rationale for such refinements is captured in *side-claims* (yellow ellipses). Further, doubts, concerns, or counter-claims on any part of the case are captured as *defeaters* (red ellipses) [8]. Defeaters provide the ability for authors and reviewers to challenge the case, which can later be addressed by adjusting the case or analyzing and disputing the challenge.

Recently, Assurance 2.0 is augmented with the notion of *theories* [10]. Theories are reusable assurance case fragments that resemble claims in structure but are designed to be applicable to a wide range of objects rather than specific system elements. They are comparable to function calls in typed programming languages. When applying a theory in an actual assurance case, the author instantiates it with concrete objects of the system in consideration. For example, instead of defining the assurance case for Tim (Figure 1), a "Theory of Safe Driver", Figure 2, can be defined for a generic human "X". When applying this theory, the author must systematically

instantiate  $X = \text{Tim}$  (or any other driver), and supply all the evidence obligations for Tim, to establish that Tim is a safe driver. A primary benefit of using theories is that many of these theories will have been used and analyzed previously, and therefore provide “pre-certified” sub-cases whose defeaters have been considered and eliminated.

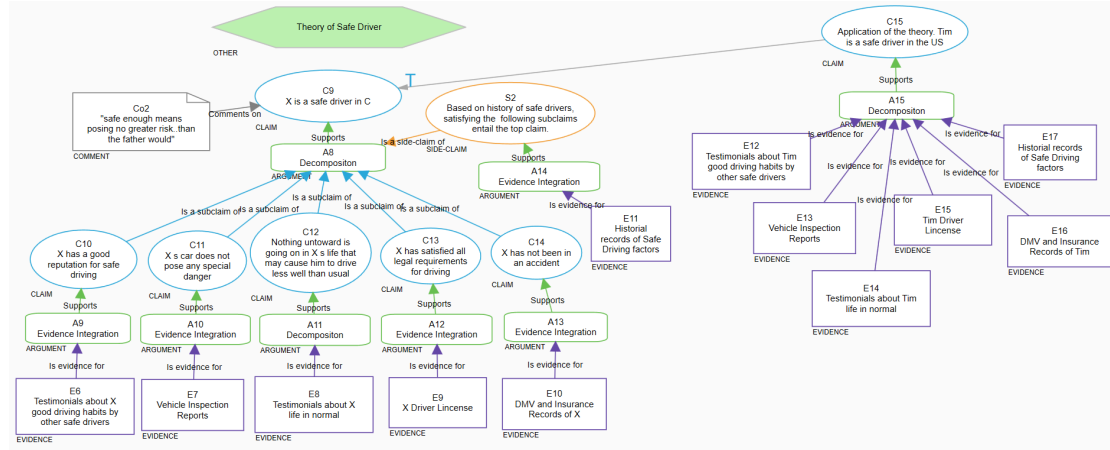


Figure 2: Theory of Safe Driver

### 3. Assurance Case to Logic with Semantic Formalism

To a large extent, assurance cases rely primarily on free-form natural language to document arguments and evidence, despite having a well-structured graphical outlook. As the complexity of the system being considered increases, assurance cases become excessively large and complex. In current practice, authors and evaluators manually review the entire case to ensure semantic correctness (the top-level claim logically follows from its sub-claims, arguments, and evidence), and that there are no logical inconsistencies or fallacies. Unfortunately, this process is time-consuming and intellectually demanding, due to the inherent ambiguity, and inconsistency of natural language. Consequently, automating semantic reasoning would greatly reduce human effort and enhance the quality and confidence in these cases. However, free-form natural language is not conducive to such automation. Conversely, fully formal notations are impractical to represent assurance arguments.

To that end, we define a balanced approach that presents details in an intuitive and "minimally" formal way. We take a two-step approach, where we first categorically ground the terms used in the descriptions. Then the assurance case is transformed into a logical notation that can be subject to various formal analyses at the back end.

#### Step 1: Objects-Properties-Environment Specification:

Fundamentally, assurance cases consist of blocks or "nodes" that describe the properties or relationships applicable to objects in a certain environment. Leveraging this general structure of description, we independently define the "Properties," "Objects," and "Environment" for each node, allowing us to formally express statements like "Object O satisfies property P in

environment E." For instance, the assertion "Tim is a safe driver in the US" can be deconstructed into "Object = Tim," "Property = safe driver," and "Environment = US." When multiple objects and properties are associated with a node, it becomes necessary to specify their relationships, such as each property is applicable to all objects or each object or a specific object. This relationship is preserved and meticulously converted into appropriate logic predicates.

Currently, our ASCE assurance case authoring tool enables users to explicitly define objects, properties, and environments, along with their relationships to the assurance nodes, in addition to providing natural language descriptions. While the tool currently supports this dual specification, we are actively exploring methods to automatically extract Objects-Properties-Environment information from the descriptions, thereby streamlining the process.

## Step 2: Transformation to Logical Notation

In order to convert the assurance case into a logical notation suitable for the desired semantic analysis, we opt for the domain of Logical Programming (LP). This choice is based on the fact that each concept in Assurance 2.0 and the intended analysis can be readily mapped to corresponding concepts in LP. The mapping between these concepts is illustrated in Table 1.

Assurance 2.0 Concept	Mapping to LP
Top-level Claim	In the assurance case, the top-level claim aligns with the concept of <i>queries</i> in LP, which can be subjected to logical entailment analysis.
Sub-Claims and Side-Claims	Sub-claims and side claims are analogous to <b>predicates</b> in LP serving the purpose of capturing the relationships between objects.
Argument	Arguments that establish relationships among sets of objects-properties-environment can be equated to <b>rules</b> in LP where logical implications are utilized to describe these relationships.
Evidence	Evidence artifacts that establish the truth about the system are <b>facts</b> in LP
Defeater	Defeaters are counter-claims to provide support for not believing that claim, which is the same as <b>negation</b> in formal logic
Binding	theory definitions require the specification of variables for objects and the environment, that will be subsequently instantiated with terms when applied to a specific assurance case. This process aligns with the concept of <b>substitution</b> in LP, which involves dynamically mapping variables to terms.
Reasoning	Assurance case reasoning revolves around demonstrating that a top-level claim is entailed based on its arguments and evidence. This directly corresponds to <b>proofs</b> in logic, where the validity or truth of a claim is through logical reasoning.

**Table 1**

Concept Mapping between Assurance 2.0 and LP

By utilizing the Objects-Properties-Environment specification provided in each assurance case node, and based on the above concept mapping, the ASCE tool automatically translates the case into predicates in Prolog, a widely used logical programming language. The transformation process follows the guidelines presented in Table 2. In this table, the term 'ClaimPredicate' refers to a Prolog predicate represented as the claim([O], [P], [E]), where [O], [P], and [E] represent comma-separated lists of Objects, Properties, and Environments associated with each assurance node. As we formalize the properties, the object-property relationships specified during the

definition of objects and properties are preserved. This preservation ensures precision in the analysis. Consequently, the 'PropertyList' is derived from the ClaimPredicate based on the specified object-property relationship, as enumerated in Figure 3. Furthermore, to maintain the structure of the case during export and ensure traceability, certain metadata (such as node identifiers, descriptions, etc.) is also included within the exported predicates.

Node	Prolog Translation
Claim	claimStmt('Claim_ID', Application, ClaimPredicate, 'Description') :- [claimStmt evidenceStmt side_ClaimStmt], ClaimPredicate, theory ('Theory_ID', Application, Claim). theory ('Theory_ID', Application, Claim).
Evidence	evidenceStmt('Evidence_ID', Application, ClaimPredicate, 'Artefact', 'URI') :- ClaimPredicate. ClaimPredicate :- PropertyList.
Side Claim	side_ClaimStmt('Side_Claim_ID', Application, ClaimPredicate, 'Justification') :- [claimStmt evidenceStmt side_ClaimStmt], ClaimPredicate. ClaimPredicate :- PropertyList.
Defeater	false :- defeater('Defeater_ID', defeats('Node_ID'), [O],[P],[E]). defeater('Defeater_ID', defeats('Node_ID'), [O],[P],[E]) :- PropertyList.

**Table 2**  
Assurance 2.0 Node Mapping to Prolog

Relationship	Property List
Each property applies to all objects	$P(X, Y), Q(X, Y)$
Each property applies to each object	$P(X), P(Y), Q(X), Q(Y)$
Each property applies to a specific object	$P(X), Q(Y)$

**Figure 3:** Object-Property Relationships

When ASCE tool exports the assurance case as Prolog predicates, to methodize the analysis, they are categorically saved in separate files: (i) the top-level claim is saved as a query, (ii) the negation of the top-level claim is saved as a negative query, (iii) the body of the assurance case – arguments, sub-claims, side-claims, and evidence – are saved together as rules and facts, (iv) theory definitions are saved separately, while their reference to applications are saved along with the body of the assurance case, and (v) defeaters are saved as integrity constraints that counter the claims. In the following section, we elaborate on various formal analyses.

## 4. Semantic Reasoning

The fundamental reason to translate the assurance case into Prolog predicates is to allow semantic analysis, which is otherwise not possible. In particular, our interest is to rigorously analyze the assurance case for the various properties that include but are limited to the following:

- *Indefeasibly Justified*: Possession of adequate arguments and evidence that logically entails the top-level claim, and no unresolved defeaters remain.
- *Theory Application Correctness*: Guarantee of correctness of application of theories
- *Property-Object-Environment Definition Consistency*: Absence of contradictions in the property-object-environment definitions.
- *Evidence Validity and Adequacy*: Existence and appropriateness of all evidence to support their respective claim
- *Completeness*: Completeness with respect to the assurance case covering the domain of objects, properties, and environments defined for the system in consideration
- *Harmonious Coexistence of Theories*: Ability among theories (definitions and their applications) to coexists within the same assurance case without causing logical violations.

In order to perform the aforementioned analysis of Prolog predicates, we leverage the capabilities of s(CASP), a logic programming solver. Our choice of s(CASP) is based on several reasons. Firstly, s(CASP) supports *automated commonsense reasoning*, which enables inferences to be drawn from a set of predicates or knowledge in a manner similar to human reasoning. This feature is particularly well-suited for reasoning about assurance cases. Moreover, s(CASP) facilitates *deductive and abductive reasoning*, which is essential for proving whether a top-level claim can be deduced based on the arguments, evidence, and assumptions in assurance cases. Also, s(CASP) supports reasoning about *global constraints* and invariants, making it capable of analyzing scenarios that violate these constraints. Additionally, the top-down solving strategy employed by s(CASP) generates concise, human-understandable *justifications* [11]. These justifications play a crucial role in precisely identifying the reasons for assurance failure and resolving concerns. Furthermore, s(CASP) offers support for various forms of negation, which have diverse applications in assurance reasoning. For instance, we currently utilize *default negation* to derive detailed justifications for why the top-level claim cannot be proven. Additionally, we are exploring the use of *Negation as Failure*, which involves deriving negative conclusions from the absence of positive information, to automatically identify defeaters. Finally, s(CASP) possesses the ability to perform *non-monotonic reasoning*, allowing for the revision of conclusions in light of new information. This feature is particularly valuable for incrementally assessing and improving the strength of assurance cases during the authoring process.

In essence, using s(CASP), the verification of the correctness of the assurance case involves executing the query along with the body of the assurance case and defeaters. A successful execution results in the display of a “model,” which essentially is a detailed explanation as to why the top-level claim is entailed. Conversely, if the query fails, no model is returned. In such cases, executing the negated query allows retrieving an explanation for the cause of the failure and its specific location. The returned model is presented in a tree structure, facilitating the understanding of the trace of what was entailed or failed. Failures typically occur due to the violation of one or more of the analyzed properties listed below.

### **Indefeasibly Justified**

This semantic property is fundamental to an assurance case. Possessing this property implies: (a) The top-level claim is sufficiently supported by well-founded arguments and evidence, ensuring

justification. (b) There are no unresolved defeaters that could potentially alter the decision regarding the top-level claim, called indefeasibility. When the positive query is successfully executed in s(CASP) and an explanation is provided, it signifies that the assurance case indeed possesses this property. Conversely, if the negative query is successfully executed, s(CASP) returns the unresolved defeaters as global integrity violations. An example output illustrating this scenario, for the safe driver example where claim node "C8" is defeated, is shown in Figure 4.

```
Query:
?- claimStmt(C1,application(C1),claim(application(C1),[tim],[safe_driver],[us]),Tim is a safe driver in the US (to take
Jon to the game)).
Answer:
no models
Justification:    
FAILURE to prove the denial 1. :- defeater('D1',defeats('C8'),[tim_0],[not_enough_driving_history_0],[us_0]).
```

**Figure 4:** Execution output showing failure due to defeater in Safe Driver Case

### Theory Application Correctness

As previously explained, a theory consists of properties that are applicable to specific types of objects and environments. In theory nodes, objects, and environments are represented as Prolog variables (names starting with an uppercase letter). During the authoring of assurance cases, "types" of objects and environments, as well as all possible instantiations (constants defined with lowercase letters) are all pre-defined. The author appropriately selects them during theory definition and application. To ensure that incorrect instantiations are avoided, we utilize s(CASP). The analysis involving theories is more complex compared to others since s(CASP) dynamically associates the variables in theories with the constants in theory application.

When utilizing theories in the analysis process with s(CASP), it involves verifying the correctness of theory application based on two key factors: (a) Ensuring that the constants chosen as objects and environments align with the respective instantiations defined in the theory. (b) Validating that the properties specified in the theory nodes correspond to the properties in the corresponding application nodes. For instance, let's consider the scenario depicted in the "Theory of Safe Driver" definition and its application in Figure 2. A correct application of this theory will yield a comprehensive justification tree, exemplified in Figure 5. However, if there were any inaccuracies in specifying the properties or objects, executing the negated query will result in a justification that can be used to trace the error and identify the issue at hand.

### Property-Object-Environment Definition Consistency

Inconsistencies within the specifications of properties, objects, and environments in the nodes of an assurance case can pose a problem as they may lead to flawed validation of assurance arguments. While logical solvers can easily identify conflicts between predicates and their negations in logical representations, conflicts based on natural language words are not as straightforward to detect. For example, in natural language, words like 'good driver' and 'bad



```

Query:
?- claimStmnt(C7,application(C7),claim(application(C7),[tim,tim_driver_license,tim_car,safe_driving_factor_list],[safe_driver],[us]),Application of the theory. Tim is a safe driver in the US).

Answer:
yes

Model:
{ claimStmnt(C7,application(C7),claim(application(C7),[tim,tim_driver_license,tim_car,safe_driving_factor_list],[safe_driver],[us]),Application of the theory. Tim is a safe driver in the US), claimStmnt(C15,application(C7),claim(application(C7),[tim],[satisfies_legal_requirements_for_driving],[us]),X has satisfied all legal requirements for driving), evidenceStmnt(E8,application(C7),fact(application(C7),[tim_driver_license],[valid],[us]),X Driver License.none), fact(application(C7),[tim_driver_license],[valid],[us]), claim(application(C7),[tim],[satisfies_legal_requirements_for_driving],[us]), claimStmnt(C10,application(C7),claim(application(C7),[tim],[has_good_reputation],[us]),X has a good reputation for safe driving), evidenceStmnt(E6,application(C7),fact(application(C7),[tim,tim],[testimonials_of_good_reputation],[us]),Testimonials about X good driving habits by other safe drivers.none), fact(application(C7),[tim,tim],[testimonials_of_good_reputation],[us]), claim(application(C7),[tim],[has_good_reputation],[us]), claimStmnt(C12,application(C7),claim(application(C7),[tim],[life_normal],[us]),Nothing untoward is going on in X s life that may cause him to drive less well than usual ), ....

Justification: Expand All +1 -1 Collapse All

- ▶ claimStmnt(C7,application(C7),claim(application(C7),[tim,tim_driver_license,tim_car,safe_driving_factor_list],[safe_driver],[us]),Application of the theory. Tim is a safe driver in the US) :-
  - ▶ claimStmnt(C15,application(C7),claim(application(C7),[tim],[satisfies_legal_requirements_for_driving],[us]),X has satisfied all legal requirements for driving) :-
  - ▶ claimStmnt(C10,application(C7),claim(application(C7),[tim],[has_good_reputation],[us]),X has a good reputation for safe driving) :-
  - ▶ claimStmnt(C12,application(C7),claim(application(C7),[tim],[life_normal],[us]),Nothing untoward is going on in X s life that may cause him to drive less well than usual ) :-
  - ▶ claimStmnt(C11,application(C7),claim(application(C7),[tim_car],[doesnt_pose_danger],[us]),X s car does not pose any special danger) :-
  - ▶ claimStmnt(C16,application(C7),claim(application(C7),[tim],[no_accident_history],[us]),X has not been in an accident) :-
  - ▶ side_ClaimStmnt(S2,application(C7),claim(application(C7),[safe_driving_factor_list],[is_historically_safe],[us]),Based on history of safe drivers, satisfying the following subclaims entail the top claim.) :-
    - claim(application(C7),[tim,tim_driver_license,tim_car,safe_driving_factor_list],[safe_driver],[us]),
    - theory(C9,application(C7),claim(application(C7),[tim,tim_driver_license,tim_car,safe_driving_factor_list],[safe_driver],[us])) :-

```

**Figure 5:** Snippet of the Execution output of consistent application of Theory of Safe Driver

driver’ are considered opposites, but in logic, they may not be. However, from an assurance case perspective, it is crucial to semantically identify and report such opposites and conflicts. To address this issue, we propose the introduction of specialized, domain-specific rules. For instance, in the safe driver assurance case, conflicts among property specifications such as ‘good driving habits’ for Tim in one node, and ‘bad driving habits’ for Tim in another, can be detected and reported by s(CASP) by incorporating a rule that ‘bad driving habits’ for any person is the negation of ‘good driving habits’ for that person.

### Evidence Validity and Adequacy

The provision of appropriate evidence is crucial to assurance cases. While ASCE tool structurally detects claims with missing evidence nodes, incorrect or inadequate evidence within the evidence node is essentially a semantic check. By introducing domain-specific rules relevant to evidence will help automatically detect and report the issues. For instance, by introducing a rule that requires testimonials of ‘good driving habit’ by two or more safe drivers (E1 node in Figure 1), s(CASP) can report inadequacies in the provided evidence. Since s(CASP) can perform substitution, these rules can be defined using Prolog variables and reused for other assurance cases in the domain. We are currently investigating generic rules that can be broadly applied to a class of evidence. For instance, test-case artifacts should also always have an associated coverage analysis report.

### Completeness

In general, completeness is the state of encompassing everything that is needed. Given a pre-defined global set of objects-properties-environment for the system in consideration, and by defining a rule such as ‘domain completeness’ that requires every property-object-environment is used at least once in a claim or evidence node, we can leverage s(CASP) to analyze the

completeness of assurance case. For instance, in the safe driver case, if there are multiple types of vehicle inspection records available in the domain, while Tim's case only accounts for one of them, s(CASP) can help report the incompleteness; this can either be resolved by adding the evidence to the case or removing the additional one from the domain.

Completeness refers to the state of encompassing all the necessary elements. To assess the completeness of an assurance case, we can utilize the pre-defined global set of objects, properties, and environments for the system under consideration. By defining say, a "domain completeness" rule that required every property-object-environment combination to be used at least once in a claim or evidence node, and employing s(CASP) for analysis, we can evaluate the completeness of the assurance case. For example, in the case of safe driver assurance, if there are multiple types of vehicle inspection records available in the domain, but Tim's case only included one of them, s(CASP) can be leveraged to precisely identify the incompleteness.

### **Harmonious Coexistence of Theories**

Theories are reusable assurance fragments that can be independently specified, and 'pre-certified' and be applied to different cases. when constructing assurance cases, more than one theory may be applied to an assurance case. However, when multiple theories are linked and applied within an assurance case, there is a risk of conflicting properties among their definitions or in their concrete applications. We are currently exploring approaches to define novel rules and will allow s(CASP) to check for the presence of conflicts, or determine that the theories can harmoniously coexist within that assurance case.

The realm of semantic analysis offers a wide range of possible analyses. As part of our ongoing work, we are exploring valuable and powerful properties to analyze. Automating these analyses will provide valuable insights and relieve humans from repetitive tasks, leading to improved decision-making regarding assurance cases.

## **5. Conclusion**

Our Assurance 2.0 framework is designed to facilitate reasoning and communication regarding the behavior and trustworthiness of engineered systems, with the ultimate goal of achieving certification. In this paper, we have presented our approach to augment it with semantic analysis capabilities. We begin by systematically converting the Assurance 2.0 case into Prolog predicates and utilize s(CASP) to conduct various semantic analyses. We have conducted evaluations on complex systems, and the preliminary results show promise. As a result, we are further enhancing the capabilities of our tools and techniques to enable more in-depth analysis of the semantics of assurance cases.

## **6. Acknowledgement**

This work is supported by DARPA Automated Rapid Certification Of Software (ARCOS) program under agreement number FA875020C0512 for Consistent Logical Automated Reasoning for Integrated System Software Assurance (CLARISSA) project. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

## References

- [1] C. M. Holloway, P. J. Graydon, DOT/FAA/TC-17/67 : Explicate 78: Assurance Case Applicability to Digital Systems, Technical Report, FAA, 2018.
- [2] M. Maksimov, N. L. Fung, S. Kokaly, M. Chechik, Two decades of assurance case tools: a survey, in: Computer Safety, Reliability, and Security: SAFECOMP 2018 Workshops, Sweden, September 18, 2018, Proceedings 37, Springer, 2018, pp. 49–59.
- [3] R. Bloomfield, J. Rushby, Assurance 2.0: A manifesto, Preprint arXiv:2004.10474 (2020).
- [4] A. LLP, Assurance and safety case environment (asce), Accessed on June 20, 2023. URL: <http://www.adelard.com/asce>.
- [5] J. Arias, M. Carro, E. Salazar, K. Marple, G. Gupta, Constraint Answer Set Programming without Grounding, TPLP 18 (2018) 337–354.
- [6] G. Gupta, E. Salazar, S. C. Varanasi, K. Basu, J. Arias, F. Shakerin, R. Min, F. Li, H. Wang, Automating commonsense reasoning with asp and s(casp) (2022).
- [7] E. Heavner, C. M. Holloway, Assurance arguments for the non-graphically-inclined: Two approaches, Technical Report, 2017.
- [8] R. Bloomfield, K. Netkatchova, J. Rushby, Defeaters and Eliminative Argumentation in CLARISSA (To be published), Technical Report, SRI, 2023.
- [9] S. Varadarajan, R. Bloomfield, J. Rushby, G. Gupta, A. Murugesan, R. Stroud, K. Netkachova, I. H. Wong, Clarissa: Foundations, tools & automation for assurance case, Accepted at 42nd AIAA/IEEE Digital Avionics Systems Conference (2023).
- [10] J. Rushby, R. Bloomfield, Assessing confidence with assurance 2.0, arXiv preprint arXiv:2205.04522 (2022).
- [11] J. Arias, M. Carro, Z. Chen, G. Gupta, Justifications for goal-directed constraint answer set programming, in: Proceedings 36th International Conference on Logic Programming (Technical Communications), volume 325 of *EPTCS*, 2020, pp. 59–72.